Learn Scala fast
with small, easy lessons

ALVIN ALEXANDER

Hello, Scala

Alvin Alexander

Learn Scala fast
with small, easy lessons

Copyright

Hello, Scala

Version 1.0, published September 3, 2018

Book errata can be found at alvinalexander.com/hello-scala[2]

Other books by Alvin Alexander:

- Functional Programming, Simplified[3]
- Scala Cookbook[4]

[1] https://alvinalexander.com
[2] https://alvinalexander.com/hello-scala
[3] https://kbhr.co/hs-fps
[4] https://kbhr.co/hs-cook

Contents

1
Preface

Have you ever fallen in love with a programming language? I still remember when I first saw the book, *The C Programming Language*, and how I fell in love with its simple syntax and the ability to interact with a computer at a low level. In 1996 I loved Java because it made OOP simple. A few years later I found Ruby and loved its elegance.

Then in 2011 I was aimlessly wandering around Alaska and stumbled across the book, *Programming in Scala*, and I was stunned by its remarkable marriage of Ruby and Java:

- The syntax was as elegant and concise as Ruby
- It feels dynamic, but it's statically typed
- It compiles to class files that run on the JVM
- You can use the thousands of Java libraries in existence with your Scala code

In the first edition of the book, *Beginning Scala*, David Pollak states that Scala will change the way you think about programming, and *that's a good thing*. Learning Scala has not only been a joy, but it's led me on a journey to appreciate concepts like modular programming, immutability, referential transparency, and functional programming, and most importantly, how those ideas help to dramatically reduce bugs in my code.

1.1 Is Scala DICEE?

DICEE is an acronym that was coined by Guy Kawasaki, who became famous as a developer evangelist for the original Apple Macintosh team. He says that great products are DICEE, meaning Deep, Indulgent, Complete, Elegant, and Emotive:

- *Deep*: The product doesn't run out of features and functionality after a few weeks of use. Its creators have anticipated what you'll need once you come up to speed. As your demands get more sophisticated, you won't need a different product.
- *Indulgent*: A great product is a luxury. It makes you feel special when you buy it (and use it).

- *Complete*: A great product is more than a physical thing. Documentation counts. Customer service counts. Tech support counts.
- *Elegant*: A great product has an elegant user interface. Things work the way you'd think they would. A great product doesn't fight you, it enhances you.
- *Emotive*: A great product incites you to action. It is so deep, indulgent, complete, and elegant that it compels you to tell other people about it. You're bringing the good news to help others, not yourself.

Two years after discovering Scala — way back in 2013 — I came to the conclusion that it meets the definition of DICEE, and I think it's just as true today:

- Scala is *deep*: After all these years I continue to learn new techniques to write better code.
- Scala is *indulgent*: Just like Ruby, I feel special and fortunate to use a language that's so well thought out.
- Scala is *complete*: The documentation is excellent, terrific frameworks exist, and the support groups are terrific.
- Scala is *elegant*: Once you grasp its main concepts you'll fall in love with how it works just like you expect it to.
- Scala is *emotive*: Everyone who works with it wants to tell you how special it is. Myself, I had never written a programming book in my life, but by 2012 I was eagerly mailing people at O'Reilly to tell them how much I wanted to write the Scala Cookbook[1].

As I write this book many years later I hope to share not just the nuts and bolts of the Scala language, but also its elegance and the joy of using it.

Alvin Alexander
https://alvinalexander.com

[1] http://kbhr.co/hs-cook

2

Prelude: A Taste of Scala

My hope in this book is to demonstrate that Scala[1] is a beautiful, modern, expressive programming language. To get started with that, in this first chapter I jump right in and provide a whirlwind tour of Scala's main features in about ten pages. After the tour, the book continues with a more traditional "Getting Started" chapter.

> In this book I assume that you've used a language like C or Java before, and are ready to see a series of Scala examples to get a feel for the language. Although it's not 100% necessary, it will also help if you've already downloaded and installed Scala[2] so you can test the examples as you go along.

2.1 Overview

Before we jump into the examples, here are a few important things to know about Scala:

- It's a high-level language
- It's statically typed
- Its syntax is concise but still readable — we call it *expressive*
- It supports the object-oriented programming (OOP) paradigm
- It supports the functional programming (FP) paradigm
- It has a sophisticated type inference system
- It has *traits*, which are a combination of interfaces and abstract classes that can be used as mixins, and support a modular programming style
- Scala code results in *.class* files that run on the Java Virtual Machine (JVM)
- It's easy to use Java libraries in Scala

[1] http://scala-lang.org/
[2] https://www.scala-lang.org/download/

2.2 Hello, world

Ever since the book, The C Programming Language[3], it's been a tradition to begin programming books with a "Hello, world" example, and not to disappoint, this is one way to write that example in Scala:

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

After you save that code to a file named *Hello.scala* you can compile it with `scalac`:

```
$ scalac Hello.scala
```

`scalac` is just like `javac`, and that command creates two files:

- Hello$.class
- Hello.class

These are the same ".class" bytecode files you create with `javac`, and they're ready to run in the JVM. You run the `Hello` application with the `scala` command:

```
$ scala Hello
```

I share more "Hello, world" examples in the lessons that follow, so I'll leave that introduction as is for now.

2.3 The Scala REPL

The Scala REPL ("Read-Evaluate-Print-Loop") is a command-line interpreter that you use as a "playground" area to test your Scala code. I introduce it early here so you can use it with the code examples that follow.

To start a REPL session, just type `scala` at your operating system command line, and you'll see something like this:

[3] http://amzn.to/2CsDmYa

```
$ scala
Welcome to Scala 2.12.4 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_131).
Type in expressions for evaluation. Or try :help.

scala> _
```

Because the REPL is a command-line interpreter, it sits there waiting for you to type something. Inside the REPL you type Scala expressions to see how they work:

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

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

As those examples show, after you type an expression, the REPL shows the result of the expression on the line following the prompt.

2.4 Two types of variables

Scala has two types of variables:

- `val` is an immutable variable — like `final` in Java — and should be preferred
- `var` creates a mutable variable, and should only be used when there is a specific reason to use it

Examples:

```
val x = 1   //immutable
var y = 0   //mutable
```

2.5 Implicit and explicit variable types

In Scala, you typically create variables without declaring their type:

```
val x = 1
val s = "a string"
val p = new Person("Kimberly")
```

This is known as an *implicit type* style.

You can also *explicitly* declare a variable's type, but that's not usually necessary:

```
val x: Int = 1
val s: String = "a string"
val p: Person = new Person("Kimberly")
```

Because showing a variable's type like that isn't necessary — and actually feels needlessly verbose — I rarely use this explicit syntax. (I explain *when* I use it later in the book.)

2.6 Testing object equality

In Scala everything is an object, and you use == to test object equality:

```
val a = "foo"
val b = "foo"
a == b  // true

case class Store(name: String)
val a = Store("Flowers By Hala")
val b = Store("Flowers By Hala")
a == b  // true
```

2.7 Control structures

Here's a quick tour of Scala's control structures.

2.7.1 if/else

Scala's if/else control structure is similar to other languages:

```
if (test1) {
    doA()
} else if (test2) {
    doB()
```

```
} else if (test3) {
    doC()
} else {
    doD()
}
```

The if/else construct is an *expression* that returns a value, so you can also use it as a ternary operator:

```
val x = if (a < b) a else b
```

2.7.2 match expressions

Scala has a `match` expression, which in its most basic use is like a Java `switch` statement:

```
val result = i match {
    case 1 => "one"
    case 2 => "two"
    case _ => "not 1 or 2"
}
```

As shown, the `_` case is a catch-all case that handles any pattern that isn't matched by the previous `case` statements.

The `match` expression isn't limited to just integers, it can be used with any data type. Here it's used with a `Boolean` variable named `bool`:

```
val booleanAsString = bool match {
    case true => "true"
    case false => "false"
}
```

Here's an example of `match` being used as the body of a method, and matching against many different types:

```
def getClassAsString(x: Any): String = x match {
    case s: String => s + " is a String"
    case i: Int => "Int"
    case f: Float => "Float"
    case l: List[_] => "List"
    case p: Person => "Person"
    case _ => "Unknown"
}
```

Powerful match expressions are a big feature of Scala.

2.7.3 try/catch

Scala's try/catch control structure lets you catch exceptions. It's similar to Java, but its syntax is consistent with `match` expressions:

```
try {
    writeToFile(text)
} catch {
    case fnfe: FileNotFoundException => println(fnfe)
    case ioe: IOException => println(ioe)
}
```

2.7.4 for loops and expressions

Scala `for` loops — which I refer to in this book as *for-loops* — look like this:

```
for (arg <- args) println(arg)

// "x to y" syntax
for (i <- 0 to 5) println(i)

// "x to y by" syntax
for (i <- 0 to 10 by 2) println(i)
```

You can also add the `yield` keyword to for-loops to create *for-expressions* that yield a result. Here's a for-expression that doubles each value in the sequence 1 to 3:

```
val x = for (i <- 1 to 3) yield i * 2    //yields Vector(2, 4, 6)
```

Here's another for-expression that iterates over a list of strings:

```
val fruits = List("apple", "banana", "lime", "orange")

val fruitLengths = for {
    f <- fruits
    if f.length > 4
} yield f.length
```

Because Scala code generally just makes sense, I'll imagine that you can guess how that code works, even if you've never seen a for-expression or Scala `List` until now.

> Scala also has `while` and `do/while` loops, but I rarely use them.

2.8 Classes

Here's an example of a Scala class:

```
class Person(var firstName: String, var lastName: String) {
    def printFullName() {
        println(s"$firstName $lastName")
    }
}
```

Here's an example of how to use that class:

```
val p = new Person("Julia", "Kern")
println(p.firstName)    //Julia

p.lastName = "Manes"
p.printFullName()       //Julia Manes
```

Notice that there's no need to create "get" and "set" methods to access the fields in the class.

As a more complicated example, here's a `Pizza` class that you'll see later in the book:

```
class Pizza (
    var crustSize: CrustSize,
    var crustType: CrustType,
    val toppings: ArrayBuffer[Topping]
) {
    def addTopping(t: Topping): Unit = { toppings += t }
    def removeTopping(t: Topping): Unit = { toppings -= t }
    def removeAllToppings(): Unit = { toppings.clear() }
}
```

In that code, an `ArrayBuffer` is like Java's `ArrayList`. I don't show the `CrustSize`, `CrustType`, and `Topping` classes, but I suspect that you can understand how that code works without needing to see those classes.

2.9 Scala methods

Just like other OOP languages, Scala classes have methods, and this is what Scala's method syntax looks like:

```
def sum(a: Int, b: Int): Int = a + b
def concatenate(s1: String, s2: String): String = s1 + s2
```

You don't have to declare a method's return type, so it's perfectly legal to write those two methods like this, if you prefer:

```
def sum(a: Int, b: Int) = a + b
def concatenate(s1: String, s2: String) = s1 + s2
```

This is how you call those methods:

```
val x = sum(1, 2)
val y = concatenate("foo", "bar")
```

There are more things you can do with methods, such as providing default values for method parameters, but that's a good start for now.

2.10 Traits

Traits in Scala are a lot of fun, and they also let you break your code down into small, modular units. To demonstrate traits, here's an example from later in the book. Given these three traits:

```
trait Speaker {
    def speak(): String  // has no body, so it's abstract
}

trait TailWagger {
    def startTail(): Unit = { println("tail is wagging") }
    def stopTail(): Unit = { println("tail is stopped") }
}

trait Runner {
    def startRunning(): Unit = { println("I'm running") }
    def stopRunning(): Unit = { println("Stopped running") }
}
```

You can create a `Dog` class that extends all of those traits while providing behavior for the `speak` method:

```
class Dog(name: String) extends Speaker with TailWagger with Runner {
    def speak(): String = "Woof!"
}
```

Similarly, here's a `Cat` class that shows how to override trait methods:

```
class Cat extends Speaker with TailWagger with Runner {
    def speak(): String = "Meow"
    override def startRunning(): Unit = { println("Yeah ... I don't run") }
    override def stopRunning(): Unit = { println("No need to stop") }
}
```

If that code makes sense — great, you're comfortable with traits! If not, don't worry, I explain them in detail later in the book.

2.11 Collections classes

Based on my own experience, here's an important rule to know about Scala's collections classes:

> If you're coming to Scala from Java, forget what you know about Java's collections classes, and use the Scala collections classes.

You *can* use the Java collections classes in Scala, and I did so for several months, but when you do that you're slowing down your own learning process. The Scala collections classes offer many powerful methods that you'll want to start using ASAP.

2.11.1 Populating lists

There are times when it's helpful to create sample lists that are populated with data, and Scala offers many ways to populate lists. Here are just a few:

```
val nums = List.range(0, 10)
val nums = 1 to 10 by 2 toList
val letters = ('a' to 'f').toList
val letters = ('a' to 'f') by 2 toList
```

2.11.2 Sequence methods

While there are many sequential collections classes you can use, let's look at some examples of what you can do with the Scala `List` class. Given these two lists:

```
val nums = (1 to 10).toList
val names = List("joel", "ed", "chris", "maurice")
```

This is the `foreach` method:

```
scala> names.foreach(println)
joel
ed
chris
maurice
```

Here's the `filter` method, followed by `foreach`:

```
scala> nums.filter(_ < 4).foreach(println)
1
2
3
```

Here are some examples of the `map` method:

```
scala> val doubles = nums.map(_ * 2)
doubles: List[Int] = List(2, 4, 6, 8, 10, 12, 14, 16, 18, 20)

scala> val capNames = names.map(_.capitalize)
capNames: List[String] = List(Joel, Ed, Chris, Maurice)

scala> val lessThanFive = nums.map(_ < 5)
lessThanFive: List[Boolean] = List(true, true, true, true, false, false, false,
false, false, false)
```

Even though I didn't explain it, you can see how `map` works: It applies an algorithm you supply to every element in the collection, returning a new, transformed value for each element.

If you're ready to see one of the most powerful collections methods, here's `reduce`:

```
scala> nums.reduce(_ + _)
res0: Int = 55

scala> nums.reduce(_ * _)
res1: Int = 3628800
```

Even though I didn't explain `reduce`, you can guess that the first example yields the sum of the numbers in `nums`, and the second example returns the product of all those numbers.

There are many (many!) more methods available to Scala collections classes, but hopefully this gives you an idea of their power.

There's so much power in the Scala collections class, I spend over 100 pages discussing them in the Scala Cookbook[4].

2.12 Tuples

Tuples let you put a heterogenous collection of elements in a little container. Tuples can contain between two and 22 variables, and they can all be different types. For example, given a `Person` class like this:

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

You can create a tuple that contains three different types like this:

```
val t = (11, "Eleven", new Person("Eleven"))
```

You can access the tuple values by number:

```
t._1    // 11
t._2    // "Eleven"
t._3    // Person("Eleven")
```

Or assign the tuple fields to variables:

```
val(num, string, person) = (11, "Eleven", new Person("Eleven"))
```

I don't overuse tuples, but they're nice for those times when you need to put a little "bag" of things together for a little while.

2.13 What I haven't shown

While that was a whirlwind introduction to Scala in about ten pages, there are *many* features I haven't shown yet, including:

- Strings and built-in numeric types
- Packaging and imports

[4] http://kbhr.co/hs-cook

- How to use Java collections classes in Scala
- How to use Java libraries in Scala
- How to build Scala projects
- How to perform unit testing in Scala
- How to write Scala shell scripts
- Maps, Sets, and other collections classes
- Object-oriented programming
- Functional programming
- Concurrency with Futures and Akka
- More …

If you like what you've seen so far, I hope you'll like the rest of the book.

2.14 A bit of background

Scala was created by Martin Odersky[5], who studied under Niklaus Wirth[6], who created Pascal and several other languages. Mr. Odersky is one of the co-designers of Generic Java, and is also known as the "father" of the `javac` compiler.

[5] https://en.wikipedia.org/wiki/Martin_Odersky
[6] https://en.wikipedia.org/wiki/Niklaus_Wirth

3

The Scala Programming Language

The name *Scala* comes from the word *scalable*, and true to that name, it's used to power the busiest websites in the world, including Twitter, Netflix, Tumblr, LinkedIn, Foursquare, and many more.

Here are a few more nuggets about Scala:

- It's a modern programming language created by Martin Odersky[1], and influenced by Java, Ruby, Standard ML, Pizza[2], Lisp, Haskell, OCaml, and others.
- It's a high-level language.
- It's statically typed.
- It has a sophisticated type inference system.
- It's syntax is concise but still readable — we call it *expressive*.
- It's a pure object-oriented programming (OOP) language. Every variable is an object, and every "operator" is a method.
- It's also a functional programming (FP) language, so functions are also variables, and you can pass them into other functions. You can write your code using OOP, FP, or combine them in a hybrid style.
- Scala source code compiles to ".class" files that run on the JVM.
- Scala also works extremely well with the thousands of Java libraries that have been developed over the years.
- The Akka library[3] provides an *Actors* API, which was originally based on the actors concurrency model built into Erlang.
- The Play Framework[4] is a lightweight, stateless, web development framework

[1] https://twitter.com/odersky
[2] https://en.wikipedia.org/wiki/Pizza_(programming_language)
[3] https://akka.io
[4] https://www.playframework.com/

that's built with Scala and Akka. (In addition to Play there are several other popular web frameworks.)

- A great thing about Scala is that you can be productive with it on Day 1, but it's also a deep language, so as you go along you'll keep learning, and finding newer, better ways to write code. It's said that Scala will change the way you think about programming (and that's a good thing).
- Of all of Scala's benefits, what I like best is that it lets you write concise, readable code. The time a programmer spends reading code compared to the time spent writing code is said to be at least a 10:1 ratio, so writing code that's *concise and readable* is a big deal. Because Scala has these attributes, programmers say that it's *expressive*.

4
Hello, World

Let's look at the "Hello, world" example again:

```
object Hello {
    def main(args: Array[String]) {
        println("Hello, world")
    }
}
```

Using a text editor, save that source code in a file named *Hello.scala*. After saving it, run this `scalac` command at your command line prompt to compile it:

```
$ scalac Hello.scala
```

`scalac` is just like `javac`, and that command creates two new files:

- Hello$.class
- Hello.class

These are the same types of ".class" bytecode files you create with `javac`, and they're ready to work with the JVM.

Now you can run the `Hello` application with the `scala` command:

```
$ scala Hello
```

4.1 Discussion

Here's the original source code again:

```
object Hello {
    def main(args: Array[String]) {
        println("Hello, world")
    }
}
```

Here's a short description of that code:

- It defines a method named `main` inside a Scala `object` named `Hello`
- An `object` is similar to a `class`, but you specifically use it when you want a singleton object
    - If you're coming to Scala from Java, this means that `main` is just like a `static` method (I write more on this later)
- `main` takes an input parameter named `args` that is a string array
- `Array` is a class that wraps the Java `array` primitive

That Scala code is pretty much the same as this Java code:

```
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello, world")
    }
}
```

4.2 Going deeper: Scala creates *.class* files

As I mentioned, when you run the `scalac` command it creates *.class* JVM bytecode files. You can see this for yourself. As an example, run this `javap` command on the *Hello.class* file:

```
$ javap Hello.class
Compiled from "Hello.scala"
public final class Hello {
  public static void main(java.lang.String[]);
}
```

As that output shows, the `javap` command reads that *.class* file just as if it was created from Java source code. Scala code runs on the JVM and can use existing Java libraries, and both are terrific benefits for Scala programmers.

4.2.1 Peaking behind the curtain

To be more precise, what happens is that Scala source code is initially compiled to Java source code, and then that source code is turned into bytecode that works with the JVM. I explain some details of this process in the Scala Cookbook[1].

If you're interested in more details on this process right now, see the "Using scalac print options" section of my How to disassemble and decompile Scala code[2] tutorial.

[1] http://kbhr.co/hs-cook
[2] http://kbhr.co/hs-scalac

5
Hello, World (Version 2)

While that first "Hello, World" example works just fine, Scala provides a way to write applications more conveniently. Rather than including a `main` method, your `object` can just extend the `App` trait, like this:

```
object Hello2 extends App {
    println("Hello, world")
}
```

If you save that code to *Hello.scala*, compile it with `scalac` and run it with `scala`, you'll see the same result as the previous lesson.

What happens here is that the `App` trait has its own `main` method, so you don't need to write one. I'll show later on how you can access command-line arguments with this approach, but the short story is that it's easy: they're made available to you in a string array named `args`.

> A Scala `trait` is similar to an abstract class in Java. More accurately, it's a combination of an abstract class and an interface — more on this later!

5.1 Extra credit

If you want to see how command-line arguments work when your object extends the `App` trait, save this source code in a file named *HelloYou.scala*:

```
object HelloYou extends App {
    if (args.size == 0)
        println("Hello, you")
    else
        println("Hello, " + args(0))
}
```

Then compile it with `scalac`:

```
scalac HelloYou.scala
```

Then run it with and without command-line arguments. Here's an example:

```
$ scala HelloYou
Hello, you

$ scala HelloYou Al
Hello, Al
```

This shows:

- When you extend the `App` trait, command-line arguments are automatically made available to you in a variable named `args`.
- You determine the number of elements in `args` with `args.size` (or `args.length`, if you prefer).
- `args` is an `Array`, and you access `Array` elements as `args(0)`, `args(1)`, etc. Because `args` is an object, you access the array elements with parentheses (not `[]` or any other special syntax).

6

The Scala REPL

The Scala REPL ("Read-Evaluate-Print-Loop") is a command-line interpreter that you use as a playground area to test your Scala code. To start a REPL session just type `scala` at your operating system command line, and you'll see something like this:

```
$ scala
Welcome to Scala 2.12.4 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_131).
Type in expressions for evaluation. Or try :help.

scala> _
```

Because the REPL is a command-line interpreter, it just sits there waiting for you to type something. Once you're in the REPL, you can type Scala expressions to see how they work:

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

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

As those examples show, just type your expressions inside the REPL, and it shows the result of each expression on the line following the prompt.

6.1 Variables are created as needed

If you don't assign the result of your expression to a variable, the REPL automatically creates variables that start with the name `res`. The first variable is `res0`, the second one is `res1`, etc.:

```
scala> 2 + 2
res0: Int = 4
```

```
scala> 3 / 3
res1: Int = 1
```

These are actual variable names that are dynamically created, and you can use them in your expressions:

```
scala> val z = res0 + res1
z: Int = 5
```

You're going to use the REPL a lot in this book, so go ahead and start experimenting with it. Here are a few expressions you can try to see how it all works:

```
val name = "John Doe"
"hello".head
"hello".tail
"hello, world".take(5)
println("hi")
1 + 2 * 3
(1 + 2) * 3
if (2 > 1) println("greater") else println("lesser")
```

While I prefer to use the REPL, there are a couple of other, similar tools you can use:

- The Scala IDE for Eclipse has a Worksheet plugin that lets you do the same things inside your IDE
- IntelliJ IDEA also has a Worksheet
- scalafiddle.io[1] lets you do a similar thing in a web browser (just remember to press "Run")

[1] https://scalafiddle.io/

6.2 More information

For more information on the Scala REPL, see these links:

- The REPL overview on scala-lang.org[2]
- My Getting started with the Scala REPL tutorial[3]

[2] https://docs.scala-lang.org/overviews/repl/overview.html
[3] http://kbhr.co/hs-repl

7

Two Types of Variables

In Java you declare new variables like this:

```
String s = "hello";
int i = 42;
Person p = new Person("Joel Fleischman");
```

Each variable declaration is preceded by its type.

By contrast, Scala has only two types of variables:

- `val` creates an *immutable* variable (like `final` in Java)
- `var` creates a *mutable* variable

This is what variable declaration looks like in Scala:

```
val s = "hello"   // immutable
var i = 42        // mutable
```

Here are some more examples:

```
val p = new Person("Joel Fleischman")
val nums = List(1, 2, 3)
```

Those examples show that the Scala compiler is usually smart enough to infer the variable's data type from the code on the right side of the = sign. This is considered an *implicit* form. You can also *explicitly* declare the variable type if you prefer:

```
val s: String = "hello"
var i: Int = 42
```

In most cases the compiler doesn't need to see those explicit types, but you can add them if you think it makes your code easier to read. I usually use the explicit form

when the type isn't obvious.

> As a practical matter I generally do this when working with complex code, and when using methods in third-party libraries, especially when I don't use the library often or if their method names don't make the type clear. (I show examples of this later in the book.)

7.1 The difference between `val` and `var`

The difference between `val` and `var` is that `val` makes a variable *immutable* — like `final` in Java — and `var` makes a variable *mutable*. Because `val` fields can't vary, some people refer to them as *values* rather than variables.

The REPL shows what happens when you try to reassign a `val` field:

```
scala> val a = 'a'
a: Char = a

scala> a = 'b'
<console>:12: error: reassignment to val
       a = 'b'
         ^
```

That fails with a "reassignment to val" error, as expected. Conversely, you can reassign a `var`:

```
scala> var a = 'a'
a: Char = a

scala> a = 'b'
a: Char = b
```

In Scala the general rule is that you should always use a `val` field unless there's a good reason not to. This simple rule (a) makes your code more like algebra and (b) helps get you started down the path to functional programming, where *all* fields are immutable.

7.2 "Hello, world" with a `val` field

Here's what a "Hello, world" app looks like with a `val` field:

```
object Hello3 extends App {
    val hello = "Hello, world"
    println(hello)
}
```

As before:

- Save that code in a file named *Hello3.scala*
- Compile it with `scalac Hello3.scala`
- Run it with `scala Hello3`

7.3 A note about `val` fields in the REPL

The REPL isn't 100% the same as working with source code in an IDE, so there are a few things you can do in the REPL that you can't do when working on real-world code in a project. One example of this is that you can reassign a `val` field in the REPL, like this:

```
scala> val age = 18
age: Int = 18

scala> val age = 19
age: Int = 19
```

I thought I'd mention that because I didn't want you to see it one day and think, "Hey, Al said `val` fields couldn't be reassigned." They can be reassigned like that, but only in the REPL.

8

The Type is Optional

As I showed in the previous lesson, when you create a new variable in Scala you can *explicitly* declare its type, like this:

```
val count: Int = 1
val name: String = "Alvin"
```

But …

8.1 The explicit form feels verbose

In most cases your code is easier to read when you leave the type off, so the implicit form is preferred. For instance, in this example it's obvious that the data type is `Person`, so there's no need to declare the type on the left side of the expression:

```
val p = new Person("Candy")
```

Indeed, when you put the type next to the variable name, the code feels unnecessarily verbose:

```
val p: Person = new Person("Leo")
```

When creating new variables I *rarely* use that style.

8.2 Use the explicit form when you need to be clear

One place where you'll want to show the data type is when you want to be clear about what you're creating. That is, if you don't explicitly declare the data type, the compiler may make a wrong assumption about what you want to create. Some examples of this are when you want to create numbers with specific data types. I show this in the next lesson.

9

A Few Built-In Types

Scala comes with the standard numeric data types you'd expect. In Scala all of these data types are full-blown objects (not primitive data types).

These examples show how to declare variables of the basic numeric types:

```
val b: Byte = 1
val x: Int = 1
val l: Long = 1
val s: Short = 1
val d: Double = 2.0
val f: Float = 3.0
```

In the first four examples, if you don't explicitly specify a type, the number `1` will default to an `Int`, so if you want one of the other data types — `Byte`, `Long`, or `Short` — you need to explicitly declare those types, as shown. Numbers with a decimal (like 2.0) will default to a `Double`, so if you want a `Float` you need to declare a `Float`, as shown in the last example.

Because `Int` and `Double` are the default numeric types, you typically create them without explicitly declaring the data type:

```
val i = 123   // defaults to Int
val x = 1.0   // defaults to Double
```

The REPL confirms this:

```
scala> val i = 123
i: Int = 123

scala> val x = 1.0
x: Double = 1.0
```

All of those data types have the same data ranges[1] as their Java equivalents.

9.1 BigInt and BigDecimal

For large numbers Scala also includes the types `BigInt` and `BigDecimal`:

```
var b = BigInt(1234567890)
var b = BigDecimal(123456.789)
```

Here's a link for more information about BigInt and BigDecimal[2].

9.2 String and Char

Scala also has `String` and `Char` data types, which I always declare with the implicit form:

```
val name = "Bill"
val c = 'a'
```

[1] http://kbhr.co/hs-data-ranges
[2] http://kbhr.co/hs-bigint

10

Two Notes About Strings

Scala strings have a lot of nice features, but I want to take a moment to highlight two features that I'll use in the rest of this book. The first feature is that Scala has a nice, Ruby-like way to merge multiple strings. Given these three variables:

```
val firstName = "John"
val mi = 'C'
val lastName = "Doe"
```

you can append them together like this, if you want to:

```
val name = firstName + " " + mi + " " + lastName
```

However, Scala provides this more convenient form:

```
val name = s"$firstName $mi $lastName"
```

This creates a very readable way to print multiple strings:

```
val name = println(s"Name: $firstName $mi $lastName")
```

As shown, all you have to do to use this approach is to precede the string with the letter `s`, and then put a `$` symbol before your variable names inside the string. This feature is known as *string interpolation*.

> You can also precede strings with the letter `f`, which lets you use *printf* style formatting inside strings. See my Scala string interpolation tutorial[1] for more information.

[1] http://kbhr.co/hs-string-interp

10.1 Multiline strings

A second great feature of Scala strings is that you can create multiline strings by including the string inside three parentheses:

```
val speech = """Four score and
               seven years ago
               our fathers ..."""
```

That's very helpful for when you need to work with multiline strings. One drawback of this basic approach is that lines after the first line are indented, as you can see in the REPL:

```
scala> val speech = """Four score and
     |                 seven years ago
     |                 our fathers ..."""
speech: String =
Four score and
                  seven years ago
                  our fathers ...
```

A simple way to fix this problem is to put a `|` symbol in front of all lines after the first line, and call the `stripMargin` method after the string:

```
val speech = """Four score and
               |seven years ago
               |our fathers ...""".stripMargin
```

The REPL shows that when you do this, all of the lines are left-justified:

```
scala> val speech = """Four score and
     |                 |seven years ago
     |                 |our fathers ...""".stripMargin
speech: String =
Four score and
seven years ago
our fathers ...
```

Because this is generally what you want, this is a common way to create multiline strings.

There are many more cool things you can do with strings. See my collection of over 100 Scala string examples[2] for more details and examples.

[2] http://kbhr.co/hs-strings

11
Command-Line I/O

To get ready to show `for` loops, `if` expressions, and other Scala constructs, let's take a look at how to handle command-line input and output with Scala.

11.1 Writing output

As I've already shown, you write output to standard out (STDOUT) using `println`:

```
println("Hello, world")
```

That function adds a newline character after your string, so if you don't want that, just use `print` instead:

```
print("Hello without newline")
```

When needed, you can also write output to standard error (STDERR) like this:

```
System.err.println("yikes, an error happened")
```

> Because `println` is so commonly used, there's no need to import it. The same is true of other commonly-used types like `String`, `Int`, `Float`, etc.

11.2 Reading input

There are several ways to read command-line input, but the easiest way is to use the `readLine` method in the *scala.io.StdIn* package.

To demonstrate how `readLine` works, let's create a little example. Put this source code in a file named *HelloInteractive.scala*:

```
import scala.io.StdIn.readLine

object HelloInteractive extends App {

    print("Enter your first name: ")
    val firstName = readLine()

    print("Enter your last name: ")
    val lastName = readLine()

    println(s"Your name is $firstName $lastName")

}
```

Then compile it with `scalac`:

```
$ scalac HelloInteractive.scala
```

Then run it with `scala`:

```
$ scala HelloInteractive
```

When you run the program and enter your first and last names at the prompts, the interaction looks like this:

```
$ scala HelloInteractive
Enter your first name: Alvin
Enter your last name: Alexander
Your name is Alvin Alexander
```

11.2.1 A note about imports

As you saw in this application, you bring classes and methods into scope in Scala just like you do with Java and other languages, with `import` statements:

```
import scala.io.StdIn.readLine
```

That import statement brings the `readLine` method into the current scope so you can use it in the application.

12

Control Structures

Scala has the basic control structures you'd expect to find in a programming language, including:

- `if/then/else`
- `for` loops
- `try/catch/finally`

It also has a few advanced constructs, including:

- `match` expressions
- `for` expressions

I'll demonstrate all of those in the following lessons.

13

The if/then/else Construct

A basic Scala `if` statement looks like this:

```
if (a == b) doSomething()
```

You can also write that statement like this:

```
if (a == b) {
    doSomething()
}
```

The `if/else` construct looks like this:

```
if (a == b) {
    doSomething()
} else {
    doSomethingElse()
}
```

The complete Scala if/else-if/else expression looks like this:

```
if (test1) {
    doX()
} else if (test2) {
    doY()
} else {
    doZ()
}
```

13.1 `if` expressions always return a result

A great thing about the Scala `if` construct is that it always returns a result. You can ignore the result as I did in the previous examples, but a more common approach —

especially in functional programming — is to assign the result to a variable:

```
val minValue = if (a < b) a else b
```

This is cool because it means that Scala doesn't require a special "ternary" operator.

13.2 Aside: Expression-oriented programming

As a brief note about programming in general, when every expression you write returns a value, that style is referred to as *expression-oriented programming*, or EOP. This is an example of an *expression*:

```
val minValue = if (a < b) a else b
```

Conversely, lines of code that don't return values are called *statements*, and statements are used for their *side-effects*. For example, these lines of code don't return values, so they're used for their side effects:

```
if (a == b) doSomething()
println("Hello")
```

The first example runs the `doSomething` method as a side effect when `a` is equal to `b`. The second example is used for the side effect of writing a string to STDOUT. As you learn more about Scala you'll find yourself writing more *expressions* and fewer *statements*.

14

for and while Loops

In its most simple use, a Scala *for-loop* can be used to iterate over the elements in a collection. For example, given a sequence of integers in a `Vector`:

```
val nums = Vector(1,2,3)
```

you can loop over them and print out their values like this:

```
for (n <- nums) println(n)
```

This is what that code looks like in the REPL:

```
scala> val nums = Vector(1,2,3)
nums: scala.collection.immutable.Vector[Int] = Vector(1, 2, 3)

scala> for (n <- nums) println(n)
1
2
3
```

That example stores a sequence of integers in a `Vector`, resulting in the data type `Vector[Int]`. Similarly, here's a `List` of strings, which has the data type `List[String]`:

```
val people = List(
    "Bill",
    "Candy",
    "Karen",
    "Leo",
    "Regina"
)
```

You print its values using a `for` loop just like the previous example:

```
for (p <- people) println(p)
```

> `Vector` and `List` are two types of sequential collections classes. In Scala these classes are generally preferred over `Array`. (More on this later.)

14.1 The `foreach` method

For the purpose of iterating over a collection of elements and printing its contents you can also use the `foreach` method that's available to Scala collections classes. For example, this is how you use `foreach` to print the previous list of strings:

```
people.foreach(println)
```

These days I generally use `for` loops, but `foreach` is also available on data types like `Vector`, `List`, `Array`, `ArrayBuffer`, `Map`, `Set`, and more.

14.2 Using `for` and `foreach` with Maps

You can also use `for` and `foreach` when working with a Scala `Map` (which is similar to a Java `HashMap`). For example, given this `Map` of movie names and ratings:

```
val ratings = Map(
    "Lady in the Water"  -> 3.0,
    "Snakes on a Plane"  -> 4.0,
    "You, Me and Dupree" -> 3.5
)
```

You can print the names and ratings using `for` like this:

```
for ((name,rating) <- ratings) println(s"Movie: $name, Rating: $rating")
```

Here's what that looks like in the REPL:

```
scala> for ((name,rating) <- ratings) println(s"Movie: $name, Rating: $rating")
Movie: Lady in the Water, Rating: 3.0
Movie: Snakes on a Plane, Rating: 4.0
```

```
Movie: You, Me and Dupree, Rating: 3.5
```

In this example, `name` corresponds to each *key* in the map, and `rating` is the name for each *value* in the map.

You can also print the ratings with `foreach` like this:

```
ratings.foreach {
    case(movie, rating) => println(s"key: $movie, value: $rating")
}
```

When I first started working with Scala I used `foreach` quite a bit, but after learning about functional programming I rarely use `foreach`, mainly because it's only used for *side effects*. Therefore, I won't discuss the `case` syntax used in this example. However, I will discuss `match` expressions and `case` statements later in this book.

14.3 while and do/while

Scala also has *while* and *do/while* loops, which are also used for side effects. Here's the while loop:

```
var i = 0
while (i < 3) {
    println(i)
    i += 1
}
```

This is the do/while loop syntax:

```
var i = 0
do {
    println(i)
    i += 1
} while (i < 3)
```

As shown, you use `+=` to increment an `Int` variable. Similarly, you use `-=` to decrement one.

15

for Expressions

If you recall what I wrote about Expression-Oriented Programming (EOP) and the difference between *expressions* and *statements*, you'll notice that in the previous lesson I used the `for` keyword and `foreach` method as tools for side effects: I used them to print the values in collections to STDOUT using `println`. Java has similar tools, and that's how I used them for many years without ever giving much thought to how they could be improved.

After I started working with Scala I learned that in functional programming languages you can use powerful for-expressions (also known as for-comprehensions) in addition to for-loops. In Scala, a for-expression is a different use of the `for` construct. While a *for-loop* is used for side effects (such as printing output), a *for-expression* is used to create a new collection from an existing collection. (In advanced Scala code it has even more uses.)

For example, given this list of integers:

```
val nums = Seq(1,2,3)
```

You can create a new list of integers where all of the values are doubled, like this:

```
val doubledNums = for (n <- nums) yield n * 2
```

That expression can be read as, "For every number `n` in the list of numbers `nums`, double each value, and then assign all of the new values to the variable `doubledNums`." This is what it looks like in the Scala REPL:

```
scala> val doubledNums = for (n <- nums) yield n * 2
doubledNums: Seq[Int] = List(2, 4, 6)
```

As the REPL output shows, the new list `doubledNums` contains these values:

```
List(2,4,6)
```

The result of the for-expression is that it creates a new variable named `doubledNums` whose values were created by doubling each value in the original list, `nums`.

15.1 Capitalizing a list of strings

You can use the same approach with a list of strings. For example, given this list of lowercase strings:

```
val names = List("adam", "david", "frank")
```

You can create a list of capitalized strings with this for-expression:

```
val capNames = for (name <- names) yield name.capitalize
```

The REPL shows how this works:

```
scala> val capNames = for (name <- names) yield name.capitalize
capNames: List[String] = List(Adam, David, Frank)
```

Success! Each name in the new variable `capNames` is capitalized.

15.2 The `yield` keyword

Notice that both of those for-expressions use the `yield` keyword:

```
val doubledNums = for (n <- nums) yield n * 2
                                  -----

val capNames = for (name <- names) yield name.capitalize
                                    -----
```

Using `yield` after `for` is the "secret sauce" that says, "I want to yield a new collection from the existing collection that I'm iterating over in the for-expression, using the algorithm shown."

It's important to note that the original collections `nums` and `names` have not been changed. The for-expressions shown create the new collections `doubledNums` and `capNames` from those original collections without modifying them.

15.3 Using a block of code after `yield`

The code after the `yield` expression can be as long as necessary to solve the current problem. For example, given a list of strings like this:

```
val names = List("_adam", "_david", "_frank")
```

Imagine that you want to create a new list that has the capitalized names of each person. To do that, you first need to remove the underscore character at the beginning of each name, and then capitalize each name. To remove the underscore from each name, you call the `tail` method on each `String`, which returns every character after the first character. After you do that, you call the `capitalize` method on each string. Here's a for-expression that implements this algorithm:

```
val capNames = for (name <- names) yield {
    val nameWithoutUnderscore = name.tail
    val capName = nameWithoutUnderscore.capitalize
    capName
}
```

If you put that code in the REPL, you'll see this result:

```
capNames: List[String] = List(Adam, David, Frank)
```

15.3.1 How `tail` works

The `tail` method works on sequential collections, and returns every element in the collection after the first element (which is known as the *head* element). Because a `String` is a sequence of characters (`Seq[Char]`), the `head` and `tail` methods work on strings like this:

```
scala> val result = "fred".head
result: Char = f

scala> val result = "fred".tail
result: String = red
```

15.3.2 A shorter version of the solution

I show the verbose form of the solution in that example so you can see how to use multiple lines of code after `yield`. However, for this particular example you can also write the code like this, which is more of the Scala style:

```
val capNames = for (name <- names) yield name.tail.capitalize
```

You can also put curly braces around the algorithm, if you prefer:

```
val capNames = for (name <- names) yield { name.tail.capitalize }
```

Lastly, you can also explicitly show the variable type, if you prefer:

```
val capNames: List[String] = for (name <- names) yield name.tail.capitalize
            --------------
```

15.4 See also

- My Scala for-loop examples and syntax[1]
- My How to create Scala for-expressions[2]
- List Comprehensions on Wikipedia[3]

[1] http://kbhr.co/hs-for-loop
[2] http://kbhr.co/hs-for-expr
[3] https://en.wikipedia.org/wiki/List_comprehension

16

match Expressions

Scala has a concept of a `match` expression. In the most simple case you can use a `match` expression like a Java `switch` statement:

```
// i is an integer
i match {
    case 1  => println("January")
    case 2  => println("February")
    case 3  => println("March")
    case 4  => println("April")
    case 5  => println("May")
    case 6  => println("June")
    case 7  => println("July")
    case 8  => println("August")
    case 9  => println("September")
    case 10 => println("October")
    case 11 => println("November")
    case 12 => println("December")
    // catch-all case for any other number
    case _  => println("Invalid month")
}
```

As shown, with a `match` expression you write a number of `case` statements that you use to match possible values. In this example I match the integer values `1` through `12`. Any other value falls down to the `_` case, which is the catch-all, default case.

`match` expressions are nice because they also return values, so rather than directly printing a string as in that example, you can assign the string result to a new value:

```
val monthName = i match {
    case 1  => "January"
    case 2  => "February"
    case 3  => "March"
```

```
    case 4  => "April"
    case 5  => "May"
    case 6  => "June"
    case 7  => "July"
    case 8  => "August"
    case 9  => "September"
    case 10 => "October"
    case 11 => "November"
    case 12 => "December"
    case _  => "Invalid month"
}
```

Using a `match` expression to yield a result like this is a common use.

16.1 Aside: A quick look at Scala methods

Scala also makes it easy to use a `match` expression as the body of a method. I haven't shown how to write Scala methods yet, so as a brief introduction, let me share a method named `convertBooleanToString` that takes a `Boolean` value named `bool` and returns a `String`:

```
def convertBooleanToString(bool: Boolean): String = {
    if (bool) "true" else "false"
}
```

Even though I haven't introduced the method syntax yet, I hope you can see how that code works. These REPL examples demonstrate it with `true` and `false` values:

```
scala> val answer = convertBooleanToString(true)
answer: String = true

scala> val answer = convertBooleanToString(false)
answer: String = false
```

16.2 Using a `match` expression as the body of a method

Now that you've seen an example of a Scala method, here's a second example that works just like the previous one, taking a `Boolean` value named `bool` as an input parameter

and returning a `String` message. The big difference is that this method uses a `match` expression for the body of the method:

```
def convertBooleanToString(bool: Boolean): String = bool match {
    case true => "true"
    case false => "false"
}
```

The body of that method is a match expression with two `case` statements, one that matches `true` and another that matches `false`. Because those are the only possible `Boolean` values, there's no need for a default `case` statement.

The REPL shows how you call that method and then print its result:

```
scala> val result = convertBooleanToString(true)
result: String = true

scala> println(result)
true
```

Using a `match` expression as the body of a method is a common technique.

16.3 Handling alternate cases

Scala `match` expressions are extremely powerful, so I'll demonstrate a few other things you can do with them.

`match` expressions let you handle multiple cases in a single `case` statement. To demonstrate this, imagine that you want to evaluate "boolean equality" like the Perl programming language handles it: a `0` or a blank string evaluates to false, and anything else evaluates to true. This is how you write a method using a `match` expression that evaluates to true and false in the manner described:

```
def isTrue(a: Any) = a match {
    case 0 | "" => false
    case _ => true
}
```

Because the input parameter `a` is defined to be the `Any` type — which is the root of all Scala classes, like `Object` in Java — this method works with any data type that's passed in:

```
scala> isTrue(0)
res0: Boolean = false

scala> isTrue("")
res1: Boolean = false

scala> isTrue(1.1F)
res2: Boolean = true

scala> isTrue(new java.io.File("/etc/passwd"))
res3: Boolean = true
```

The key part of this solution is that this single `case` statement lets both `0` and the empty string evaluate to `false`:

```
case 0 | "" => false
```

Before I move on, here's another example that shows many matches in each `case` statement:

```
val evenOrOdd = i match {
    case 1 | 3 | 5 | 7 | 9 => println("odd")
    case 2 | 4 | 6 | 8 | 10 => println("even")
    case _ => println("some other number")
}
```

Here's another example that shows how to handle multiple strings in multiple `case` statements:

```
cmd match {
    case "start" | "go" => println("starting")
    case "stop" | "quit" | "exit" => println("stopping")
    case _ => println("doing nothing")
}
```

16.4 Using `if` expressions in `case` statements

Another great thing about `match` expressions is that you can use `if` expressions in `case` statements for powerful pattern matching. In this example the second and third `case` statements both use `if` expressions to match ranges of numbers:

```
count match {
    case 1 => println("one, a lonely number")
    case x if x == 2 || x == 3 => println("two's company, three's a crowd")
    case x if x > 3 => println("4+, that's a party")
    case _ => println("i'm guessing your number is zero or less")
}
```

Scala doesn't require you to use parentheses in the `if` expressions, but you can use them if you think that makes them more readable:

```
count match {
    case 1 => println("one, a lonely number")
    case x if (x == 2 || x == 3) => println("two's company, three's a crowd")
    case x if (x > 3) => println("4+, that's a party")
    case _ => println("i'm guessing your number is zero or less")
}
```

You can also write the code on the right side of the `=>` on multiple lines if you think that's easier to read. Here's one example:

```
count match {
    case 1 =>
        println("one, a lonely number")
    case x if x == 2 || x == 3 =>
        println("two's company, three's a crowd")
    case x if x > 3 =>
        println("4+, that's a party")
    case _ =>
        println("i'm guessing your number is zero or less")
}
```

Here's a variation of that example that uses parentheses around the body of each case:

```
count match {
    case 1 => {
        println("one, a lonely number")
    }
    case x if x == 2 || x == 3 => {
        println("two's company, three's a crowd")
    }
    case x if x > 3 => {
        println("4+, that's a party")
    }
    case _ => {
        println("i'm guessing your number is zero or less")
    }
}
```

Here are a few other examples of how you can use `if` expressions in `case` statements. First, another example of how to match ranges of numbers:

```
i match {
    case a if 0 to 9 contains a => println("0-9 range: " + a)
    case b if 10 to 19 contains b => println("10-19 range: " + a)
    case c if 20 to 29 contains c => println("20-29 range: " + a)
    case _ => println("Hmmm...")
}
```

Lastly, this example shows how to reference class fields in `if` expressions:

```
stock match {
    case x if (x.symbol == "XYZ" && x.price < 20) => buy(x)
    case x if (x.symbol == "XYZ" && x.price > 50) => sell(x)
    case x => doNothing(x)
}
```

16.5 Even more …

`match` expressions are very powerful, and there are even more things you can do with them. Please see the match expressions on this page[1] or the Scala Cookbook[2] for more examples.

[1] http://kbhr.co/hs-match
[2] http://kbhr.co/hs-cook

17

try/catch/finally Expressions

Like Java, Scala has a try/catch/finally construct to let you catch and manage exceptions. The main difference is that for consistency, Scala uses the same syntax that `match` expressions use: `case` statements to match the different possible exceptions that can occur.

17.1 A try/catch example

Here's an example of Scala's try/catch syntax. In this example, `openAndReadAFile` is a method that does what its name implies: it opens a file and reads the text in it, assigning the result to the variable named `text`:

```
var text = ""
try {
    text = openAndReadAFile(filename)
} catch {
    case e: FileNotFoundException => println("Couldn't find that file.")
    case e: IOException => println("D'oh, an IOException!")
}
```

Scala uses the *java.io.** classes to work with files, so attempting to open and read a file can result in both a `FileNotFoundException` and an `IOException`. Those two exceptions are caught in the `catch` block of this example.

17.2 try, catch, and finally

The Scala try/catch syntax also lets you use a `finally` clause, which is typically used when you need to close a resource. Here's an example of what that looks like:

```
try {
    // your scala code here
}
catch {
    case foo: FooException => handleFooException(foo)
    case bar: BarException => handleBarException(bar)
    case _: Throwable => println("Got some other kind of Throwable")
} finally {
    // your scala code here, such as closing a database connection
    // or file handle
}
```

17.3 More later

I'll cover more details about Scala's try/catch/finally syntax in later lessons, such as in the "Error Handling" lessons, but these examples demonstrate the syntax. It's great that it's consistent with the `match` expression syntax because it's easier to remember, and therefore less of a burden on my brain.

18
Classes

In support of object-oriented programming (OOP), Scala provides a *class* construct. The syntax is more concise than languages like Java and C#, but it's also still easy to use and read.

18.1 Basic class constructor

Here's a Scala class whose constructor defines two parameters, `firstName` and `lastName`:

```
class Person(var firstName: String, var lastName: String)
```

With that definition you can create new `Person` instances like this:

```
val p = new Person("Bill", "Panner")
```

Defining parameters in a class constructor automatically creates fields in the class, and in this example you can access the `firstName` and `lastName` fields like this:

```
scala> println(p.firstName + " " + p.lastName)
Bill Panner
```

In this example, because both fields are defined as `var` fields, they're also mutable, meaning they can be changed. This is how you change them:

```
scala> p.firstName = "Forest"
p.firstName: String = Forest

scala> p.lastName = "Bernheim"
p.lastName: String = Bernheim
```

If you're coming to Scala from Java, this Scala code:

```
class Person(var firstName: String, var lastName: String)
```

is pretty much the equivalent of this Java code:

```
public class Person {

    private String firstName;
    private String lastName;

    public Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public String getFirstName() {
        return this.firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return this.lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

}
```

18.2 `val` makes fields read-only

In that first example I defined both fields as `var` fields:

```
class Person(var firstName: String, var lastName: String)
             ---                    ---
```

That makes those fields mutable. You can also define them as `val` fields, which makes them immutable:

```
class Person(val firstName: String, val lastName: String)
             ---                    ---
```

If you now try to change the first or last name of a `Person` instance, you'll see an error:

```
scala> p.firstName = "Fred"
<console>:12: error: reassignment to val
       p.firstName = "Fred"
                   ^

scala> p.lastName = "Jones"
<console>:12: error: reassignment to val
       p.lastName = "Jones"
                  ^
```

> Pro tip: If you use Scala to write OOP code, create your fields as `var` fields so you can easily mutate them. When you write FP code with Scala, you'll generally use *case classes* instead of classes like this. (More on this later.)

18.3 Class constructors

In Scala, the primary constructor of a class is a combination of:

- The constructor parameters
- Methods that are called in the body of the class
- Statements and expressions that are executed in the body of the class

Fields declared in the body of a Scala class are handled in a manner similar to Java; they're assigned when the class is first instantiated.

This `Person` class demonstrates several of the things you can do inside the body of a class.

```
class Person(var firstName: String, var lastName: String) {

    println("the constructor begins")

    // fields have 'public' access by default
    var age = 0

    // a private class field
    private val HOME = System.getProperty("user.home")

    // some methods
    override def toString(): String =
        s"$firstName $lastName is $age years old"

    def printHome(): Unit = println(s"HOME = $HOME")

    def printFullName(): Unit = println(this)

    printHome()
    printFullName()
    println("you've reached the end of the constructor")

}
```

Putting this code in the REPL demonstrates how it works:

```
scala> val p = new Person("Kim", "Carnes")
the constructor begins
HOME = /Users/al
Kim Carnes is 0 years old
you've reached the end of the constructor
p: Person = Kim Carnes is 0 years old
// that last line is output by the REPL, not my code

scala> p.age
res0: Int = 0

scala> p.age = 36
p.age: Int = 36
```

```
scala> p
res1: Person = Kim Carnes is 36 years old

scala> p.printHome
HOME = /Users/al

scala> p.printFullName
Kim Carnes is 36 years old
```

Speaking from my own experience, this constructor approach felt a little unusual at first, but once I understood how it works I found it to be logical and convenient.

18.4 A note about the special procedure syntax

In that example I declared these two methods to return the `Unit` type:

```
def printHome(): Unit = println(s"HOME = $HOME")

def printFullName(): Unit = println(this)
```

The `Unit` return type means that these methods don't return anything; in this case they just print some output. Methods that don't return anything are known as *procedures*. Up through at least Scala 2.12 you can also use this special procedure syntax to declare procedures:

```
def printHome { println(s"HOME = $HOME") }

def printFullName { println(this) }
```

Because these methods don't have any input parameters and also have no return type, that's a perfectly legal way to define these methods.

> However, be aware that this syntax may go away in future Scala releases. (There's a concern that this is a special syntax just to save a few characters of typing.)

18.5 Other Scala class examples

Before we move on, here are a few other examples of Scala classes:

```
class Pizza (
    var crustSize: Int,
    var crustType: String,
    var toppings: Seq[Topping]
)

// a stock, like AAPL or GOOG
class StockPriceInstance(
    var symbol: String,
    var price: BigDecimal,
    var datetime: Date
)

// a network socket
class Socket(val timeout: Int, val linger: Int) {
    override def toString = s"timeout: $timeout, linger: $linger"
}

class Address (
    var street1: String,
    var street2: String,
    var city: String,
    var state: String
)
```

19
Auxiliary Class Constructors

Auxiliary class constructors are defined by creating methods in the class that are named `this`. There are only a few rules to know:

- Each auxiliary constructor must have a different signature (different parameter lists)
- Each auxiliary constructor must call one of the previously defined constructors

Here's an example of a `Pizza` class that defines multiple constructors:

```
val DEFAULT_CRUST_SIZE = 12
val DEFAULT_CRUST_TYPE = "THIN"

// the primary constructor
class Pizza (var crustSize: Int, var crustType: String) {

    // one-arg auxiliary constructor
    def this(crustSize: Int) {
        this(crustSize, DEFAULT_CRUST_TYPE)
    }

    // one-arg auxiliary constructor
    def this(crustType: String) {
        this(DEFAULT_CRUST_SIZE, crustType)
    }

    // zero-arg auxiliary constructor
    def this() {
        this(DEFAULT_CRUST_SIZE, DEFAULT_CRUST_TYPE)
    }
```

```
    override def toString = s"A $crustSize inch pizza with a $crustType crust"

}
```

With all of those constructors defined, you can create pizza instances in several different ways:

```
val p1 = new Pizza(DEFAULT_CRUST_SIZE, DEFAULT_CRUST_TYPE)
val p2 = new Pizza(DEFAULT_CRUST_SIZE)
val p3 = new Pizza(DEFAULT_CRUST_TYPE)
val p4 = new Pizza
```

I encourage you to paste that class and those examples into the Scala REPL to see how they work.

> Note: The `DEFAULT_CRUST_SIZE` and `DEFAULT_CRUST_TYPE` variables aren't a great example of how to handle this situation, but because I haven't shown how to handle enumerations yet, I use this approach to keep things simple.

20

Supplying Default Values for Constructor Parameters

A convenient Scala feature is that you can supply default values for constructor parameters. In the previous lessons I showed that you can define a `Socket` class like this:

```
class Socket(var timeout: Int, var linger: Int) {
    override def toString = s"timeout: $timeout, linger: $linger"
}
```

That's nice, but you can make this class even better by supplying default values for the `timeout` and `linger` parameters:

```
class Socket(var timeout: Int = 2000, var linger: Int = 3000) {
    override def toString = s"timeout: $timeout, linger: $linger"
}
```

By supplying default values for the parameters, you can now create a new `Socket` in a variety of different ways:

```
new Socket
new Socket()
new Socket(1000)
new Socket(4000, 6000)
```

This is what those examples look like in the REPL:

```
scala> new Socket
res0: Socket = timeout: 2000, linger: 3000

scala> new Socket()
res1: Socket = timeout: 2000, linger: 3000
```

```
scala> new Socket(1000)
res2: Socket = timeout: 1000, linger: 3000

scala> new Socket(4000, 6000)
res3: Socket = timeout: 4000, linger: 6000
```

20.1 Bonus: Named parameters

Another nice thing about Scala is that you can use a different feature called *named parameters* when creating an instance of a class. For example, given this class:

```
class Socket(var timeout: Int, var linger: Int) {
    override def toString = s"timeout: $timeout, linger: $linger"
}
```

you can create a new `Socket` using named parameters like this:

```
val s = new Socket(timeout=2000, linger=3000)
```

I personally don't use this feature very often, but it comes in handy every once in a while, especially when every class constructor parameters has the same type, such as `Int` in this example. For instance, some people find that this code:

```
val s = new Socket(timeout=2000, linger=3000)
```

is more readable than this code:

```
val s = new Socket(2000, 3000)
```

You can also use named parameters when calling methods.

21

A First Look at Methods

In Scala, *methods* are defined inside classes (just like Java), but for testing purposes you can also create them in the REPL. This lesson shows a few examples of methods so you can see what the syntax looks like.

21.1 Defining a method that takes one input parameter

This is how you define a method named `double` that takes one integer input parameter named `a` and returns the doubled value of that integer:

```
def double(a: Int) = a * 2
```

In that example the method name and signature are shown on the left side of the = sign:

```
def double(a: Int) = a * 2
    --------------
```

`def` is the keyword you use to define a method, the method name is `double`, and the input parameter `a` has the type `Int`, which is Scala's integer data type.

The body of the method is shown on the right side, and in this example it simply doubles the value of the input parameter `a`:

```
def double(a: Int) = a * 2
                     -----
```

After you paste that method into the REPL, you call it (invoke it) by giving it an `Int` value:

```
scala> double(2)
res0: Int = 4
```

```
scala> double(10)
res1: Int = 20
```

21.2 Showing the method's return type

In the previous example I don't show the method's return type, but you can show it, and indeed, I normally do:

```
def double(a: Int): Int = a * 2
                  -----
```

Writing a method like this *explicitly* declares the method's return type. When I first started working with Scala I tended to leave the return type off of my method declarations, but after a while I found that it was easier to maintain my code when I declared the return type. That way I could just scan the function signature to easily see its input and output types.

> That being said, that's just my personal preference; use whatever you like.

If you paste that method into the REPL, you'll see that it works just like the previous method.

21.3 Methods with multiple input parameters

To show something a little more complex, here's a method that takes two input parameters:

```
def add(a: Int, b: Int) = a + b
```

Here's the same method, with the method's return type explicitly shown:

```
def add(a: Int, b: Int): Int = a + b
```

Here's a method that takes three input parameters:

```
def add(a: Int, b: Int, c: Int): Int = a + b + c
```

21.4 Multiline methods

When a method is only one line long I use the format I just showed, but when the method body gets longer, you must put the lines inside curly braces:

```
def addThenDouble(a: Int, b: Int): Int = {
    val sum = a + b
    val doubled = sum * 2
    doubled
}
```

If you paste that code into the REPL, you'll see that it works just like the previous examples:

```
scala> addThenDouble(1, 1)
res0: Int = 4
```

21.5 `return` is optional

You can use the `return` keyword to return a value from your method:

```
def addThenDouble(a: Int, b: Int): Int = {
    val sum = a + b
    val doubled = sum * 2
    return doubled   //<-- return this result
}
```

However, it isn't required, and in fact, Scala programmers rarely ever use it:

```
def addThenDouble(a: Int, b: Int): Int = {
    val sum = a + b
    val doubled = sum * 2
    doubled  //<-- `return` isn't needed
}
```

In fact, that method can be reduced to this:

```
def addThenDouble(a: Int, b: Int): Int = {
    val sum = a + b
    sum * 2
}
```

or this:

```
def addThenDouble(a: Int, b: Int): Int = {
    (a + b) * 2
}
```

or this:

```
def addThenDouble(a: Int, b: Int): Int = (a + b) * 2
```

21.5.1 Why we don't use `return`

We don't use `return` for a couple of reasons. First, any code inside parentheses is really just a block of code that evaluates to a result. When you think about your code this way, you're not really "returning" anything; the block of code just evaluates to a result. For instance, if you paste this code into the REPL, you'll begin to see that it doesn't feel right to "return" a value from a block of code:

```
val c = {
    val a = 1
    val b = 2
    a + b
}
```

The second reason we don't use `return` is that when you write *pure functions*, the general feeling is that you're writing algebraic equations. If you remember your algebra, you know that you don't use `return` with mathematical expressions:

```
x = a + b
y = x * 2
```

Similarly, as your code becomes more functional and you write it more like math expressions, you'll find that you won't use `return` any more.

21.6 See also

If you're interested in pure functions and functional programming, I write *much* more about them in my book, Functional Programming, Simplified[1].

[1] http://kbhr.co/hs-fps

22

Enumerations (and a Complete Pizza Class)

In this lesson I'll demonstrate how to create enumerations in Scala. By doing this now, I can show you what an example `Pizza` class looks like when written in an object-oriented manner.

Enumerations are a useful tool for creating small groups of constants, things like the days of the week, months in a year, suits in a deck of cards, etc., situations where you have a group of related, constant values.

Because I'm jumping ahead a little bit here I'm not going to explain this syntax too much, but this is how you create an enumeration for the days of a week:

```
sealed trait DayOfWeek
case object Sunday extends DayOfWeek
case object Monday extends DayOfWeek
case object Tuesday extends DayOfWeek
case object Wednesday extends DayOfWeek
case object Thursday extends DayOfWeek
case object Friday extends DayOfWeek
case object Saturday extends DayOfWeek
```

Similarly, this is how you create an enumeration for the suits in a deck of cards:

```
sealed trait Suit
case object Clubs extends Suit
case object Spades extends Suit
case object Diamonds extends Suit
case object Hearts extends Suit
```

I'll discuss traits and case objects later in this book, but if you'll trust me that this is how you create enumerations, I can now create a little OOP version of a `Pizza` class.

22.1 Pizza-related enumerations

Given that brief introduction to enumerations, here are some useful pizza-related enumerations:

```
sealed trait Topping
case object Cheese extends Topping
case object Pepperoni extends Topping
case object Sausage extends Topping
case object Mushrooms extends Topping
case object Onions extends Topping

sealed trait CrustSize
case object SmallCrustSize extends CrustSize
case object MediumCrustSize extends CrustSize
case object LargeCrustSize extends CrustSize

sealed trait CrustType
case object RegularCrustType extends CrustType
case object ThinCrustType extends CrustType
case object ThickCrustType extends CrustType
```

Those enumerations provide a nice way to work with pizza toppings, crust sizes, and crust types.

22.2 A sample Pizza class

Now that I have those enumerations, I can define a `Pizza` class like this:

```
class Pizza (
    var crustSize: CrustSize = MediumCrustSize,
    var crustType: CrustType = RegularCrustType
) {

    // ArrayBuffer is a mutable sequence (list)
    val toppings = scala.collection.mutable.ArrayBuffer[Topping]()

    def addTopping(t: Topping): Unit = { toppings += t }
    def removeTopping(t: Topping): Unit = { toppings -= t }
```

```
    def removeAllToppings(): Unit = { toppings.clear() }

}
```

If you save all of that code — including the enumerations — in a file named *Pizza.scala*, you can compile it with the usual command:

```
$ scalac Pizza.scala
```

> That code will create a lot of individual files, so I recommend putting it in a separate directory.

There's nothing to run yet because this class doesn't have a `main` method, but …

22.3 A complete Pizza class with a main method

If you're ready to have some fun, replace all of the code in *Pizza.scala* with the following code, which includes a new `toString` method in the `Pizza` class and a new driver `App` named `PizzaTest`:

```
import scala.collection.mutable.ArrayBuffer

sealed trait Topping
case object Cheese extends Topping
case object Pepperoni extends Topping
case object Sausage extends Topping
case object Mushrooms extends Topping
case object Onions extends Topping

sealed trait CrustSize
case object SmallCrustSize extends CrustSize
case object MediumCrustSize extends CrustSize
case object LargeCrustSize extends CrustSize

sealed trait CrustType
case object RegularCrustType extends CrustType
case object ThinCrustType extends CrustType
case object ThickCrustType extends CrustType
```

```
class Pizza (
    var crustSize: CrustSize = MediumCrustSize,
    var crustType: CrustType = RegularCrustType
) {

    // ArrayBuffer is a mutable sequence (list)
    val toppings = ArrayBuffer[Topping]()

    def addTopping(t: Topping): Unit = { toppings += t }
    def removeTopping(t: Topping): Unit = { toppings -= t }
    def removeAllToppings(): Unit = { toppings.clear() }

    override def toString(): String = {
        s"""
        |Crust Size: $crustSize
        |Crust Type: $crustType
        |Toppings:   $toppings
        """.stripMargin
    }
}

// a little "driver" app
object PizzaTest extends App {
   val p = new Pizza
   p.addTopping(Cheese)
   p.addTopping(Pepperoni)
   println(p)
}
```

Notice how you can put all of the enumerations, a `Pizza` class, and a `PizzaTest` object in the same file. That's a very convenient Scala feature.

Next, compile that code with the usual command:

```
$ scalac Pizza.scala
```

Then run the `PizzaTest` object with this command:

```
$ scala PizzaTest
```

The output should look like this:

```
$ scala PizzaTest

Crust Size: MediumCrustSize
Crust Type: RegularCrustType
Toppings:   ArrayBuffer(Cheese, Pepperoni)
```

I put several different concepts together to create that code — including two things I haven't discussed yet in the `import` statement and the `ArrayBuffer` — but if you have experience with Java and other languages, I hope it's not too much to throw at you at one time.

At this point I encourage you to work with that code as desired. Make changes to the code, and try using the `removeTopping` and `removeAllToppings` methods to make sure they work the way you expect them to work.

23

Traits and Abstract Classes

Scala traits are a great feature of the language. As I'll show in the following lessons, you can use them just like a Java interface, and you can also use them to "mix in" new behaviors. Scala classes can also extend multiple traits.

Scala also has the concept of an abstract class, and I'll show when you should use an abstract class instead of a trait.

24

Using Traits as Interfaces

One way to use a Scala `trait` is like a Java `interface`, where you define the desired interface for some piece of functionality, but you don't implement any behavior.

24.1 A simple example

As an example to get us started, imagine that you want to write some code to model animals like dogs, cats, or any animal that has a tail. In Scala you write a trait to start that modeling process like this:

```
trait TailWagger {
    def startTail(): Unit
    def stopTail(): Unit
}
```

That code declares a trait named `TailWagger` that states that any class that extends `TailWagger` should implement `startTail` and `stopTail` methods. Both of those methods take no input parameters and have no return value. This code is equivalent to this Java interface:

```
public interface TailWagger {
    public void startTail();
    public void stopTail();
}
```

24.2 Extending a trait

Given this trait:

```
trait TailWagger {
    def startTail(): Unit
    def stopTail(): Unit
}
```

you can write a class that extends the trait and implements those methods like this:

```
class Dog extends TailWagger {
    // the implemented methods
    def startTail(): Unit = { println("tail is wagging") }
    def stopTail(): Unit = { println("tail is stopped") }
}
```

Notice that you use the `extends` keyword to create a class that extends a single trait.

If you paste the `TailWagger` trait and `Dog` class into the Scala REPL, you can test the code like this:

```
scala> val d = new Dog
d: Dog = Dog@234e9716

scala> d.startTail
tail is wagging

scala> d.stopTail
tail is stopped
```

That demonstrates how to implement a single Scala trait with a class that extends the trait.

24.3 Extending multiple traits

Scala lets you create very modular code with traits. For example, you can break down the attributes of animals into small, logical, modular units:

```
trait Speaker {
    def speak(): String
}
```

```
trait TailWagger {
    def startTail(): Unit
    def stopTail(): Unit
}

trait Runner {
    def startRunning(): Unit
    def stopRunning(): Unit
}
```

Once you have those small pieces, you can create a `Dog` class by extending all of them, and implementing the necessary methods:

```
class Dog extends Speaker with TailWagger with Runner {

    // Speaker
    def speak(): String = "Woof!"

    // TailWagger
    def startTail(): Unit = { println("tail is wagging") }
    def stopTail(): Unit = { println("tail is stopped") }

    // Runner
    def startRunning(): Unit = { println("I'm running") }
    def stopRunning(): Unit = { println("Stopped running") }

}
```

Key points of this code:

- Use `extends` to extend the first trait
- Use `with` to extend subsequent traits

So far you've seen that Scala traits work just like Java interfaces. But there's more …

25

Using Traits Like Abstract Classes

Traits have much more functionality than what I just showed. You can also add real, working methods to them and use them like abstract classes, or more accurately, as *mixins*.

25.1 A first example

To demonstrate this, here's a Scala trait that has a *concrete* method named `speak`, and an *abstract* method named `comeToMaster`:

```
trait Pet {
    def speak { println("Yo") }   // concrete implementation
    def comeToMaster(): Unit      // abstract
}
```

When a class extends a trait each defined method must be implemented, so here's a `Dog` class that extends `Pet` and defines `comeToMaster`:

```
class Dog(name: String) extends Pet {
    def comeToMaster(): Unit = println("Woo-hoo, I'm coming!")
}
```

Unless you want to override `speak`, there's no need to redefine it, so this is a perfectly complete Scala class. Now you can create a new `Dog` like this:

```
val d = new Dog("Zeus")
```

Then you can call `speak` and `comeToMaster`. This is what it looks like in the REPL:

```
scala> val d = new Dog("Zeus")
d: Dog = Dog@4136cb25
```

```
scala> d.speak
Yo

scala> d.comeToMaster
Woo-hoo, I'm coming!
```

25.2 Overriding an implemented method

A class can also override a method that's defined in a trait. Here's an example:

```
class Cat extends Pet {
    // override 'speak'
    override def speak(): Unit = println("meow")
    def comeToMaster(): Unit = println("That's not gonna happen.")
}
```

The REPL shows how this works:

```
scala> val c = new Cat
c: Cat = Cat@1953f27f

scala> c.speak
meow

scala> c.comeToMaster
That's not gonna happen.
```

25.3 Mixing in multiple traits that have behaviors

A great thing about Scala traits is that you can mix multiple traits that have behaviors into classes. For example, here's a combination of traits, one of which defines an abstract method, and the others that define concrete method implementations:

```
trait Speaker {
    def speak(): String   //abstract
}
```

```
trait TailWagger {
    def startTail(): Unit = println("tail is wagging")
    def stopTail(): Unit = println("tail is stopped")
}

trait Runner {
    def startRunning(): Unit = println("I'm running")
    def stopRunning(): Unit = println("Stopped running")
}
```

Now you can create a `Dog` class that extends all of those traits while providing behavior for the `speak` method:

```
class Dog(name: String) extends Speaker with TailWagger with Runner {
    def speak(): String = "Woof!"
}
```

And here's a `Cat` class:

```
class Cat extends Speaker with TailWagger with Runner {
    def speak(): String = "Meow"
    override def startRunning(): Unit = println("Yeah ... I don't run")
    override def stopRunning(): Unit = println("No need to stop")
}
```

The REPL shows that this all works like you'd expect it to work. First, a `Dog`:

```
scala> d.speak
res0: String = Woof!

scala> d.startRunning
I'm running

scala> d.startTail
tail is wagging
```

Then a `Cat`:

```
scala> val c = new Cat
c: Cat = Cat@1b252afa
```

```
scala> c.speak
res1: String = Meow

scala> c.startRunning
Yeah ... I don't run

scala> c.startTail
tail is wagging
```

25.4 Mixing traits in on the fly

As a last note, another interesting thing you can do with traits that have concrete methods is that you can mix them in on the fly. For example, given these traits:

```
trait TailWagger {
    def startTail(): Unit = println("tail is wagging")
    def stopTail(): Unit = println("tail is stopped")
}

trait Runner {
    def startRunning(): Unit = println("I'm running")
    def stopRunning(): Unit = println("Stopped running")
}
```

and this `Dog` class:

```
class Dog(name: String)
```

you can create a `Dog` instance that mixes in those traits when you create a `Dog` instance:

```
val d = new Dog("Fido") with TailWagger with Runner
                        ---------------------------
```

Once again the REPL shows that this works:

```
scala> val d = new Dog("Fido") with TailWagger with Runner
d: Dog with TailWagger with Runner = $anon$1@50c8d274
```

```
scala> d.startTail
tail is wagging

scala> d.startRunning
I'm running
```

This example works because all of the methods in the `TailWagger` and `Runner` traits are defined (they're not abstract).

25.5 See also

There are many more things you can do with Scala traits. For more details and examples, please see the Scala Cookbook[1].

[1] http://kbhr.co/hs-cook

26
Abstract Classes

Scala also has a concept of an abstract class that's similar to Java's abstract class. But because traits are so powerful, you rarely need to use an abstract class. In fact, you only need to use an abstract class when:

- You want to create a base class that requires constructor arguments
- Your Scala code will be called from Java code

26.1 Scala traits don't allow constructor parameters

Regarding the first reason, Scala traits don't allow constructor parameters:

```
// this won't compile
trait Animal(name: String)
```

Therefore, you need to use an abstract class whenever a base behavior must have constructor parameters:

```
abstract class Animal(name: String)
```

However, be aware that a class can only extend one abstract class.

26.2 When Scala code will be called from Java code

Regarding the second point, because Java doesn't know anything about Scala traits, if you want to call your Scala code from Java code you'll need to use an abstract class rather than a trait.

I won't show how to do this in this book, but if you're interested in an example, please see the Scala Cookbook[1].

26.3 Abstract class syntax

The abstract class syntax is similar to the trait syntax. For example, here's an abstract class named `Pet` that's similar to the `Pet` trait I defined in the previous lesson:

```
abstract class Pet (name: String) {
    def speak(): Unit = println("Yo")    // concrete implementation
    def comeToMaster(): Unit             // abstract method
}
```

Given that abstract `Pet` class, you can define a `Dog` class like this:

```
class Dog(name: String) extends Pet(name) {
    override def speak() = println("Woof")
    def comeToMaster() = println("Here I come!")
}
```

The REPL shows that this all works as advertised:

```
scala> val d = new Dog("Rover")
d: Dog = Dog@51f1fe1c

scala> d.speak
Woof

scala> d.comeToMaster
Here I come!
```

26.3.1 Notice how name was passed along

All of that code is similar to Java, so I won't explain it in detail. One thing to notice is how the `name` constructor parameter is passed from the `Dog` class constructor to the

[1] http://kbhr.co/hs-cook

`Pet` constructor:

```
class Dog(name: String) extends Pet(name) {
          ----                      ----
```

Remember that `Pet` is declared to take `name` as a constructor parameter:

```
abstract class Pet (name: String) { ...
                    ----
```

Therefore, this example shows how to pass the constructor parameter from the `Dog` class to the abstract `Pet` class. You can verify that this works with this code:

```
abstract class Pet (name: String) {
    def speak(): Unit = println(s"My name is $name")
}

class Dog(name: String) extends Pet(name)

val d = new Dog("Fido")
d.speak
```

I encourage you to copy and paste that code into the REPL to be sure it works as you expect.

27

Collections Classes

If you're coming to Scala from Java, the best thing you can do is forget about the Java collections classes and use the Scala collections classes. Speaking from my own experience, when I first started working with Scala I tried to use Java collections classes in my Scala code, and in retrospect, that really slowed down my learning process. I would have been much better off using the Scala collections classes and their methods because they would have taught me the "Scala way" much more quickly.

27.1 The main Scala collections classes

The main Scala collections classes you'll use on a regular basis are:

Class	Description
`ArrayBuffer`	an indexed, mutable sequence
`List`	a linear (linked list), immutable sequence
`Vector`	an indexed, immutable sequence
`Map`	the base `Map` (key/value pairs) class
`Set`	the base `Set` class

`Map` and `Set` come in both mutable and immutable versions.

I'll demonstrate the basics of these classes in the following lessons.

> In the following lessons on the collections classes, whenever I use the word *immutable* it's safe to assume that the class is intended for use in a *functional programming* (FP) style.

28

ArrayBuffer Class

If you're an OOP developer coming to Scala from Java, the `ArrayBuffer` class will probably be most comfortable for you, so I'll demonstrate it first. Like Java's `ArrayList` it's a *mutable* sequence, so you can use its methods to modify its contents.

To use an `ArrayBuffer` you must first import it:

```
import scala.collection.mutable.ArrayBuffer
```

After it's imported into the local scope, you create an empty `ArrayBuffer` like this:

```
val ints = ArrayBuffer[Int]()
val names = ArrayBuffer[String]()
```

Once you have an `ArrayBuffer` you add elements to it in a variety of ways. The `+=` method is a common approach:

```
val ints = ArrayBuffer[Int]()
ints += 1
ints += 2
```

The REPL shows how `+=` works:

```
scala> ints += 1
res0: ints.type = ArrayBuffer(1)

scala> ints += 2
res1: ints.type = ArrayBuffer(1, 2)
```

That's just one way create an `ArrayBuffer` and add elements to it. You can also create an `ArrayBuffer` with initial elements like this:

```
val nums = ArrayBuffer(1, 2, 3)
```

Here are a few ways you can add more elements to this `ArrayBuffer`:

```
// add one element
nums += 4

// add two or more elements
nums += (5, 6)

// add elements from another collection
nums ++= List(7, 8)
```

You remove elements from an `ArrayBuffer` with the `-=` and `--=` methods:

```
// remove one element
nums -= 9

// remove two or more elements
nums -= (7, 8)

nums --= Array(5, 6)
```

Here's what all of those examples look like in the REPL:

```
scala> nums += 4
res2: nums.type = ArrayBuffer(1, 2, 3, 4)

scala> nums += (5, 6)
res3: nums.type = ArrayBuffer(1, 2, 3, 4, 5, 6)

scala> nums ++= List(7, 8)
res4: nums.type = ArrayBuffer(1, 2, 3, 4, 5, 6, 7, 8)

scala> nums -= 9
res5: nums.type = ArrayBuffer(1, 2, 3, 4, 5, 6, 7, 8)

scala> nums -= (7, 8)
res6: nums.type = ArrayBuffer(1, 2, 3, 4, 5, 6)
```

```
scala> nums --= Array(5, 6)
res7: nums.type = ArrayBuffer(1, 2, 3, 4)
```

28.1 More ways to work with `ArrayBuffer`

There are many more ways to work with an `ArrayBuffer`. Here are some of the most common methods:

```
val a = ArrayBuffer(1, 2, 3)          // ArrayBuffer(1, 2, 3)
a.append(4)                           // ArrayBuffer(1, 2, 3, 4)
a.append(5, 6)                        // ArrayBuffer(1, 2, 3, 4, 5, 6)
a.appendAll(Seq(7,8))                 // ArrayBuffer(1, 2, 3, 4, 5, 6, 7, 8)
a.clear                               // ArrayBuffer()
val a = ArrayBuffer(9, 10)            // ArrayBuffer(9, 10)
a.insert(0, 8)                        // ArrayBuffer(8, 9, 10)
a.insert(0, 6, 7)                     // ArrayBuffer(6, 7, 8, 9, 10)
a.insertAll(0, Vector(4, 5))          // ArrayBuffer(4, 5, 6, 7, 8, 9, 10)
a.prepend(3)                         // ArrayBuffer(3, 4, 5, 6, 7, 8, 9, 10)
a.prepend(1, 2)                 // ArrayBuffer(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
a.prependAll(Array(0))         // ArrayBuffer(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
val a = ArrayBuffer.range('a', 'h')  // ArrayBuffer(a, b, c, d, e, f, g)
a.remove(0)                           // ArrayBuffer(b, c, d, e, f, g)
a.remove(2, 3)                        // ArrayBuffer(b, c, g)
val a = ArrayBuffer.range('a', 'h')  // ArrayBuffer(a, b, c, d, e, f, g)
a.trimStart(2)                        // ArrayBuffer(c, d, e, f, g)
a.trimEnd(2)                          // ArrayBuffer(c, d, e)
```

Please see the Scala Cookbook[1] and my big page of ArrayBuffer examples[2] for more details on the `ArrayBuffer` class.

[1] http://kbhr.co/hs-cook
[2] http://kbhr.co/hs-arraybuffer

29
List Class

The Scala `List` class is a linear, immutable sequence. This means that it's a linked-list that you can't modify. Any time you want to add or remove `List` elements, you create a new `List` from an existing `List`.

29.1 Creating Lists

This is how you create an initial `List`:

```
val ints = List(1, 2, 3)
val names = List("Joel", "Chris", "Ed")
```

You can also explicitly declare the `List`'s type, if you prefer:

```
val ints: List[Int] = List(1, 2, 3)
val names: List[String] = List("Joel", "Chris", "Ed")
```

29.2 Adding elements to a List

Because `List` is immutable you can't add new elements to it. Instead, you create a new list by prepending or appending elements to an existing `List`. For instance, given this `List`:

```
val a = List(1,2,3)
```

You *prepend* single elements to a `List` with the `+:` method:

```
val b = 0 +: a
```

and prepend multiple elements with the `++:` method:

```
val b = List(-1, 0) ++: a
```

The REPL shows how this works:

```
scala> val b = 0 +: a
b: List[Int] = List(0, 1, 2, 3)

scala> val b = List(-1, 0) ++: a
b: List[Int] = List(-1, 0, 1, 2, 3)
```

You *append* single elements to it while creating a new list with the `:+` method:

```
val b = a :+ 4
```

and append multiple elements with the `++` method:

```
val b = a ++ Vector(4, 5)
```

Again the REPL shows how this works:

```
scala> val a = List(1,2,3)
a: List[Int] = List(1, 2, 3)

scala> val b = a :+ 4
b: List[Int] = List(1, 2, 3, 4)

scala> val b = a ++ Vector(4, 5)
b: List[Int] = List(1, 2, 3, 4, 5)
```

Because `List` is a singly-linked list, you should really only *prepend* elements to it; appending elements to it is significantly slower, especially when you work with large sequences.

> If you want to prepend and append elements to an immutable sequence, use `Vector` instead.

Here's a summary of those examples:

```
val a = List(1,2,3)

// prepend
val b = 0 +: a
val b = List(-1, 0) ++: a

// append
val b = a :+ 4
val b = a ++ Vector(4, 5)
```

> Because `List` is a linked-list class, you also shouldn't try to access the elements of large lists by their index value. For instance, if you have a `List` with one million elements in it, accessing an element like `myList(999999)` will take a long time. If you want to access elements like this, use a `Vector` or `ArrayBuffer` instead.

29.3 How to remember the method names

One way I remember those method names is to think that the `:` character represents the side that the `List` is on, so when I use `+:` I know that the `List` needs to be on the right, like this:

```
0 +: a
```

and when I use `:+` I know the `List` needs to be on the left:

```
a :+ 4
```

There are more technical ways to think about this, but I find this to be a good way to remember the method names.

One good thing about these method names is that they're consistent. The same names are used with other immutable sequence classes, such as `Seq` and `Vector`.

29.4 A bit of history

If you're interested in a little bit of history, the Scala `List` class is similar to the `List` class from the Lisp programming language. Indeed, in addition to the way I showed

how to create a `List` earlier, you can also create a `List` like this:

```
val list = 1 :: 2 :: 3 :: Nil
```

The REPL shows how this works:

```
scala> val list = 1 :: 2 :: 3 :: Nil
list: List[Int] = List(1, 2, 3)
```

This works because a `List` is a singly-linked list that ends with the `Nil` element.

> For much more information about this see my book, Functional Programming, Simplified[1].

29.5 See also

For more information on how to work with `Lists`, see these resources:

- Scala Cookbook[2]
- My big page of List class examples[3]

[1] http://kbhr.co/hs-fps
[2] http://kbhr.co/hs-cook
[3] http://kbhr.co/hs-list

30
Vector Class

`Vector` is an indexed, immutable sequence. The "indexed" part of the description means that you can access `Vector` elements very rapidly by their index value, such as accessing `listOfPeople(999999)`.

In general, except for the difference that `Vector` is indexed and `List` is not (and `List` ends with a `Nil` element), the two classes have the same methods, so I'll run through these examples quickly.

Here are a few ways to create a `Vector`:

```
val nums = Vector(1, 2, 3, 4, 5)

val strings = Vector("one", "two")

val peeps = Vector(
    Person("Bert"),
    Person("Ernie"),
    Person("Grover")
)
```

Because `Vector` is immutable, you can't add new elements to it. Instead, you create a new sequence by appending or prepending elements to an existing `Vector`. For instance, given this `Vector`:

```
val a = Vector(1,2,3)
```

you *append* elements with the `:+` and `++` methods:

```
val b = a :+ 4
val b = a ++ Vector(4, 5)
```

The REPL shows how this works:

```
scala> val a = Vector(1,2,3)
a: Vector[Int] = List(1, 2, 3)

scala> val b = a :+ 4
b: Vector[Int] = List(1, 2, 3, 4)

scala> val b = a ++ Vector(4, 5)
b: Vector[Int] = List(1, 2, 3, 4, 5)
```

You *prepend* elements with the `+:` and `++:` methods:

```
val b = 0 +: a
val b = Vector(-1, 0) ++: a
```

Once again the REPL shows how this works:

```
scala> val b = 0 +: a
b: Vector[Int] = List(0, 1, 2, 3)

scala> val b = Vector(-1, 0) ++: a
b: Vector[Int] = List(-1, 0, 1, 2, 3)
```

Because `Vector` is an indexed sequence you can prepend and append elements to it, and the speed of both approaches should be similar.

Finally, you loop over `Vector` elements just like you do with an `ArrayBuffer` or `List`:

```
val names = Vector("Joel", "Chris", "Ed")
for (name <- names) println(name)
```

See these resources for more information on how to work with `Vector`:

- Scala Cookbook[1]
- My big page of Vector class examples[2]

[1] http://kbhr.co/hs-cook
[2] http://kbhr.co/hs-vector

31
Map Class

The Scala Map class documentation[1] describes a `Map` as an iterable sequence that consists of pairs of keys and values. A simple `Map` looks like this:

```
val states = Map(
    "AK" -> "Alaska",
    "IL" -> "Illinois",
    "KY" -> "Kentucky"
)
```

Scala has both mutable and immutable `Map` classes. In this lesson I'll show how to use the *mutable* class. (Please see the Scala Cookbook[2] for examples of the immutable `Map` class.)

31.1 Creating a mutable Map

To use the mutable `Map` class, first import it:

```
import scala.collection.mutable.Map
```

Then create a `Map` like this:

```
val states = collection.mutable.Map("AK" -> "Alaska")
```

31.2 Adding elements to a Map

Now you can add a single element to the `Map` with the `+=` method:

[1] https://docs.scala-lang.org/overviews/collections/maps.html
[2] http://kbhr.co/hs-cook

```
states += ("AL" -> "Alabama")
```

Add multiple elements using `+=`:

```
states += ("AR" -> "Arkansas", "AZ" -> "Arizona")
```

Add elements from another `Map` using `++=`:

```
states ++= Map("CA" -> "California", "CO" -> "Colorado")
```

The REPL shows how these examples work:

```
scala> val states = collection.mutable.Map("AK" -> "Alaska")
states: scala.collection.mutable.Map[String,String] = Map(AK -> Alaska)

scala> states += ("AL" -> "Alabama")
res0: states.type = Map(AL -> Alabama, AK -> Alaska)

scala> states += ("AR" -> "Arkansas", "AZ" -> "Arizona")
res1: states.type = Map(AZ -> Arizona, AL -> Alabama, AR -> Arkansas, AK -> Alaska)

scala> states ++= Map("CA" -> "California", "CO" -> "Colorado")
res2: states.type = Map(CO -> Colorado, AZ -> Arizona, AL -> Alabama,
      CA -> California, AR -> Arkansas, AK -> Alaska)
```

31.3 Removing elements from a Map

You remove elements from a `Map` using the `-=` and `--=` methods while specifying the desired key values, as shown in these examples:

```
states -= "AR"
states -= ("AL", "AZ")
states --= List("AL", "AZ")
```

The REPL shows how these examples work:

```
scala> states -= "AR"
res3: states.type = Map(CO -> Colorado, AZ -> Arizona, AL -> Alabama,
      CA -> California, AK -> Alaska)
```

```
scala> states -= ("AL", "AZ")
res4: states.type = Map(CO -> Colorado, CA -> California, AK -> Alaska)

scala> states --= List("AL", "AZ")
res5: states.type = Map(CO -> Colorado, CA -> California, AK -> Alaska)
```

31.4 Updating Map elements

With a mutable `Map`, you update `Map` elements by reassigning their key to a new value:

```
states("AK") = "Alaska, A Really Big State"
```

The REPL shows how this works:

```
scala> states("AK") = "Alaska, A Really Big State"

scala> states
res6: scala.collection.mutable.Map[String,String] = Map(CO -> Colorado,
      CA -> California, AK -> Alaska, A Really Big State)
```

31.5 Traversing a Map

There are several different ways to iterate over the elements in a map. Given a sample map:

```
val ratings = Map(
    "Lady in the Water"-> 3.0,
    "Snakes on a Plane"-> 4.0,
    "You, Me and Dupree"-> 3.5
)
```

my preferred way to loop over all of the map elements is with this `for` loop syntax:

```
for ((k,v) <- ratings) println(s"key: $k, value: $v")
```

Using a `match` expression with the `foreach` method is also very readable:

```
ratings.foreach {
    case(movie, rating) => println(s"key: $movie, value: $rating")
}
```

> The `ratings` map data in this example comes from the old-but-good book, Programming Collective Intelligence[3].

31.6 See also

There are other ways to work with Scala maps, and different map classes for different needs, including `ListMap`, `SortedMap`, and more. See these resources for more information on how to work with Scala maps:

- Scala Cookbook[4]
- My big page of Map class examples[5]

[3] http://amzn.to/2CPLrb6
[4] http://kbhr.co/hs-cook
[5] http://kbhr.co/hs-map

32

Set Class

The Scala Set class[1] is an iterable collection with no duplicate elements.

Scala has both mutable and immutable `Set` classes. In this lesson I'll show how to use the *mutable* class. (Please see the Scala Cookbook[2] for examples of the immutable `Set` class.)

32.1 Adding elements to a Set

To use a mutable `Set`, first import it:

```
val set = scala.collection.mutable.Set[Int]()
```

You add elements to a mutable `Set` with the `+=`, `++=`, and `add` methods. Here are examples of the first two methods:

```
set += 1
set += (2, 3)
set += 2
set ++= Vector(4, 5)
```

The REPL shows how those examples work:

```
scala> set += 1
res0: scala.collection.mutable.Set[Int] = Set(1)

scala> set += (2, 3)
res1: scala.collection.mutable.Set[Int] = Set(1, 2, 3)
```

[1] https://docs.scala-lang.org/overviews/collections/sets.html
[2] http://kbhr.co/hs-cook

```
scala> set += 2
res2: scala.collection.mutable.Set[Int] = Set(1, 2, 3)

scala> set ++= Vector(4, 5)
res3: scala.collection.mutable.Set[Int] = Set(1, 5, 2, 3, 4)
```

Notice that the second time I try to add the value 2 to the set, the attempt is quietly ignored.

The `add` method is unique in that it returns `true` if an element is added to a set, and `false` if it wasn't added. The REPL shows how it works:

```
scala> set.add(6)
res4: Boolean = true

scala> set.add(5)
res5: Boolean = false
```

32.2 Deleting elements from a Set

You remove elements from a set using the `-=` and `--=` methods, as shown in the following examples:

```
scala> val set = scala.collection.mutable.Set(1, 2, 3, 4, 5)
set: scala.collection.mutable.Set[Int] = Set(2, 1, 4, 3, 5)

// one element
scala> set -= 1
res0: scala.collection.mutable.Set[Int] = Set(2, 4, 3, 5)

// two or more elements (-= has a varags field)
scala> set -= (2, 3)
res1: scala.collection.mutable.Set[Int] = Set(4, 5)

// multiple elements defined in another sequence
scala> set --= Array(4,5)
res2: scala.collection.mutable.Set[Int] = Set()
```

There are more methods for working with sets, including `clear` and `remove`, as shown in these examples:

```
scala> val set = scala.collection.mutable.Set(1, 2, 3, 4, 5)
set: scala.collection.mutable.Set[Int] = Set(2, 1, 4, 3, 5)

// clear
scala> set.clear

scala> set
res0: scala.collection.mutable.Set[Int] = Set()

// remove
scala> val set = scala.collection.mutable.Set(1, 2, 3, 4, 5)
set: scala.collection.mutable.Set[Int] = Set(2, 1, 4, 3, 5)

scala> set.remove(2)
res1: Boolean = true

scala> set
res2: scala.collection.mutable.Set[Int] = Set(1, 4, 3, 5)

scala> set.remove(40)
res3: Boolean = false
```

32.3 More Set classes

Scala has several more `Set` classes, including `SortedSet`, `LinkedHashSet`, and more. Please see the Scala Set class documentation[3] and the Scala Cookbook[4] for more details.

[3] https://docs.scala-lang.org/overviews/collections/sets.html
[4] http://kbhr.co/hs-cook

33

Anonymous Functions

Earlier in this book I showed that you can create a list of integers like this:

```
val ints = List(1,2,3)
```

When you want to create a larger list, you can also create them with the `List` class `range` method, like this:

```
val ints = List.range(1, 10)
```

That code creates `ints` as a list of integers whose values range from 1 to 10. You can see the result in the REPL:

```
scala> val ints = List.range(1, 10)
x: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

In this lesson I'll use lists like these to demonstrate a feature of functional programming known as *anonymous functions*. It will help to understand how these work before I demonstrate the most common Scala collections methods in the following lessons.

33.1 Examples

An anonymous function is like a little mini-function. For example, given a list like this:

```
val ints = List(1,2,3)
```

I can create a new list by doubling each element in `ints`, like this:

```
val doubledInts = ints.map(_ * 2)
```

Here's what that example looks like in the REPL:

```
scala> val doubledInts = ints.map(_ * 2)
doubledInts: List[Int] = List(2, 4, 6)
```

As that shows, `doubledInts` is now the list `List(2, 4, 6)`. In this example, this code is an anonymous function:

```
_ * 2
```

This is a shorthand way of saying, "Multiply an element by 2."

Once you're comfortable with Scala, this is a common way to write anonymous functions. But if you don't like this syntax you can also write anonymous functions in a longer format. I wrote the previous example like this:

```
val doubledInts = ints.map(_ * 2)
```

You can also write it like this:

```
val doubledInts = ints.map((i: Int) => i * 2)
val doubledInts = ints.map(i => i * 2)
```

All three lines have exactly the same meaning: double each element in `ints` to create a new list, `doubledInts`.

> The _ character in Scala is something of a wildcard character. You'll see it used in several different places. In this case it's a shorthand way of saying, "An element from the list, `ints`."

Before I go too much further, if you're coming to Scala from Java it may help to know that this `map` example is the equivalent of this Java code:

```
List<Integer> ints = new ArrayList<>(Arrays.asList(1, 2, 3));

// the `map` process
List<Integer> doubledInts = new ArrayList<Integer>();
for (int i: ints) {
    doubledInts.add(i * 2);
}
```

The map example shown is also the same as this Scala code:

```
val doubledInts = for (i <- ints) yield i * 2
```

33.2 Anonymous functions with the filter method

Another good way to show anonymous functions is with the filter method of the List class. Given this List:

```
val ints = List.range(1, 10)
```

This is how you create a new list of all integers whose value is greater than 5:

```
val x = ints.filter(_ > 5)
```

This is how you create a new list whose values are all less than 5:

```
val x = ints.filter(_ < 5)
```

And as a little more complicated example, this is how you create a new list that contains only even values, by using the modulus operator:

```
val x = ints.filter(_ % 2 == 0)
```

If that's a little confusing, remember that this example can also be written in these other ways:

```
val x = ints.filter((i: Int) => i % 2 == 0)
val x = ints.filter(i => i % 2 == 0)
```

This is what the previous examples look like in the REPL:

```
scala> val x = ints.filter(_ > 5)
x: List[Int] = List(6, 7, 8, 9)

scala> val x = ints.filter(_ < 5)
x: List[Int] = List(1, 2, 3, 4)

scala> val x = ints.filter(_ % 2 == 0)
x: List[Int] = List(2, 4, 6, 8)
```

33.3 Key points

The key points of this lesson are:

- You can write anonymous functions as little snippets of code
- You can use them with standard methods on collections classes, like `map` and `filter`
- With these little snippets of code and powerful methods like those, you can create a lot of functionality with very little code

The Scala collections classes contain many methods like `map` and `filter`, and they're a powerful way to create very expressive code.

33.4 Bonus: Digging a little deeper

You may be wondering how the `map` and `filter` examples work. The short answer is that when `map` is invoked on a list of integers — a `List[Int]` to be more precise — `map` expects to receive a function that transforms one `Int` value into another `Int` value. Because `map` expects a function (or method) that transforms one `Int` to another `Int`, this approach also works:

```
val ints = List(1,2,3)
def double(i: Int): Int = i * 2   //a method that doubles an Int
val doubledInts = ints.map(double)
```

The last two lines of that example are the same as this:

```
val doubledInts = ints.map(_ * 2)
```

Similarly, when called on a `List[Int]`, the `filter` method expects to receive a function that takes an `Int` and returns a `Boolean` value. Therefore, given a method that's defined like this:

```
def lessThanFive(i: Int): Boolean = if (i < 5) true else false
```

or more concisely, like this:

```
def lessThanFive(i: Int): Boolean = (i < 5)
```

this `filter` example:

```
val ints = List.range(1, 10)
val y = ints.filter(lessThanFive)
```

is the same as this example:

```
val y = ints.filter(_ < 5)
```

Anonymous functions are a big part of Scala, and I write much more about the details behind this in both of my books, the Scala Cookbook[1], and Functional Programming, Simplified[2].

[1] http://kbhr.co/hs-cook
[2] http://kbhr.co/hs-fps

34

Common Methods on Sequences

A great strength of the Scala collections classes is that they come with dozens of pre-built methods. The benefit of this is that you no longer need to write custom `for` loops every time you need to work on a collection. If that doesn't sound like enough of a benefit, it also means that you no longer have to read custom `for` loops written by other developers. ;)

Because there are so many methods available to you, I'm not going to show them all here. (I do that in the Scala Cookbook[1].) Instead, I'll just show how to use some of the most commonly-used methods, including:

- `map`
- `filter`
- `foreach`
- `head`
- `tail`
- `take`, `takeWhile`
- `drop`, `dropWhile`
- `find`
- `reduce`, `fold`

The methods I'll show work on all of the sequential collections classes, including `Array`, `ArrayBuffer`, `List`, `Vector`, etc., but in these examples I'll use a `List` unless otherwise specified.

[1] http://kbhr.co/hs-cook

34.1 Note: The methods don't mutate the list

As a very important note, none of these methods mutate the list that they're called on. They all work in a functional style, which means that they return a new sequence with the modified results.

34.2 Sample lists

In the following examples I'll use these lists:

```
val nums = (1 to 10).toList
val names = List("joel", "ed", "chris", "maurice")
```

This is what these lists look like in the REPL:

```
scala> val nums = (1 to 10).toList
nums: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> val names = List("joel", "ed", "chris", "maurice")
names: List[String] = List(joel, ed, chris, maurice)
```

34.3 map

The `map` method steps through each element in the existing list, applies the algorithm you supply to each element, and then returns a new list with all of the modified elements.

Here's an example of the `map` method being applied to the `nums` list:

```
scala> val doubles = nums.map(_ * 2)
doubles: List[Int] = List(2, 4, 6, 8, 10, 12, 14, 16, 18, 20)
```

As I wrote in the lesson on anonymous functions, you can also write the anonymous function like this:

```
scala> val doubles = nums.map(i => i * 2)
doubles: List[Int] = List(2, 4, 6, 8, 10, 12, 14, 16, 18, 20)
```

However, in this lesson I'll always use the first, shorter form.

With that background, here are a few more examples of the `map` method being applied to the `nums` and `names` lists:

```
scala> val capNames = names.map(_.capitalize)
capNames: List[String] = List(Joel, Ed, Chris, Maurice)

scala> val doubles = nums.map(_ * 2)
doubles: List[Int] = List(2, 4, 6, 8, 10, 12, 14, 16, 18, 20)

scala> val lessThanFive = nums.map(_ < 5)
lessThanFive: List[Boolean] = List(true, true, true, true, false, false, false,
    false, false, false)
```

As that last example shows, it's perfectly legal — and very common — to use `map` to return a list with a type (`List[Boolean]`) that's different than the original type (`List[Int]`).

34.4 filter

The `filter` method creates a new, filtered list from the given list. Here are a few examples:

```
scala> val lessThanFive = nums.filter(_ < 5)
lessThanFive: List[Int] = List(1, 2, 3, 4)

scala> val evens = nums.filter(_ % 2 == 0)
evens: List[Int] = List(2, 4, 6, 8, 10)

scala> val shortNames = names.filter(_.length <= 4)
shortNames: List[String] = List(joel, ed)
```

34.5 foreach

The `foreach` method is used to loop over all elements in a collection. As I mentioned previously, `foreach` is used for side-effects, such as printing information. Here's an example with the `names` list:

```
scala> names.foreach(println)
joel
ed
chris
maurice
```

The `nums` list is a little long, so I don't want to print out all of those elements. But a great thing about Scala's approach is that you can chain methods together to solve problems. For example, this is one way to print the first three elements from `nums`:

```
nums.filter(_ < 4).foreach(println)
```

The REPL shows the result:

```
scala> nums.filter(_ < 4).foreach(println)
1
2
3
```

34.6 head

The `head` method comes from Lisp and other functional programming languages. It's used to print the first element (the head element) of a list:

```
scala> nums.head
res0: Int = 1

scala> names.head
res1: String = joel
```

Because a `String` is a sequence of characters, you can also treat it like a list. This is how `head` works on strings:

```
scala> "foo".head
res2: Char = f

scala> "bar".head
res3: Char = b
```

34.7 `tail`

The `tail` method also comes from Lisp and other functional programming languages. It's used to print every element in a list after the head element. Here are a few examples:

```
scala> nums.tail
res0: List[Int] = List(2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> names.tail
res1: List[String] = List(ed, chris, maurice)
```

Just like `head`, `tail` also works on strings:

```
scala> "foo".tail
res2: String = oo

scala> "bar".tail
res3: String = ar
```

34.8 `take`, `takeWhile`

The `take` and `takeWhile` methods give you a nice way of "taking" the elements out of a list that you want to create a new list. This is `take`:

```
scala> nums.take(1)
res0: List[Int] = List(1)

scala> nums.take(2)
res1: List[Int] = List(1, 2)

scala> names.take(1)
res2: List[String] = List(joel)

scala> names.take(2)
res3: List[String] = List(joel, ed)
```

And this is `takeWhile`:

```
scala> nums.takeWhile(_ < 5)
res4: List[Int] = List(1, 2, 3, 4)

scala> names.takeWhile(_.length < 5)
res5: List[String] = List(joel, ed)
```

34.9 `drop`, `dropWhile`

`drop` and `dropWhile` are essentially the opposite of `take` and `takeWhile`. This is `drop`:

```
scala> nums.drop(1)
res0: List[Int] = List(2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> nums.drop(5)
res1: List[Int] = List(6, 7, 8, 9, 10)

scala> names.drop(1)
res2: List[String] = List(ed, chris, maurice)

scala> names.drop(2)
res3: List[String] = List(chris, maurice)
```

And this is `dropWhile`:

```
scala> nums.dropWhile(_ < 5)
res4: List[Int] = List(5, 6, 7, 8, 9, 10)

scala> names.dropWhile(_ != "chris")
res5: List[String] = List(chris, maurice)
```

34.10 `reduce`

When you hear the term, "map reduce," the "reduce" part refers to methods like `reduce`. It takes a function (or anonymous function) and applies that function to successive elements in a list, returning a single "reduced" value as its result.

The best way to explain `reduce` is to create a little helper method you can pass into it. For example, this is an `add` method that adds two integers together, but also gives us some nice debug output:

```
def add(x: Int, y: Int): Int = {
    val theSum = x + y
    println(s"received $x and $y, their sum is $theSum")
    theSum
}
```

Now, given that method and this list:

```
val a = List(1,2,3,4)
```

this is what happens when I pass the `add` method into `a.reduce`:

```
scala> a.reduce(add)
received 1 and 2, their sum is 3
received 3 and 3, their sum is 6
received 6 and 4, their sum is 10
res0: Int = 10
```

As that result shows, `reduce` uses `add` to reduce the list `a` into a single value, in this case, the sum of the integers in the list.

Once you get used to `reduce`, you'll write a "sum" algorithm like this:

```
scala> a.reduce(_ + _)
res0: Int = 10
```

Similarly, this is what a "product" algorithm looks like:

```
scala> a.reduce(_ * _)
res1: Int = 24
```

I know that might be a little mind-blowing if you've never seen it before, but because this is an "introduction" book, I'm going to leave it at that for now. For more details,

please see the Scala Cookbook[2], which has over 100 pages on the Scala collections, and Functional Programming, Simplified[3], which has a very detailed chapter on how methods like `reduce` and `fold` work.

> It's worth repeating that `reduce` is used to walk through each element in a collection with the algorithm you supply to *reduce* the collection down to a single value.

34.11 That's all for now

There are literally dozens of additional methods on the Scala sequential collections classes that will keep you from ever needing to write another `for` loop. However, because this is a simple introduction book, I won't cover them all. As mentioned, please see the *Scala Cookbook* and *Functional Programming, Simplified* for many more details.

[2] http://kbhr.co/hs-cook
[3] http://kbhr.co/hs-fps

35

Common Map Methods

In this lesson I'll demonstrate some of the most commonly used `Map` methods. I'll first show examples of an *immutable* map, and then show examples of a *mutable* map. I won't break the `Map` methods down into individual sections; I'll just provide a brief comment before each method.

35.1 Immutable Map examples

Given this immutable `Map`:

```
val m = Map(
    1 -> "a",
    2 -> "b",
    3 -> "c",
    4 -> "d"
)
```

Here are some examples of methods available to that `Map`:

```
// how to iterate over Map elements
scala> for ((k,v) <- m) printf("key: %s, value: %s\n", k, v)
key: 1, value: a
key: 2, value: b
key: 3, value: c
key: 4, value: d

// how to get the keys from a Map
scala> val keys = m.keys
keys: Iterable[Int] = Set(1, 2, 3, 4)
```

```
// how to get the values from a Map
scala> val values = m.values
values: Iterable[String] = MapLike.DefaultValuesIterable(a, b, c, d)

// how to test if a Map contains a value
scala> val contains3 = m.contains(3)
contains3: Boolean = true

// how to transform Map values
scala> val ucMap = m.transform((k,v) => v.toUpperCase)
ucMap: scala.collection.immutable.Map[Int,String] = Map(1 -> A, 2 -> B, 3 -> C, 4 -> D)

// how to filter a Map by its keys
scala> val twoAndThree = m.filterKeys(Set(2,3))
twoAndThree: scala.collection.immutable.Map[Int,String] = Map(2 -> b, 3 -> c)

// how to take the first two elements from a Map
scala> val firstTwoElements = m.take(2)
firstTwoElements: scala.collection.immutable.Map[Int,String] = Map(1 -> a, 2 -> b)
```

Note that the last example probably only makes sense for a sorted Map.

35.2 Mutable Map examples

Here are a few examples of methods that are available on the mutable `Map` class. Given this initial mutable `Map`:

```
val states = scala.collection.mutable.Map(
    "AL" -> "Alabama",
    "AK" -> "Alaska"
)
```

Here are some things you can do with a mutable `Map`:

```
// add elements with +=
states += ("AZ" -> "Arizona")
states += ("CO" -> "Colorado", "KY" -> "Kentucky")
```

```
// remove elements with -=
states -= "KY"
states -= ("AZ", "CO")

// update elements by reassigning them
states("AK") = "Alaska, The Big State"

// retain elements by supplying a function that operates on
// the keys and/or values
states.retain((k,v) => k == "AK")
```

35.3 See also

These resources show many more things you can do with Scala maps:

- The official Scala `Map` documentation[1]
- My big page of Map class examples[2]

[1] https://docs.scala-lang.org/overviews/collections/maps.html
[2] http://kbhr.co/hs-map

36

A Few Miscellaneous Items

In the next several lessons I'll cover a few miscellaneous items about Scala:

- Tuples
- An example of using Java's Swing GUI library with Scala
- A Scala OOP example of a pizza restaurant order-entry system

37
Tuples

A *tuple* is a neat little class that gives you a simple way to store objects of different types in a container. Rather than having to create a class to store things in, like this:

```
class SomeThings(i: Int, s: String, p: Person)
```

you can just create a tuple like this:

```
val t = (3, "Three", new Person("Al"))
```

As shown, just put some elements inside parentheses, and you have a tuple. Scala tuples can contain between two and 22 items, and I find them useful for those times when I just need to combine a few things together, and I don't want the baggage of having to define a class, especially when that class feels a little "artificial" or phony.

> Technically, Scala has classes named `Tuple2`, `Tuple3` … up to `Tuple22`. As a practical matter you rarely need to know this, but I find that it's also good to know what's going on under the hood.

37.1 A few more details

Here's a two-element tuple:

```
scala> val d = ("Maggie", 30)
d: (String, Int) = (Maggie,30)
```

Notice that it contains two different types, `String` and `Int`.

Next, given this `Person` class:

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

Here's a three-element tuple:

```
scala> val t = (3, "Three", new Person("Melissa"))
t: (Int, java.lang.String, Person) = (3,Three,Person(Melissa))
```

There are a few ways to access tuple elements. One approach is to access them by element number, where the number is preceded by an underscore:

```
scala> t._1
res1: Int = 3

scala> t._2
res2: java.lang.String = Three

scala> t._3
res3: Person = Person(Melissa)
```

Another cool approach is to access them like this:

```
scala> val(x, y, z) = (3, "Three", new Person("Melissa"))
x: Int = 3
y: String = Three
z: Person = Person(Melissa)
```

Technically this approach involves a form of pattern-matching, and it's a great way to assign tuple elements to variables.

37.2 Returning a tuple from a method

A place where this last technique is nice is when you want to return multiple values from a method. For example, here's a method that returns a tuple:

```
def getStockInfo = {
    // other code here ...
    ("NFLX", 100.00, 101.00)  // this is a Tuple3
}
```

Now you can call that method and assign variable names to the return values:

```
val (symbol, currentPrice, bidPrice) = getStockInfo
```

The REPL demonstrates how this works:

```
scala> val (symbol, currentPrice, bidPrice) = getStockInfo
symbol: String = NFLX
currentPrice: Double = 100.0
bidPrice: Double = 101.0
```

I don't use tuples a great deal, but for cases like this where it feels like overkill to create a class for the method's return type, I find them convenient.

37.3 Tuples aren't collections

Technically, tuples aren't collections classes, they're just a convenient little container. Because they aren't a collection, they don't have methods like `map`, `filter`, etc.

38

Scala and Swing

Scala works with Java Swing classes like `JFrame`, `JTextArea`, etc., very easily. Here's an example of a Scala application that opens a `JFrame`, adds a few components to it, and then displays it:

```
import java.awt.BorderLayout
import java.awt.Dimension
import javax.swing.{JFrame, JScrollPane, JTextArea}

object SwingExample extends App {

    val textArea = new JTextArea
    textArea.setText("Hello, Swing world")
    val scrollPane = new JScrollPane(textArea)

    val frame = new JFrame("Hello, Swing")
    frame.getContentPane.add(scrollPane, BorderLayout.CENTER)
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)
    frame.setSize(new Dimension(600, 400))
    frame.setLocationRelativeTo(null)
    frame.setVisible(true)

}
```

To see how that code works, save it to a file named *SwingExample.scala*, then compile it:

```
$ scalac SwingExample.scala
```

and run it:

```
$ scala SwingExample
```

You should see that it opens a `JFrame` with a `JTextArea` inside a `JScrollPane`.

I've written a few Scala/Swing applications totaling thousands of lines of code, and I haven't had any compatibility problems.

> There's also a Scala project known as Scala Swing[1], which is something different. That project is an effort to make Swing GUI code look more like it would have looked if someone knew Scala and then wrote a GUI framework on top of it.

38.1 Experiment with the code yourself

To experiment with this on your own, see the *SwingExample* project in this book's GitHub repository, which you can find at this URL:

- github.com/alvinj/HelloScalaExamples[2]

If you know how to use the Scala Build Tool (SBT)[3] you can use it to run this application, otherwise you can compile and run the source code file that's in the project's root directory using `scalac` and `scala`, as shown above.

Note: SBT is introduced later in this book.

[1] https://github.com/scala/scala-swing
[2] https://github.com/alvinj/HelloScalaExamples
[3] http://www.scala-sbt.org/

39
An OOP Example

In this lesson I share an example of an OOP-style application written with Scala. The example shows code you might write for a pizza store order-entry system.

39.1 Source code for the example

To experiment with this on your own, please see the *PizzaOopExample* project in this book's GitHub repository:

- github.com/alvinj/HelloScalaExamples[1]

To compile this project it will help to either a) use Eclipse or IntelliJ IDEA, or b) know how to use the Scala Build Tool (SBT)[2], which I introduce later in this book.

39.2 A few enumerations

As I showed earlier in the book, you create enumerations in Scala like this:

```
sealed trait Topping
case object Cheese extends Topping
case object Pepperoni extends Topping
case object Sausage extends Topping
case object Mushrooms extends Topping
case object Onions extends Topping
```

[1] https://github.com/alvinj/HelloScalaExamples
[2] http://www.scala-sbt.org/

```
sealed trait CrustSize
case object SmallCrustSize extends CrustSize
case object MediumCrustSize extends CrustSize
case object LargeCrustSize extends CrustSize

sealed trait CrustType
case object RegularCrustType extends CrustType
case object ThinCrustType extends CrustType
case object ThickCrustType extends CrustType
```

Even though I haven't discussed sealed traits or case objects, I think you can still figure out how this code works.

39.3 A few classes

Given those enumerations, I can now start to create a few pizza-related classes for my order-entry system. First, here's a `Pizza` class:

```
import scala.collection.mutable.ArrayBuffer

class Pizza (
    var crustSize: CrustSize,
    var crustType: CrustType,
    var toppings: ArrayBuffer[Topping]
)
```

Next, here's an `Order` class, where an `Order` consists of a mutable list of pizzas and a `Customer`:

```
class Order (
    var pizzas: ArrayBuffer[Pizza],
    var customer: Customer
)
```

Here's a `Customer` class to work with that code:

```
class Customer (
    var name: String,
    var phone: String,
    var address: Address
)
```

Finally, here's an `Address` class:

```
class Address (
    var street1: String,
    var street2: String,
    var city: String,
    var state: String,
    var zipCode: String
)
```

So far those classes just look like data structures — like a `struct` in C — so let's add a little behavior to them.

39.4 Adding behavior to Pizza

An OOP-style `Pizza` class needs a few methods to add and remove toppings, and adjust the crust size and type. Here's a `Pizza` class with a few methods added to handle those behaviors:

```
class Pizza (
    var crustSize: CrustSize,
    var crustType: CrustType,
    val toppings: ArrayBuffer[Topping]
) {

    def addTopping(t: Topping): Unit = { toppings += t }
    def removeTopping(t: Topping): Unit = { toppings -= t }
    def removeAllToppings(): Unit = { toppings.clear() }

}
```

You can also argue that a pizza should be able to calculate its own price, so here's another method you *could* add to that class:

```
def getPrice(
    toppingsPrices: Map[Topping, Int],
    crustSizePrices: Map[CrustSize, Int],
    crustTypePrices: Map[CrustType, Int]
): Int = ???
```

I'm not going to finish implementing this method, but note that it's a perfectly legal method as written: it's legal to use the `???` syntax for the body of a method. Teachers use it when they don't want to write out the body of a method, and I sometimes use it in my own code to say, "This is what my method signature looks like, but I don't want to write the method body yet." A great thing for those times is that this code compiles.

> But don't *call* that method. If you do, you'll get a `NotImplementedError`, which is very descriptive of the situation.

39.5 Adding behavior to Order

You should be able to do a few things with an order, including:

- Add and remove pizzas
- Update customer information
- Get the order price

Here's an `Order` class that lets you do those things:

```
class Order (
    val pizzas: ArrayBuffer[Pizza],
    var customer: Customer
) {

    def addPizza(p: Pizza): Unit = {
        pizzas += p
    }
```

```
    def removePizza(p: Pizza): Unit = {
        pizzas -= p
    }

    // need to implement these
    def getBasePrice(): Int = ???
    def getTaxes(): Int = ???
    def getTotalPrice(): Int = ???

}
```

Once again I'm not concerned with how to calculate the price of an order — I'm trying to keep things simple — so I leave that as an exercise for the reader.

39.6 Testing those classes

You can use a little "driver" class to test those classes. With the addition of a `printOrder` method on the `Order` class and a `toString` method in the `Pizza` class, you'll find that this code works as advertised:

```
import scala.collection.mutable.ArrayBuffer

object MainDriver extends App {

    val p1 = new Pizza (
        MediumCrustSize,
        ThinCrustType,
        ArrayBuffer(Cheese)
    )

    val p2 = new Pizza (
        LargeCrustSize,
        ThinCrustType,
        ArrayBuffer(Cheese, Pepperoni, Sausage)
    )
```

```
    val address = new Address (
        "123 Main Street",
        "Apt. 1",
        "Talkeetna",
        "Alaska",
        "99676"
    )

    val customer = new Customer (
        "Alvin Alexander",
        "907-555-1212",
        address
    )

    val o = new Order(
        ArrayBuffer(p1, p2),
        customer
    )

    o.addPizza(
        new Pizza (
            SmallCrustSize,
            ThinCrustType,
            ArrayBuffer(Cheese, Mushrooms)
        )
    )

    // print the order
    o.printOrder

}
```

Once again, I encourage you to clone the source code for this project from its Github repository and make changes to it until you're comfortable with how it all works.

40

A Scala + JavaFX Example

If you're ready to tackle a slightly larger Scala application, I wrote a little "Notes" application using Scala and JavaFX to accompany this book. If you'd like to see how it works, I put a short two-minute video introduction at this URL:

- kbhr.co/hs-javafx[1]

If you want to dig into that application, the source code for the project is at this Github URL:

- github.com/alvinj/NotesWithScalaJavaFX[2]

I won't describe the application here because the video is the best way to show how it works, and I include a lengthy README file with the source code. So, please see that Github project for more information.

[1] http://kbhr.co/hs-javafx
[2] https://github.com/alvinj/NotesWithScalaJavaFX

41

SBT and ScalaTest

In the next few lessons I'll introduce a couple of tools that are commonly used in Scala projects:

- The The Scala Build Tool (SBT)[1]
- ScalaTest[2], a code testing framework

I'll begin by showing how to use SBT, and then I'll show how to use ScalaTest and SBT together.

[1] http://www.scala-sbt.org/
[2] http://www.scalatest.org/

42

The Scala Build Tool (SBT)

You can use several different tools to build your Scala projects, including Ant, Maven, Gradle, and more. I generally use a tool named SBT[1]. It was the first build tool that was specifically created for Scala, and these days it's supported by Lightbend[2], the company that was co-founded by Scala creator Martin Odersky, and also maintains Akka[3], the Play Framework[4], Scala, and more.

If you haven't already installed SBT, here's a link to its download page[5].

42.1 The SBT directory structure

Like Maven, SBT uses a standard project directory structure. If you use that directory structure I think you'll find that it's relatively simple to build your first projects.

Underneath your main project directory SBT expects a structure that looks like this:

(see the top of the next page)

[1] http://www.scala-sbt.org/
[2] https://www.lightbend.com/
[3] https://akka.io/
[4] https://www.playframework.com/
[5] http://www.scala-sbt.org/download.html

```
build.sbt
lib/
project/
src/
-- main/
   |-- java/
   |-- resources/
   |-- scala/
-- test/
   |-- java/
   |-- resources/
   |-- scala/
-- target/
```

42.2 Creating a first SBT project

Creating this directory structure is pretty simple, and I usually use a shell script I wrote named sbtmkdirs[6] to create new projects. But you don't have to use that script; assuming that you're using a Unix/Linux system, you can just use these commands to create your first SBT project directory structure:

```
mkdir HelloWorld
cd HelloWorld
mkdir -p src/{main,test}/{java,resources,scala}
mkdir lib project target
```

If you run a `find .` command after running those commands, you should see this result:

```
$ find .
.
./lib
./project
./src
./src/main
```

[6] https://alvinalexander.com/sbtmkdirs

```
./src/main/java
./src/main/resources
./src/main/scala
./src/test
./src/test/java
./src/test/resources
./src/test/scala
./target
```

If you see that, you're in great shape for the next step.

> There are other ways to create the files and directories for an SBT project. One way is to use the `sbt new` command, which is documented here[7]. I don't show that approach here because some of the files it creates are too complicated for an introduction like this.

42.3 Creating a first *build.sbt* file

Now you only need two more things to run a "Hello, world" project:

- A *build.sbt* file in the root directory of the project
- A *Hello.scala* file

For a little project like this, the *build.sbt* file only needs to contain a few lines, like this:

```
name := "HelloWorld"

version := "1.0"

scalaVersion := "2.12.4"
```

Each line should be separated by a blank line, as shown. Because SBT projects use a standard directory structure, SBT already knows everything else it needs to know.

Now you just need to add a little "Hello, world" program.

[7] http://www.scala-sbt.org/1.x/docs/Hello.html

42.4 A "Hello, world" program

In large projects, all of your Scala source code files will go under the *src/main/scala* and *src/test/scala* directories, but for a little sample project like this, you can put your source code file in the root directory of your project. Go ahead and create a file named *HelloWorld.scala* in the root directory with these contents:

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

Now you can use SBT to run your project. Use the `sbt run` command to compile and run your project. When you do so, you'll see output that looks like this:

```
$ sbt run

Updated file /Users/al/Projects/Scala/Hello/project/build.properties
setting sbt.version to: 0.13.15
[warn] Executing in batch mode.
[warn]  For better performance, hit [ENTER] to switch to interactive mode, or
[warn]  consider launching sbt without any commands, or explicitly passing 'shell'
[info] Loading project definition from /Users/al/Projects/Scala/Hello/project
[info] Updating {file:/Users/al/Projects/Scala/Hello/project/}hello-build...
[info] Resolving org.fusesource.jansi#jansi;1.4 ...
[info] Done updating.
[info] Set current project to Hello (in build file:/Users/al/Projects/Scala/Hello/)
[info] Updating {file:/Users/al/Projects/Scala/Hello/}hello...
[info] Resolving jline#jline;2.14.5 ...
[info] Done updating.
[info] Compiling 1 Scala source to
  /Users/al/Projects/Scala/Hello/target/scala-2.12/classes...
[info] Running HelloWorld
Hello, world
[success] Total time: 4 s, completed Jan 6, 2018 3:08:59 PM
```

The first time you run `sbt` it can take a while to run, but after it downloads everything it needs it gets much faster. As the first comment in that output shows, it can also be faster to run SBT interactively. To do that, first run the `sbt` command by itself:

```
> sbt
[info] Loading project definition from /Users/al/Projects/Scala/Hello/project
[info] Set current project to Hello (in build file:/Users/al/Projects/Scala/Hello/)
```

Then execute its `run` command like this:

```
> run
[info] Running HelloWorld
Hello, world
[success] Total time: 0 s, completed Jan 6, 2018 3:12:21 PM
```

There, that's much faster.

If you type `help` at the SBT command prompt you'll see a list of other commands you can run. But for now, just type `exit` (or `CTRL-D`) to leave the SBT shell.

Note: I'll show more SBT examples in the lessons that follow.

42.5 See also

I don't cover them in this book, but other build tools you can use to build Scala projects are:

- Ant[8]
- CBT[9]
- Gradle[10]
- Maven[11]

[8] http://ant.apache.org/
[9] https://github.com/cvogt/cbt
[10] https://gradle.org/
[11] https://maven.apache.org/

43

Using ScalaTest with SBT

ScalaTest[1] is one of the main testing libraries for Scala projects, and in this lesson I'll show how to create a Scala project that uses ScalaTest, which you can compile, test, and run with SBT.

43.1 Creating the project directory structure

As with the previous lesson, create an SBT project directory structure for a project named *HelloScalaTest* using my sbtmkdirs[2] script, or with the following commands:

```
mkdir HelloScalaTest
cd HelloScalaTest
mkdir -p src/{main,test}/{java,resources,scala}
mkdir lib project target
```

43.2 Creating the *build.sbt* file

Next, create a *build.sbt* file in the root directory of your project with these contents:

```
name := "HelloScalaTest"

version := "1.0"

scalaVersion := "2.12.4"
```

[1] http://www.scalatest.org/
[2] https://alvinalexander.com/sbtmkdirs

```
libraryDependencies ++= Seq(
    "org.scalactic" %% "scalactic" % "3.0.4",
    "org.scalatest" %% "scalatest" % "3.0.4" % "test"
)
```

The first three lines of this file are basically the same as the example in the previous lesson, and the `libraryDependencies` lines tell SBT to include the dependencies (jar files) that are needed to run ScalaTest:


```
libraryDependencies ++= Seq(
    "org.scalactic" %% "scalactic" % "3.0.4",
    "org.scalatest" %% "scalatest" % "3.0.4" % "test"
)
```


> The ScalaTest documentation has always been good, and you can always find the up to date information on what those lines should look like on the Installing ScalaTest[3] page.

43.3 Create a Scala program

Next, create a Scala program that you can use to demonstrate ScalaTest. First, from the root directory of your project, create a directory under *src/main/scala* named *simpletest*:

```
$ mkdir src/main/scala/simpletest
```

Then, inside that directory, create a file named *Hello.scala* with these contents:

```
package simpletest

object Hello extends App {
    val p = new Person("Alvin Alexander")
    println(s"Hello ${p.name}")
}

class Person(var name: String)
```

[3] http://www.scalatest.org/install

There isn't much that can go wrong with that source code, but it provides a simple way to demonstrate ScalaTest. At this point you can run your project with the `sbt run` command. Your output should look like this:

```
> sbt run

[warn] Executing in batch mode.
[warn]  For better performance, hit [ENTER] to switch to interactive mode, or
[warn]  consider launching sbt without any commands, or explicitly passing 'shell'
...
...
[info] Compiling 1 Scala source to
  /Users/al/Projects/Scala/HelloScalaTest/target/scala-2.12/classes...
[info] Running simpletest.Hello
Hello Alvin Alexander
[success] Total time: 4 s, completed Jan 6, 2018 4:38:07 PM
```

Now let's create a ScalaTest file.

43.4 Your first ScalaTest tests

ScalaTest is very flexible, and there are a lot of different ways to write tests, but a simple way to get started is to write tests using the ScalaTest "FunSuite." To get started, create a directory named *simpletest* under the *src/test/scala* directory:

```
$ mkdir src/test/scala/simpletest
```

Next, create a file named *HelloTests.scala* in that directory with the following contents:

```
package simpletest

import org.scalatest.FunSuite

class HelloTests extends FunSuite {

    // test 1
    test("the name is set correctly in constructor") {
        val p = new Person("Barney Rubble")
        assert(p.name == "Barney Rubble")
```

```
    }

    // test 2
    test("a Person's name can be changed") {
        val p = new Person("Chad Johnson")
        p.name = "Ochocinco"
        assert(p.name == "Ochocinco")
    }

}
```

This file demonstrates the ScalaTest `FunSuite` approach. A few important points:

- Your class should extend `FunSuite`
- You create tests as shown, by giving each `test` a unique name
- At the end of each test you should call `assert` to test that a condition has been satisfied

Using ScalaTest like this is similar to JUnit, so if you're coming to Scala from Java, I hope it looks relatively familiar.

Now you can run these tests with the `sbt test` command. Skipping the first few lines of output, the result looks like this:

```
> sbt test
[info] Set current project to HelloScalaTest (in build
  file:/Users/al/Projects/Scala/HelloScalaTest/)
[info] HelloTests:
[info] - the name is set correctly in constructor
[info] - a Person's name can be changed
[info] Run completed in 277 milliseconds.
[info] Total number of tests run: 2
[info] Suites: completed 1, aborted 0
[info] Tests: succeeded 2, failed 0, canceled 0, ignored 0, pending 0
[info] All tests passed.
[success] Total time: 1 s, completed Jan 6, 2018 4:46:18 PM
```

43.5 TDD tests

What I just showed is a Test-Driven Development (TDD) style of testing with ScalaTest. In the next lesson I'll show how to write Behavior-Driven Development (BDD) tests with ScalaTest and SBT.

Keep the project you just created. We'll use it again in the next lesson.

44

Writing BDD-style tests with ScalaTest and SBT

In the previous lesson I showed how to write Test-Driven Development (TDD) tests with ScalaTest[1]. ScalaTest also supports a Behavior-Driven Development (BDD)[2] style of testing, which I'll demonstrate next.

> This lesson uses the same SBT project as the previous lesson, so you don't have to go through the initial setup work again

44.1 Create a Scala class to test

First, create a new Scala class to test. In *src/main/scala/simpletest*, create a new file named *MathUtils.scala* with these contents:

```
package simpletest

object MathUtils {

    def double(i: Int) = i * 2

}
```

The BDD tests you'll write next will test this `double` method.

[1] http://www.scalatest.org/
[2] https://dannorth.net/introducing-bdd/

44.2 Creating ScalaTest BDD-style tests

Next, create a file named *MathUtilsTests.scala* in the *src/test/scala/simpletest* directory, and put these contents in that file:

```
package simpletest

import org.scalatest.FunSpec

class MathUtilsSpec extends FunSpec {

    describe("MathUtils::double") {

        it("should handle 0 as input") {
            val result = MathUtils.double(0)
            assert(result == 0)
        }

        it("should handle 1") {
            val result = MathUtils.double(1)
            assert(result == 2)
        }

        it("should handle really large integers") (pending)

    }

}
```

As you can see, this style looks different than the TDD tests in the previous lesson. If you've never used a BDD style of testing before, a main idea is that the tests should be relatively easy to read for one of the "domain experts" who work with the programmers to create the application. A few notes about this code:

- This code uses the `FunSpec` class, whereas the TDD tests used `FunSuite`
- A set of tests begins with `describe`
- Each test begins with `it`. The idea is that the test should read like, "It should do XYZ…," where "it" is the `double` function
- In this example I also showed how to mark a test as "pending"

44.3 Running the tests

With those files in place you can again run `sbt test`. The important part of the output looks like this:

```
> sbt test

[info] HelloTests:
[info] - the name is set correctly in constructor
[info] - a Person's name can be changed
[info] MathUtilsSpec:
[info] MathUtils::double
[info] - should handle 0 as input
[info] - should handle 1
[info] - should handle really large integers (pending)
[info] Total number of tests run: 4
[info] Suites: completed 2, aborted 0
[info] Tests: succeeded 4, failed 0, canceled 0, ignored 0, pending 1
[info] All tests passed.
[success] Total time: 4 s, completed Jan 6, 2018 4:58:23 PM
```

A few notes about that output:

- `sbt test` ran the previous `HelloTests` as well as the new `MathUtilsSpec` tests
- The pending test shows up in the output and is marked "(pending)"
- All of the tests passed

If you want to have a little fun with this, change one or more of the tests so they intentionally fail, and then see what the output looks like.

44.4 See also

For more information about SBT and ScalaTest, see the following resources:

- In the Scala Cookbook[3] I cover both SBT and ScalaTest

[3] http://kbhr.co/hs-cook

- The main SBT documentation[4]
- The ScalaTest documentation[5]
- If you want to look into something different, *ScalaCheck* is a property-based testing tool, which I introduce here[6]

[4] http://www.scala-sbt.org/documentation.html
[5] http://www.scalatest.org/user_guide
[6] http://kbhr.co/hs-scalacheck

45

Functional Programming

Scala lets you write code in an object-oriented programming (OOP) style, a functional programming (FP) style, and even in a hybrid style, using both approaches in combination. I assume that you're coming to Scala from an OOP language like Java, C++, or C#, so outside of covering Scala classes and methods, I haven't written any special sections about OOP in this book. But because the FP style is still relatively new to many developers, I will provide a brief introduction to Scala's support for FP in the next several lessons.

Functional programming is a style of programming that emphasizes writing applications using only pure functions and immutable values. As I wrote in Functional Programming, Simplified[1], rather than using that description, it's more accurate to say that functional programmers have a strong desire to see their code as algebra — to see the combination of their functions as a series of algebraic equations. In that regard, you could say that functional programmers like to think of themselves as mathematicians. That's the driving desire that leads them to use *only* pure functions and immutable values, because that's what you use in algebra and other forms of math.

Functional Programming, Simplified is a large book, and there's no way I can condense all of that FP knowledge into this little book, but what I can do is give you a little taste of functional programming, and show some of the tools Scala provides for developers to write functional code.

[1] http://kbhr.co/hs-fps

46

Pure Functions

A first feature Scala offers to help you write functional code is the ability to write pure functions. In Functional Programming, Simplified[1] I define a *pure function* like this:

- The function's output depends *only* on its input variables
- It doesn't mutate any hidden state
- It doesn't have any "back doors": it doesn't read data from the outside world (including the console, web services, databases, files, etc.), or write data to the outside world

As a result of this definition, any time you call a pure function with the same input value(s), you'll always get the same result. For example, you can call a `double` function an infinite number of times with the input value `2` and you'll always get the result `4`.

46.1 Examples of pure functions

Given that definition of pure functions, as you might imagine, methods like these in the *scala.math.** package are pure functions:

- `abs`
- `ceil`
- `max`
- `min`

These Scala `String` methods are also pure functions:

- `isEmpty`

[1] http://kbhr.co/hs-fps

- `length`
- `substring`

Many methods on the Scala collections classes also work as pure functions, including `drop`, `filter`, and `map`.

46.2 Examples of impure functions

Conversely, the following functions are *impure* because they violate my definition.

The `foreach` method on collections classes is impure because it's only used for its side effects, such as printing to STDOUT.

> A great hint that `foreach` is impure is that it's method signature declares that it returns the type `Unit`. Because it returns nothing, logically the only reason you ever call it is to achieve some side effect.

Date and time related methods like `getDayOfWeek`, `getHour`, and `getMinute` are all impure because their output depends on something other than their input parameters. Their results rely on some form of hidden I/O; hidden input in these examples.

Methods like `println` and `readLine` are impure because they interact with the outside world. A method like `Random.nextInt()` is impure because it returns a different value each time it's called (and also mutates some sort of hidden state).

In general, impure functions do one or more of these things:

- Read hidden inputs, i.e., they access variables and data not explicitly passed into the function as input parameters
- Write hidden outputs
- Mutate the parameters they are given
- Perform some sort of I/O with the outside world

46.3 But impure functions are needed …

Of course an application isn't very useful if it can't read or write to the outside world, so in the Scala Cookbook[2] and Functional Programming, Simplified[3], I make this recommendation:

> Write the core of your application using pure functions, and then write an impure "wrapper" around that core to interact with the outside world. (If you like food analogies, this is like putting a layer of impure icing on top of a pure cake.)

In *Functional Programming, Simplified* I discuss ways of handling impure functions in depth, so please see that book for more details.

46.4 Writing pure functions

Writing pure functions in Scala is one of the simpler parts about functional programming: You just write them using Scala's method syntax. Here's a pure function that doubles the input value it's given:

```
def double(i: Int): Int = i * 2
```

Although I don't cover recursion in this book, if you like a good "challenge" example, here's a pure function that calculates the sum of a list of integers (`List[Int]`):

```
def sum(list: List[Int]): Int = list match {
    case Nil => 0
    case head :: tail => head + sum(tail)
}
```

Even though I haven't covered recursion, if you can understand that code, you'll see that it meets my definition of a pure function.

[2] http://kbhr.co/hs-cook
[3] http://kbhr.co/hs-fps

46.5 Key points

The first key point of this lesson is my definition of a pure function:

> A pure function is a function that depends only on its declared input parameters and its internal algorithm to produce its output. It does not read any other values from "the outside world" — the world outside of the function's scope — and it does not modify any values in the outside world.

A second key point is that real-world applications consist of a combination of pure and impure functions. My personal recommendation is to write the core of your application using pure functions, and then to use impure functions to communicate with the outside world.

47

Passing Functions Around

While every programming language ever created probably lets you write pure functions, a second great FP feature of Scala is that *you can create functions as variables*, just like you create `String` and `Int` variables. This feature has many benefits, the most common of which is that it lets you pass functions as parameters into other functions. You saw that earlier in this book when I demonstrated the `map` and `filter` methods:

```
val nums = (1 to 10).toList

val doubles = nums.map(_ * 2)
val lessThanFive = nums.filter(_ < 5)
```

In those examples I pass anonymous functions into `map` and `filter`.

In the lesson on anonymous functions I demonstrate that this example:

```
val doubles = nums.map(_ * 2)
```

is the same as passing a regular function into `map`:

```
def double(i: Int): Int = i * 2   //a method that doubles an Int
val doubles = nums.map(double)
```

As those examples show, Scala clearly lets you pass anonymous functions and regular functions into other methods. This is a powerful feature that good FP languages provide.

> If you like technical terms, a function that takes another function as an input parameter is known as a *Higher-Order Function* (HOF). (And if you like humor, as someone once wrote, that's like saying that a class that takes an instance of another class as a constructor parameter is a Higher-Order Class.)

47.1 Function or method?

Scala has a special "function" syntax[1], but as a practical matter very few people seem to use it. I think this is because of two reasons:

- That syntax can be hard to read
- You can use `def` methods just like they are functions

What I mean by that second statement is that when you define a method with `def` like this:

```
def double(i: Int): Int = i * 2
```

you can then pass `double` around as if it were a variable, like this:

```
val x = ints.map(double)
                ------
```

Even though I define `double` as a *method*, Scala lets you treat it as a *function*.

The ability to pass functions around as variables is a distinguishing feature of functional programming languages. And as you've seen in `map` and `filter` examples in this book, the ability to pass functions as parameters into other functions helps you create code that is concise and still readable.

47.2 A few examples

If you're not comfortable with the process of passing functions as parameters into other functions, here are a few more examples you can experiment with in the REPL:

```
List("foo", "bar").map(_.toUpperCase)
List("foo", "bar").map(_.capitalize)
List("adam", "scott").map(_.length)
List(1,2,3,4,5).map(_ * 10)
```

[1] http://kbhr.co/hs-fun

```
List(1,2,3,4,5).filter(_ > 2)
List(5,1,3,11,7).takeWhile(_ < 6)
```

Remember that any of those anonymous functions can also be written as "regular" functions, so you can write a function like this:

```
def toUpper(s: String): String = s.toUpperCase
```

and then pass it into `map` like this:

```
List("foo", "bar").map(toUpper)
```

or this:

```
List("foo", "bar").map(s => toUpper(s))
```

Those examples that use a "regular" function are equivalent to these anonymous function examples:

```
List("foo", "bar").map(s => s.toUpperCase)
List("foo", "bar").map(_.toUpperCase)
```

47.3 How to write functions that takes functions as parameters

In both the Scala Cookbook[2] and Functional Programming, Simplified[3] I demonstrate how to *write* methods like `map` and `filter` that take other functions as input parameters. I won't do that in this book, but when you get to the point where you want to write functions that take other functions as input parameters, it's a technique you'll want to learn.

[2] http://kbhr.co/hs-cook
[3] http://kbhr.co/hs-fps

47.4 See also

If you want to see what Scala's "function" syntax looks like, see my tutorial, The differences between `val` and `def` in Scala when creating functions[4].

[4] http://kbhr.co/hs-fun

48

No Null Values

Functional programming is like writing a series of algebraic equations, and because you don't use null values in algebra, you don't use null values in FP. That creates an interesting question: In the situations where you might normally use a null value in Java/OOP code, what do you do?

Scala's solution is to use constructs like the Option/Some/None classes. I'll provide an introduction to these techniques in this lesson.

48.1 A first example

While this first Option/Some/None example won't deal with null values, it's a good way to demonstrate the Option/Some/None classes, so I'll start with it.

Imagine that you want to write a method to make it easy to convert strings to integer values, and you want an elegant way to handle the exceptions that can be thrown when your method gets a string like `"foo"` instead of a string that converts to a number, like `"1"`. A first guess at such a function might look like this:

```
def toInt(s: String): Int = {
    try {
        Integer.parseInt(s.trim)
    } catch {
        case e: Exception => 0
    }
}
```

The idea of this function is that if a string converts to an integer, you return the converted `Int`, but if the conversion fails you return 0. This might be okay for some purposes, but it's not really accurate. For instance, the method might have received `"0"`, but it may have also received `"foo"` or `"bar"` or an infinite number of other strings. This creates a real problem: How do you know when the method really received a `"0"`,

or when it received something else? The answer is that with this approach, there's no way to know.

48.2 Using Option/Some/None

Scala's solution to this problem is to use a trio of classes known as `Option`, `Some`, and `None`. The `Some` and `None` classes are subclasses of `Option`, so the solution works like this:

- You declare that `toInt` returns an `Option` type
- If `toInt` receives a string it *can* convert to an `Int`, you wrap the `Int` inside of a `Some`
- If `toInt` receives a string it *can't* convert, it returns a `None`

The implementation of the solution looks like this:

```
def toInt(s: String): Option[Int] = {
    try {
        Some(Integer.parseInt(s.trim))
    } catch {
        case e: Exception => None
    }
}
```

This code can be read as, "When the given string converts to an integer, return the integer wrapped in a `Some` wrapper, such as `Some(1)`. When the string can't be converted to an integer, return a `None` value."

Here are two REPL examples that demonstrate `toInt` in action:

```
scala> val a = toInt("1")
a: Option[Int] = Some(1)

scala> val a = toInt("foo")
a: Option[Int] = None
```

As shown, the string `"1"` converts to `Some(1)` and the string `"foo"` converts to `None`. This is the essence of the Option/Some/None approach. It's used to handle exceptions

(as in this example), and the same technique works for handling null values.

> You'll find this approach used throughout Scala library classes, and in third-party Scala libraries.

48.3 Being a consumer of toInt

Now imagine that you're the consumer of the `toInt` method. You know that the method returns a subclass of `Option[Int]`, so the question becomes, how do you work with these return types?

There are two main answers, depending on your needs:

- Use a `match` expression
- Use a for-expression

> There are other approaches, but these are the two main approaches, especially from an FP standpoint.

48.3.1 Using a match expression

One possibility is to use a `match` expression, which looks like this:

```
toInt(x) match {
    case Some(i) => println(i)
    case None => println("That didn't work.")
}
```

In this example, if `x` can be converted to an `Int`, the first `case` statement is executed; if `x` can't be converted to an `Int`, the second `case` statement is executed.

48.3.2 Using for/yield

Another common solution is to use a for-expression — i.e., the for/yield combination I showed earlier in this book. To demonstrate this, imagine that you want to convert three strings to integer values, and then add them together. The for/yield solution looks like this:

```
val y = for {
    a <- toInt(stringA)
    b <- toInt(stringB)
    c <- toInt(stringC)
} yield a + b + c
```

When that expression finishes running, y will be one of two things:

- If all three strings convert to integers, y will be a `Some[Int]`, i.e., an integer wrapped inside a `Some`
- If any of the three strings can't be converted to an inside, y will be a `None`

You can test this for yourself in the Scala REPL. First, paste these three string variables into the REPL:

```
val stringA = "1"
val stringB = "2"
val stringC = "3"
```

Next, paste the for-expression into the REPL. When you do that, you'll see this result:

```
scala> val y = for {
     |     a <- toInt(stringA)
     |     b <- toInt(stringB)
     |     c <- toInt(stringC)
     | } yield a + b + c
y: Option[Int] = Some(6)
```

As shown, y is bound to the value `Some(6)`.

To see the failure case, change any of those strings to something that won't convert to an integer. When you do that, you'll see that y is a `None`:

```
y: Option[Int] = None
```

48.3.3 Showing for/yield's return type

Note that in the previous example you can explicitly show y's type:

```
val y: Option[Int] = for { ...
    --------------
```

This isn't required, but this is one situation where I often explicitly show a variable's data type because I think it makes the code easier to maintain.

48.4 Options can be thought of as a container of 0 or 1 items

A good way to think about the `Option` classes is that they represent a *container*, more specifically a container that has either zero or one item inside:

- `Some` is a container with one item in it
- `None` is a container, but it has nothing in it

If you prefer to think of the Option classes as being like a box, `None` is a little like getting an empty box for a birthday gift.

48.5 Using foreach

Because `Some` and `None` can be thought of containers, they can further be thought of as being like collections classes. As a result, they have all of the methods you'd expect from a collection class, including `map`, `filter`, `foreach`, etc.

This raises an interesting question: What will these two values print, if anything?

```
toInt("1").foreach(println)
toInt("x").foreach(println)
```

The answer is that the first example prints the number `1`, and the second example doesn't print anything. The first example prints `1` because:

- toInt("1") evaluates to `Some(1)`
- The expression evaluates to `Some(1).foreach(println)`

- The `foreach` method on the `Some` class knows how to reach inside the `Some` container and extract the value (`1`) that's inside it, so it passes that value to `println`

Similarly, the second example prints nothing because:

- `toInt("x")` evaluates to `None`
- The expression evaluates to `None.foreach(println)`
- The `foreach` method on the `None` class knows that `None` doesn't contain anything, so it does nothing

Somewhere in Scala's history, someone noted that the first example (the `Some`) represents the "Happy Path" of Option/Some/None approach, and the second example (the `None`) represents the "Unhappy Path." *But*, despite having two different possible outcomes, the cool thing about the approach is that the code you write to handle an `Option` looks exactly the same in both cases. The `foreach` examples look like this:

```
toInt("1").foreach(println)
toInt("x").foreach(println)
```

And the for-expression looks like this:

```
val y = for {
    a <- toInt(stringA)
    b <- toInt(stringB)
    c <- toInt(stringC)
} yield a + b + c
```

You only have to write one piece of code to handle both the Happy and Unhappy Paths, and that simplifies your code. The only time you have to think about whether you got a `Some` or a `None` is when you finally handle the result, such as in a `match` expression:

```
toInt(x) match {
    case Some(i) => println(i)
    case None => println("That didn't work.")
}
```

48.6 Using Option to replace null values

Another place where a null value can silently creep into your code is with a class like this:

```
class Address (
    var street1: String,
    var street2: String,
    var city: String,
    var state: String,
    var zip: String
)
```

While every address on Earth has a `street1` value, the `street2` value is optional. As a result, that class is subject to this type of abuse:

```
val santa = new Address(
    "1 Main Street",
    null,           // <-- D'oh! A null value!
    "North Pole",
    "Alaska",
    "99705"
)
```

To handle situations like this, developers tend to use null values or empty strings, both of which are hacks to work around the main problem: `street2` is an *optional* field. In Scala — and other modern languages — the correct solution is to declare up front that `street2` is optional:

```
class Address (
    var street1: String,
    var street2: Option[String],
    var city: String,
    var state: String,
    var zip: String
)
```

With that definition, developers can write more accurate code like this:

```
val santa = new Address(
    "1 Main Street",
    None,
    "North Pole",
    "Alaska",
    "99705"
)
```

or this:

```
val santa = new Address(
    "123 Main Street",
    Some("Apt. 2B"),
    "Talkeetna",
    "Alaska",
    "99676"
)
```

Once you have an optional field like this, you work with it as I showed in the previous examples: With `match` expressions, `for` expressions, and other built-in methods like `foreach`.

48.7 Option isn't the only solution

In this lesson I focused on the Option/Some/None solution, but Scala has a few other alternatives. For example, a trio of classes known as Try/Success/Failure work in the same manner, but a) you primarily use these classes when code can throw exceptions, and b) the `Failure` class gives you access to the exception message. I commonly use Try/Success/Failure when writing methods that interact with files, databases, and internet services, as those functions can easily throw exceptions. I demonstrate these classes in the Functional Error Handling lesson that follows.

48.8 Key points

This lesson was a little longer than the others, so here's a quick review of the key points:

- Functional programmers don't use null values
- A common replacement for null values is to use the Option/Some/None classes
- Common ways to work with Option values are `match` and `for` expressions
- Options can be thought of as containers of one item (`Some`) and no items (`None`)
- You can also use Options when defining constructor parameters

48.9 See also

- Tony Hoare invented the null reference in 1965, and refers to it as his "billion dollar mistake[1]."
- For *much* more information on Option/Some/None and Try/Success/Failure, see the Scala Cookbook[2], and Functional Programming, Simplified[3]

[1] https://en.wikipedia.org/wiki/Tony_Hoare#Apologies_and_retractions
[2] http://kbhr.co/hs-cook
[3] http://kbhr.co/hs-fps

49
Companion Objects

A *companion object* in Scala is an `object` that's declared in the same file as a `class`, and has the same name as the class. For instance, when the following code is saved in a file named *Pizza.scala*, the `Pizza` object is considered to be a companion object to the `Pizza` class:

```
class Pizza {
}

object Pizza {
}
```

As you'll see, this has several benefits. First, a companion object and its class can access each other's private members (fields and methods). This means that the `printFilename` method in this class will work because it can access the private `HIDDEN_FILENAME` field in its companion object:

```
class SomeClass {
    def printFilename(): Unit = println(SomeClass.HIDDEN_FILENAME)
}

object SomeClass {
    private val HIDDEN_FILENAME = "/tmp/foo.bar"
}
```

A companion object offers much more functionality than this, and I'll explain a few of its most important features in the rest of this lesson.

49.1 Creating new instances without the `new` keyword

You probably noticed in some examples in this book that you can create new instances of certain classes without having to use the `new` keyword before the class name, as in

this example:

```
val zenMasters = List(
    Person("Nansen"),
    Person("Joshu")
)
```

This functionality comes from the use of companion objects. What happens is that when you define an `apply` method in a companion object, it has a special meaning to the Scala compiler. There's a little syntactic sugar baked into Scala that lets you type this code:

```
val p = Person("Frank")
```

and during the compilation process the compiler turns that code into this code:

```
val p = Person.apply("Frank")
```

The `apply` method in the companion object acts as a factory method[1], and Scala's syntactic sugar lets you use the syntax shown, creating new class instances without using the `new` keyword.

49.1.1 Enabling that functionality

To demonstrate how this works, here's a class named `Person` along with an `apply` method in its companion object:

(see the next page)

[1] https://alvinalexander.com/java/java-factory-pattern-example

```
class Person {
    var name = ""
}

object Person {
    def apply(name: String): Person = {
        var p = new Person
        p.name = name
        p
    }
}
```

To test this code, paste both the class and the object into the Scala REPL at the same time using this technique:

- Start the REPL from your command line (with the `scala` command)
- Type `:paste` and press the [Enter] key
- The REPL should respond with this text:

```
// Entering paste mode (ctrl-D to finish)
```

- Now paste both the class and object into the REPL
- Press Ctrl-D to finish the "paste" process

When that process works you should see this output in the REPL:

```
defined class Person
defined object Person
```

> I use this `:paste` technique because the REPL requires that a class and its companion object be entered at the same time.

Now you can create a new instance of the `Person` class like this:

```
val p = Person.apply("Frank")
```

That code directly calls `apply` in the companion object. More importantly, you can also create a new instance like this:

```
val p = Person("Frank")
```

and multiple instances like this:

```
val zenMasters = List(
    Person("Nansen"),
    Person("Joshu")
)
```

To be clear, what happens in this process is:

- You type something like `val p = Person("Frank")`
- The Scala compiler sees that there is no `new` keyword before `Person`
- The compiler looks for an `apply` method in the companion object of the `Person` class that matches the type signature you entered
- If it finds an `apply` method, it uses it; if it doesn't, you get a compiler error

49.1.2 Creating multiple constructors

You can create multiple `apply` methods in a companion object to provide multiple constructors. This code shows how to create both one- and two-argument constructors:

```
class Person {
    var name = ""
    var age = 0
}

object Person {

    // a one-arg constructor
    def apply(name: String): Person = {
        var p = new Person
        p.name = name
        p
    }
```

```
    // a two-arg constructor
    def apply(name: String, age: Int): Person = {
        var p = new Person
        p.name = name
        p.age = age
        p
    }

}
```

If you paste that code into the REPL as before, you'll see that you can create new `Person` instances like this:

```
val fred = Person("Fred")
val john = Person("John", 42)
```

> When running tests like this, it's best to clear the REPL's memory. To do this, use the `:reset` command inside the REPL before using the `:paste` command.

49.2 Adding an `unapply` method

Just as adding an `apply` method in a companion object lets you *construct* new object instances, adding an `unapply` lets you *de-construct* object instances. I'll demonstrate this with an example.

Here's a different version of a `Person` class and a companion object:

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

object Person {
    def unapply(p: Person): String = s"${p.name}, ${p.age}"
}
```

Notice that the companion object defines an `unapply` method. That method takes an input parameter of the type `Person`, and returns a `String`. To test the `unapply` method manually, first create a new `Person` instance:

```
val p = new Person("Lori", 29)
```

Then test `unapply` like this:

```
val result = Person.unapply(p)
```

This is what the `unapply` result looks like in the REPL:

```
scala> val result = Person.unapply(p)
result: String = Lori, 29
```

As shown, `unapply` de-constructs the `Person` instance it's given. In Scala, when you put an `unapply` method in a companion object, it's said that you've created an *extractor* method, because you've created a way to extract the fields out of the object.

49.2.1 `unapply` can return different types

In that example `unapply` returns a `String`, but you can write it to return anything. Here's an example that returns the two fields in a tuple:

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

object Person {
    def unapply(p: Person): Tuple2[String, Int] = (p.name, p.age)
}
```

Here's what that method looks like in the REPL:

```
scala> val result = Person.unapply(p)
result: (String, Int) = (Lori,29)
```

Because this `unapply` method returns the class fields as a tuple, you can also do this:

```
scala> val (name, age) = Person.unapply(p)
name: String = Lori
age: Int = 29
```

49.2.2 `unapply` extractors in the real world

A benefit of using `unapply` to create an extractor is that if you follow the proper Scala conventions, it enables a convenient form of pattern-matching in match expressions.

I'll discuss that more in the next lesson, but as you'll see, the story gets even better: You rarely need to write an `unapply` method yourself. Instead, what happens is that you get `apply` and `unapply` methods for free when you create your classes as *case classes* rather than as the "regular" Scala classes you've seen so far. We'll dive into case classes in the next lesson.

49.3 Key points

As a brief summary, the key points of this lesson are:

- A *companion object* is an `object` that's declared in the same file as a `class`, and has the same name as the class
- A companion object and its class can access each other's private members
- A companion object's `apply` method lets you create new instances of a class without using the `new` keyword
- A companion object's `unapply` method lets you de-construct an instance of a class into its individual components

50
Case Classes

Another Scala feature that provides support for functional programming is the *case class*. A case class has all of the functionality of a regular class, and more. When the compiler sees the `case` keyword in front of a `class`, it generates code for you, with the following benefits:

- Case class constructor parameters are public `val` fields by default, so accessor methods are generated for each parameter.
- An `apply` method is created in the companion object of the class, so you don't need to use the `new` keyword to create a new instance of the class.
- An `unapply` method is generated, which lets you easily use case classes in `match` expressions and other situations.
- A `copy` method is generated in the class. I never use this in Scala/OOP code, but I use it all the time in Scala/FP.
- `equals` and `hashCode` methods are generated, which let you compare objects and easily use them as keys in maps.
- A default `toString` method is generated, which is helpful for debugging.

I'll demonstrate how all of these features work in the following sections.

50.1 An `apply` method means you don't need `new`

When you define a class as a `case` class, you don't have to use the `new` keyword to create a new instance:

```
scala> case class Person(name: String, relation: String)
defined class Person

// "new" not needed before Person
scala> val christina = Person("Christina", "niece")
christina: Person = Person(Christina,niece)
```

As discussed in the previous lesson, this works because a method named `apply` is generated inside `Person`'s companion object.

50.2 No mutator methods

Case class constructor parameters are `val` fields by default, so an *accessor* method is generated for each parameter:

```
scala> christina.name
res0: String = Christina
```

But, mutator methods are not generated:

```
// can't mutate the `name` field
scala> christina.name = "Fred"
<console>:10: error: reassignment to val
       christina.name = "Fred"
                 ^
```

Because in FP you never mutate data structures, it makes sense that constructor fields default to `val`.

50.3 An `unapply` method

Case classes automatically generate an `unapply` method, so you don't have to write one. To demonstrate this, imagine that you have this trait:

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

Then, create these case classes to extend that trait:

```
case class Student(name: String, year: Int) extends Person
case class Teacher(name: String, specialty: String) extends Person
```

Because those are defined as case classes — and a case class has a built-in `unapply` method — you can write a method that takes a `Person` input parameter and matches

on the `Student` and `Teacher` types:

```
def getPrintableString(p: Person): String = p match {
    case Student(name, year) =>
        s"$name is a student in Year $year."
    case Teacher(name, whatTheyTeach) =>
        s"$name teaches $whatTheyTeach."
}
```

Notice these two patterns in the `case` statements:

```
case Student(name, year) =>
case Teacher(name, whatTheyTeach) =>
```

Those patterns work because `Student` and `Teacher` have `unapply` methods whose type signature conforms to a certain standard. Technically, the specific type of pattern matching shown in these examples is known as a *constructor pattern*.

> The Scala standard is that an `unapply` method returns the case class constructor fields in a tuple that's wrapped in an `Option`. I showed the "tuple" part of the solution in the previous lesson.

To show how that code works, create an instance of `Student` and `Teacher`:

```
val s = Student("Al", 1)
val t = Teacher("Bob Donnan", "Mathematics")
```

Next, this is what the output looks like in the REPL when you call `getPrintableString` with those two instances:

```
scala> getPrintableString(s)
res0: String = Al is a student in Year 1.

scala> getPrintableString(t)
res1: String = Bob Donnan teaches Mathematics.
```

> All of this content on `unapply` methods and extractors is a little advanced for an introductory book like this, but because case classes are an important FP topic, I thought it was better to cover them, rather than skipping over them.

50.4 copy method

A case class also has an automatically-generated copy method that's extremely helpful when you need to perform the process of a) cloning an object and b) updating one or more of the fields during the cloning process. As an example, this is what the process looks like in the REPL:

```
scala> case class BaseballTeam(name: String, lastWorldSeriesWin: Int)
defined class BaseballTeam

scala> val cubs1908 = BaseballTeam("Chicago Cubs", 1908)
cubs1908: BaseballTeam = BaseballTeam(Chicago Cubs,1908)

scala> val cubs2016 = cubs1908.copy(lastWorldSeriesWin = 2016)
cubs2016: BaseballTeam = BaseballTeam(Chicago Cubs,2016)
```

As shown, when you use the copy method, all you have to do is supply the names of the fields you want to modify during the cloning process.

Because you never mutate data structures in FP, this is how you create a new instance of a class from an existing instance. I refer to this process as "update as you copy," and I discuss this process in detail in my book, Functional Programming, Simplified[1].

50.5 equals and hashCode methods

Case classes also have automatically-generated equals and hashCode methods, so instances can be compared:

```
scala> case class Person(name: String, relation: String)
defined class Person

scala> val christina = Person("Christina", "niece")
christina: Person = Person(Christina,niece)
```

[1] http://kbhr.co/hs-fps

```
scala> val hannah = Person("Hannah", "niece")
hannah: Person = Person(Hannah,niece)

scala> christina == hannah
res1: Boolean = false
```

These methods also let you easily use your objects in collections like sets and maps.

50.6 toString methods

Finally, case classes also have a good default toString method implementation, which at the very least is helpful when debugging code:

```
scala> christina
res0: Person = Person(Christina,niece)
```

50.7 The biggest advantage

While all of these features are great benefits to functional programming, as they write in the book, Programming in Scala[2], "the biggest advantage of case classes is that they support pattern matching." Pattern matching is a major feature of FP languages, and Scala's case classes provide a simple way to implement pattern matching in match expressions and other areas.

[2] http://amzn.to/2BW22sN

51
Case Objects

Before we jump into *case objects*, I should provide a little background on regular Scala objects. As I mentioned early on in this book, you use a Scala `object` when you want to create a singleton object. As the official Scala documentation states[1], "Methods and values that aren't associated with individual instances of a class belong in singleton objects, denoted by using the keyword `object` instead of `class`."

A common example of this is when I create a "utilities" object, such as this one:

```
object PizzaUtils {
    def addTopping(p: Pizza, t: Topping): Pizza = ...
    def removeTopping(p: Pizza, t: Topping): Pizza = ...
    def removeAllToppings(p: Pizza): Pizza = ...
}
```

Or this one:

```
object FileUtils {
    def readTextFileAsString(filename: String): Try[String] = ...
    def copyFile(srcFile: File, destFile: File): Try[Boolean] = ...
    def readFileToByteArray(file: File): Try[Array[Byte]] = ...
    def readFileToString(file: File): Try[String] = ...
    def readFileToString(file: File, encoding: String): Try[String] = ...
    def readLines(file: File, encoding: String): Try[List[String]] = ...
}
```

This is the most common way I use the Scala `object` construct.

[1] https://docs.scala-lang.org/tour/singleton-objects.html

51.1 Case objects

A `case object` is like an `object`, but just like a case class has more features than a regular class, a case object has more features than a regular object. It specifically has two important features that make it useful:

- It is serializable
- It has a default `hashCode` implementation

These features make a case object useful when you don't know how it will be used by other developers, such as if it will be sent across a network and referenced in a different JVM (such as with the Akka[2] actors platform, where you can send messages between JVM instances).

Because of these features, I primarily use case objects (instead of regular objects) in two places:

- When creating enumerations
- When creating containers for "messages" that I want to pass between other objects (such as with Akka)

51.2 Creating enumerations with case objects

As I showed earlier in this book, you create enumerations in Scala like this:

```
sealed trait Topping
case object Cheese extends Topping
case object Pepperoni extends Topping
case object Sausage extends Topping
case object Mushrooms extends Topping
case object Onions extends Topping
```

[2] https://akka.io/

```
sealed trait CrustSize
case object SmallCrustSize extends CrustSize
case object MediumCrustSize extends CrustSize
case object LargeCrustSize extends CrustSize

sealed trait CrustType
case object RegularCrustType extends CrustType
case object ThinCrustType extends CrustType
case object ThickCrustType extends CrustType
```

Then later in your code you use those enumerations like this:

```
case class Pizza (
    crustSize: CrustSize,
    crustType: CrustType,
    toppings: Seq[Topping]
)
```

51.3 Using case objects as messages

Another place where case objects come in handy is when you want to model the concept of a "message." For example, imagine that you're writing an application like Amazon's Alexa, and you want to be able to pass around "speak" messages like, "speak the enclosed text," "stop speaking," "pause," and "resume." In Scala you create singleton objects for those messages like this:

```
case class StartSpeakingMessage(textToSpeak: String)
case object StopSpeakingMessage
case object PauseSpeakingMessage
case object ResumeSpeakingMessage
```

Notice that `StartSpeakingMessage` is defined as a case *class* rather than a case *object*. This is because an object can't have any constructor parameters.

Given those messages, if Alexa was written using the Akka library, you'd find code like this in a "speak" class:

```
class Speak extends Actor {
  def receive = {
    case StartSpeakingMessage(textToSpeak) =>
        // code to speak the text
    case StopSpeakingMessage =>
        // code to stop speaking
    case PauseSpeakingMessage =>
        // code to pause speaking
    case ResumeSpeakingMessage =>
        // code to resume speaking
  }
}
```

This is a good, safe way to pass messages around in a Scala application.

51.4 See also

I introduce the Akka actor library later in this book, and I've also written several tutorials that demonstrate how it uses case objects for messages:

- An Akka "Hello, world" example[3]
- A 'Ping Pong' Scala Akka actors example[4]
- How to send and receive messages between Scala/Akka actors[5]
- "Alexa written with Akka" = Aleka[6]
- An Akka actors 'remote' example[7]

[3] http://kbhr.co/hs-akka1
[4] http://kbhr.co/hs-akka2
[5] http://kbhr.co/hs-akka3
[6] http://kbhr.co/hs-akka4
[7] http://kbhr.co/hs-akka5

52

Functional Error Handling

Because functional programming is like algebra, there are no null values or exceptions. But of course you can still have exceptions when you try to access servers that are down or files that are missing, so what can you do?

52.1 Option/Some/None

I already demonstrated one of the techniques to handle errors in Scala: The trio of classes named `Option`, `Some`, and `None`. Instead of writing a method like `toInt` to throw an exception or return a null value, you declare that the method returns an `Option`, in this case an `Option[Int]`:

```
def toInt(s: String): Option[Int] = {
    try {
        Some(Integer.parseInt(s.trim))
    } catch {
        case e: Exception => None
    }
}
```

Later in your code you handle the result from `toInt` using `match` and `for` expressions:

```
toInt(x) match {
    case Some(i) => println(i)
    case None => println("That didn't work.")
}

val y = for {
    a <- toInt(stringA)
    b <- toInt(stringB)
    c <- toInt(stringC)
} yield a + b + c
```

I discussed these approaches in the "No Null Values" lesson, so I won't repeat that discussion here.

52.2 Try/Success/Failure

Another trio of classes named `Try`, `Success`, and `Failure` work just like `Option`, `Some`, and `None`, but with two nice features:

- `Try` makes it very simple to catch exceptions
- `Failure` contains the exception message

Here's the `toInt` method re-written to use these classes. First, you have to import the classes into the current scope:

```
import scala.util.{Try,Success,Failure}
```

After that, this is what `toInt` looks like:

```
def toInt(s: String): Try[Int] = Try {
    Integer.parseInt(s.trim)
}
```

As you can see, that's quite a bit shorter than the Option/Some/None approach. The REPL demonstrates how this works. First, the success case:

```
scala> val a = toInt("1")
a: scala.util.Try[Int] = Success(1)
```

Second, this is what it looks like when `Integer.parseInt` throws an exception:

```
scala> val b = toInt("boo")
b: scala.util.Try[Int] =
   Failure(java.lang.NumberFormatException: For input string: "boo")
```

As that output shows, the `Failure` that's returned by `toInt` contains the reason for the failure, i.e., the exception message.

There are quite a few ways to work with the results of a `Try` — including the ability to "recover" from the failure — but common approaches still involve using `match` and `for`

expressions:

```
toInt(x) match {
    case Success(i) => println(i)
    case Failure(s) => println(s"Failed. Reason: $s")
}

val y = for {
    a <- toInt(stringA)
    b <- toInt(stringB)
    c <- toInt(stringC)
} yield a + b + c
```

Note that if the three string values all convert to `Int` values, the `for` expression returns the `Int` wrapped in a `Success`:

```
scala.util.Try[Int] = Success(6)
```

Conversely, if any of the strings won't convert to an `Int`, the `for` expression returns a `Failure` that contains the exception information:

```
scala.util.Try[Int] =
  Failure(java.lang.NumberFormatException: For input string: "a")
```

52.3 Even more …

There are other combinations of classes from third party libraries that you can also use to catch and handle exceptions, but Option/Some/None and Try/Success/Failure are my two favorites. You can use whatever you like, but I generally use Try/Success/Failure when dealing with code that can throw exceptions — because I almost always want the exception message — and I use Option/Some/None in other places, such as to avoid using null values.

53

Concurrency

In the next several lessons I'll demonstrate two of Scala's main techniques for writing parallel and concurrent applications: Akka actors and the Scala `Future`.

54

Akka Actors

In this lesson I provide a brief introduction to the Akka actors library[1]. In the lesson you'll learn about:

- Actors and the actor model
- Akka's benefits

At the end I also share a link to a video of an Alexa-like application I wrote using actors

> If you're already comfortable with the *Actor model*, feel free to move on to the next lesson, where I share some Scala/Akka code.

54.1 Actors and the Actor Model

The first thing to know about actors is the *actor model*, which is a mental model of how to think about a system built with actors. Within that model the first concept to understand is an *actor*:

- An actor is a long-running process that runs in parallel to the main application thread, and responds to messages that are sent to it.
- An actor is the smallest unit of functionality when building an actor-based system, just like a class is the smallest unit in an OOP system.
- Like a class, an actor encapsulates state and behavior.
- You can't peek inside an actor to get its state. You can send an actor a message requesting state information (like texting a person to ask how they're feeling), but you can't reach in and execute one of its methods or access its fields (just like you can't peak inside someone else's brain).

[1] http://akka.io/

- An actor has a mailbox (an inbox), and the actor's purpose in life is to process the messages in its mailbox.
- You communicate with an actor by sending it an immutable message (typically as a case class or case object in Akka). These messages go directly into the actor's mailbox.
- When an actor receives a message, it's like taking a letter out of its mailbox. It opens the letter, processes the message using one of its algorithms, then moves on to the next message in the mailbox. If there are no more messages, the actor waits until it receives one.

Akka experts recommend thinking of an actor as being like a person, such as a person in a business organization:

- You can't know what's going on inside another person. All you can do is send them a message and wait for their response.
- An actor has one parent, known as a *supervisor*. In Akka, that supervisor is the actor that created it.
- An actor may have children. For instance, a President in a business may have a number of Vice Presidents. Those VPs are like children of the President, and they may also have many subordinates. (And those subordinates may have many subordinates, etc.)
- An actor may have siblings — i.e., other actors at the same level. For instance, there may be 10 VPs in an organization, and they're all at the same level in the organization chart.

54.1.1 Actors should delegate their work

There's one more important point to know about actors: As soon as an actor receives a message, it should delegate its work. Actors need to be able to respond to messages in their mailbox as fast as possible, so the actor mantra is, "Delegate, delegate, delegate."

If you think of an actor as being a person, imagine that one message includes a task that's going to take a month to complete. If the actor worked on that one task for a month, it wouldn't be able to respond to its mailbox for a month. That's bad. But if the actor delegates that task to one of its children, it can respond to the next message in its mailbox immediately (and delegate that as well).

54.2 Akka features

Akka is the main actor library for Scala, and it's a great way to build massively parallel systems. From my own experience I can say that all of these industry buzzwords can be used to describe Akka:

- asynchronous
- event-driven
- message-driven
- reactive
- scalable ("scale up" and "scale out")
- concurrent and parallel
- non-blocking
- location transparency
- resilient and redundant (no single point of failure with multiple, distributed servers)
- fault-tolerant

All of those features are great, but the first great feature is that Akka and the actor model *greatly* simplify the process of working with long-running threads. In fact, when working with Akka, you never really think about threads, you just write actors to respond to messages in a non-blocking manner, and the threads take of themselves.

54.3 Akka benefits

In addition to those features, here are some concrete benefits of using Akka actors, mostly coming from Lightbend's Akka Quickstart Guide[2] and the Akka.io website[3]:

- Actors are *much* easier to work with than threads; you program at a much higher level of abstraction.
- Actors let you build systems that *scale up*, using the resources of a server more efficiently, and *scale out*, using multiple servers.

[2] http://developer.lightbend.com/guides/akka-quickstart-scala/
[3] http://akka.io

- Performance: Actors have been shown to process up to 50 million messages/second on a single machine.
- Lightweight: Each instance consumes only a few hundred bytes, which allows millions of concurrent actors to exist in a single application (allowing ~2.5 million actors per GB of heap).
- Location transparency: The Akka system constructs actors from a factory and returns references to the instances. Because the location of actors doesn't matter — they can be running on the current server or some other server — actor instances can start, stop, move, and restart to scale up and down, as well as recover from unexpected failures.

54.4 A video example

Way back in 2011 I started developing a "personal assistant" named SARAH, which was based on the computer assistant of the same name on the television show Eureka[4]. SARAH is like having the Amazon Echo[5] running on your computer. You speak to it to access and manage information:

- Get news headlines from different sources
- Get weather reports and stock prices
- Manage a "to-do list"
- Control iTunes with voice commands
- Check your email
- Perform Google searches

Beyond just *responding* to voice commands with spoken and displayed output, SARAH also has long-running background tasks — small pieces of software I call "agents" — so it can do other things:

- Tell me when I receive new email from people I'm interested in
- Report the time at the top of every hour ("The time is 11 a.m.")

[4] http://www.imdb.com/title/tt0796264/
[5] http://amzn.to/2y4bgoJ

The entire application is based on Akka actors. I found Akka to be a terrific way to write an application that had many threads running simultaneously.

For more information on SARAH, see the "Sarah - Version 2" video at alvinalexander.com/sarah[6]. I haven't worked on SARAH in a while, but it gives you can idea of what can be done with Akka actors.

> For a simpler version of SARAH that you can get started with today, see my tutorial, "Alexa written with Akka" = Aleka[7]

54.5 What's next

Given this background, the next lesson shows several examples of how to use Akka actors.

> From the Akka FAQ[8], the name *Akka* "is the name of a beautiful Swedish mountain in the northern part of Sweden called Laponia … Akka is also the name of a goddess in the Sámi (the native Swedish population) mythology. She is the goddess that stands for all the beauty and good in the world … Also, the name AKKA is a palindrome of the letters A and K, as in Actor Kernel."

[6] https://alvinalexander.com/sarah
[7] http://kbhr.co/hs-akka4
[8] https://doc.akka.io/docs/akka/2.5/additional/faq.html

55

Akka Actor Examples

In this lesson I'll show two examples of applications that use Akka actors, both of which can help you get started with my larger "Alexa written with Akka" = Aleka[1] application.

55.1 Source code

I originally wrote this lesson for my book, Functional Programming, Simplified[2], so you can find the source code for it at this URL:

- github.com/alvinj/FPAkkaHelloWorld[3]

55.2 An Akka "Hello, world" example

First, let's look at an example of how to write a "Hello, world" application using Akka.

55.2.1 Writing a "Hello" actor

An actor is an instance of the `akka.actor.Actor` class, and once it's created it starts running on a parallel thread, and all it does is respond to messages that are sent to it. For this "Hello, world" example I want an actor that responds to "hello" messages, so I start with code like this:

[1] http://kbhr.co/hs-akka4
[2] http://kbhr.co/hs-fps
[3] https://github.com/alvinj/FPAkkaHelloWorld

```
case class Hello(msg: String)

class HelloActor extends Actor {
    def receive = {
        case Hello(s) => {
            println(s"you said '$s'")
            println(s"$s back at you!\n")
        }
        case _ => println("huh?")
    }
}
```

In the first line of code I define a case class named `Hello`. The preferred way to send messages with Akka is to use instances of case classes and case objects because they support immutability and pattern-matching. Therefore, I define `Hello` as a simple wrapper around a string.

After that, I define `HelloActor` as an instance of `Actor`. The body of `HelloActor` is just the `receive` method, which you implement to define the actor's behavior, i.e., how the actor responds to the messages it receives.

The way this code works is that when `HelloActor` receives a new message in its inbox, `receive` is triggered as a response to that event, and the incoming message is tested against `receive`'s `case` statements. In this example, if the message is of the type `Hello`, the first `case` statement handles the message; if the message is *anything else*, the second `case` statement is triggered. (The second `case` statement is a "catch-all" statement that handles all unknown messages.)

Of course actors get more complicated than this, but that's the essence of the actor programming pattern:

- You create case classes and case objects to define the types of messages you want your actor to receive
- Because the only way the rest of your code can interact with the actor is by sending messages to it, those classes and objects become your actor's API
- Inside the actor's `receive` method you define how you want to respond to each message type

At a high level, that's all there is to writing actor code.

55.2.2 A test program

Now all you need is a little driver program to test the actor. This one will do:

```
object AkkaHelloWorld extends App {

    // an actor needs an ActorSystem
    val system = ActorSystem("HelloSystem")

    // create and start the actor
    val helloActor = system.actorOf(
        Props[HelloActor],
        name = "helloActor"
    )

    // send the actor two known messages
    helloActor ! Hello("hello")
    helloActor ! Hello("buenos dias")

    // send it an unknown message
    helloActor ! "hi!"

    // shut down the system
    system.terminate()

}
```

Here's how that code works. First, actors need an ActorSystem[4] that they can run in, so you create one like this:

```
val system = ActorSystem("HelloSystem")
```

Just give the `ActorSystem` a unique name, and you're ready to go.

> The `ActorSystem` is the main construct that takes care of the gory thread details behind the scenes. Per the Akka website, "An `ActorSystem` is a

[4] https://doc.akka.io/api/akka/current/akka/actor/ActorSystem.html

> heavyweight structure that will allocate 1…N Threads, so create one per logical application … It is also the entry point for creating or looking up actors."

Next, as that quote states, you create new actors with the `ActorSystem`, so this is how you create an instance of a `HelloActor`:

```
val helloActor = system.actorOf(
    Props[HelloActor],
    name = "helloActor"
)
```

Depending on your needs there are a few variations of that method, but the important part is that you create an instance of `HelloActor` by calling `actorOf` on the `ActorSystem`.

Besides the required `import` statements, that's the entire setup process. At this point the `helloActor` instance is up and running in parallel with the main application thread, and you can send it messages. This is how you send it a message:

```
helloActor ! Hello("hello")
```

This line of code can be read as, "Send the message `Hello("hello")` to the actor named `helloActor`, and don't wait for a reply."

The `!` character is how you send a message to an actor. More precisely, it's how you send a message to an actor *without waiting for a reply back from the actor*. This is by far the most common way to send a message to an actor; in almost every situation you don't want to wait for a reply back from the actor, because that would cause your application's thread to block at that point, and *blocking* is bad.

This `case` statement inside `HelloActor` handles `Hello` messages that are received:

```
// in HelloActor
case Hello(s) => {
    println(s"you said '$s'")
    println(s"$s back at you!\n")
}
```

Inside that `case` statement I just print two lines of output, but in real world applications this is where you normally call other functions or delegate work to a child actor.

After I send the two `Hello` messages to the `HelloActor`, I send it this message:

```
helloActor ! "hi!"
```

Because `HelloActor` doesn't know how to handle a `String` message, it responds to this message with its "catch-all" `case` statement:

```
// in HelloActor
case _ => println("huh?")
```

At this point the `AkkaHelloWorld` application reaches this line of code, which shuts down the `ActorSystem`:

```
system.terminate()
```

That's the entire Akka "Hello, world" application.

I encourage you to work with the source code for this lesson. In the *HelloWorld.scala* file, add new messages (as case classes and case objects), and then add new `case` statements to the `receive` method in `HelloActor` to respond to those messages, and add methods inside `HelloActor` to process those messages. Keep playing with it until you're sure you know how it all works.

55.3 A second example

As a slightly more complicated example, the *Echo.scala* file in the same repository contains an Akka application that responds to whatever you type at the command line. First, the application defines a case class and a case object that are used to send and receive messages:

```
case class Message(msg: String)
case object Bye
```

Next, this is how the `EchoActor` responds to the messages it receives:

```
class EchoActor extends Actor {
    def receive = {
        case Message(s) => println("\nyou said " + s)
        case Bye => println("see ya!")
        case _ => println("huh?")
    }
}
```

That follows the same pattern I showed in the first example.

Finally, here's a driver program you can use to test `EchoActor`:

```
object EchoMain extends App {

    // an actor needs an ActorSystem
    val system = ActorSystem("EchoSystem")

    // create and start the actor
    val echoActor = system.actorOf(
        Props[EchoActor],
        name = "echoActor"
    )

    // prompt the user for input
    var input = ""
    while (input != "q") {
        print("type something (q to quit): ")
        input = StdIn.readLine()
        echoActor ! Message(input)
    }

    echoActor ! Bye

    // shut down the system
    system.terminate()

}
```

Notice that after the `ActorSystem` and `echoActor` are created, the application sits in a loop prompting you for input, until you enter the character `q`. Once you type `q` and the loop terminates, the `echoActor` is sent one last message:

```
echoActor ! Bye
```

After that, the system shuts down.

This is what the output of the application looks like when you run it and type a few things at the command line:

```
type something (q to quit): hello
you said hello

type something (q to quit): hola
you said hola

type something (q to quit): q
you said q

bye!
```

55.4 More examples

I could keep showing more examples, but the pattern is the same:

- Create case classes and case objects for the messages you want your actor to handle.
- Write your actor's `receive` method so it responds to those messages as desired.
- Send messages to your actors using `!`.

If you'd like to work with a more-complicated example that builds on this second example, I created an Akka application that works a little like SARAH and the Amazon Echo[5], albeit at your computer's command line. See this page on my website for more details:

[5] http://amzn.to/2xwmlgM

- alvinalexander.com/amazon-echo-akka[6]

That web page describes how the "Akkazon Ekko" application works, but here's a quick example of some command-line input and output with it, where `ekko` is the application's command line prompt:

```
ekko: weather
stand by ...
The current temperature is 78 degrees, and the sky is partly cloudy.

ekko: forecast
stand by ...
Here's the forecast.
For Sunday, a low of 59, a high of 85, and Partly Cloudy skies.
For Monday, a low of 53, a high of 72, and Scattered Thunderstorms skies.

ekko: todo add Wake Up
1. Wake Up
```

Please see that web page for more details and the source code.

55.5 Where Akka fits in

As these examples show, an actor is an instance of the `Actor` class. Once created, an actor resides in memory, running in parallel to the main application thread, waiting for messages to appear in its inbox. When it receives a new message, it responds to the message with the `case` statements defined in its `receive` method. Therefore, an actor-based application can be any application that takes advantage of that programming model.

Depending on your needs, actors can provide a great approach for *reactive programming* because they can help to keep your application's UI responsive. In something like a Swing (or JavaFX) GUI application, the process looks like this:

[6] https://alvinalexander.com/amazon-echo-akka

- The user provides input through the GUI.
- Your GUI's event-handling code responds to that input event by sending a message to the appropriate actor.
- The Swing "Event Dispatch Thread" (EDT) remains responsive because the work is not being handled on the EDT.
- When the actor receives the message, it immediately delegates that work to a child actor. (I didn't show that process in this book, but you can find examples on my website and in the Scala Cookbook[7].)
- When the actor (and its children) finishes processing the message, it sends a message back, and that message results in the UI being updated (eventually being handled by `SwingUtilities.invokeLater()`, in the case of Swing).

This is exactly the way SARAH[8] works.

While the actor model isn't the only way to handle this situation, actors are a great choice when you want to create parallel processes that will live in memory for a long time, and have messages that they know how to respond to.

In the case of SARAH, it has many actors that know how to do different kinds of work, including:

- Actors to get news headlines, check my email, get stock quotes, search Google, get Twitter trends, etc.
- Actors to represent a mouth, ears, and brain, where the "ear actor" listens to your computer's microphone, the "mouth actor" speaks through the computer's speakers, and the "brain actor" knows how to process inputs and outputs, and delegate work to all of the other actors.

55.6 Key points

It bears repeating that the key things to know about Akka actors are:

- The primary purpose of actors is to create objects that live in RAM for a long

[7] http://kbhr.co/hs-cook
[8] https://alvinalexander.com/sarah

time, run on parallel threads, communicate only by message-passing, and know how to respond to one or more messages. (*Futures*, which you'll see in the next lesson, are better for "one shot," short-lived concurrency needs.)

- Messages are defined as case classes and case objects, and become the API for your actors.
- Actors respond to messages with pattern-matching statements in their `receive` method.
- To keep actors responsive, top-level actors should quickly delegate their work.
- Actors don't share any state with other actors, so there is no mutable, shared state in your application.

Akka is intended for building reactive, responsive, event-driven (message-driven), scalable systems, and the actor model *greatly* simplifies the process of working with multiple long-running threads.

55.7 See also

- The Akka website[9]
- Akka is based on the actor model, which is defined on Wikipedia[10]
- I wrote about Akka Actors in depth in the Scala Cookbook[11]
- I wrote an introductory 'Ping Pong' Akka actors example[12]
- I wrote a little Akka actors video game[13]
- You can learn more about SARAH at alvinalexander.com/sarah[14]
- My "Akkazon Ekko" application[15], which is a simple version of SARAH
- Akka was inspired by the Erlang language[16], which is used to "build massively scalable soft real-time systems with requirements on high availability"

[9] http://akka.io/
[10] https://en.wikipedia.org/wiki/Actor_model
[11] http://kbhr.co/hs-cook
[12] http://kbhr.co/hs-akka2
[13] http://kbhr.co/hs-akka7
[14] https://alvinalexander.com/sarah
[15] http://kbhr.co/hs-akka6
[16] https://www.erlang.org/

56

Futures

While an Akka actor runs for a long time and is intended to handle *many* messages over its lifetime, a Scala Future[1] is intended as a one-shot, "handle this potentially long-running computation, and call me back with a result when you're done" construct.

In this lesson I'll show how to use futures, including how to run several futures in parallel and combine their results in a for-expression, along with a few other useful `Future` methods.

> Note: If you find the name `Future` to be confusing in the following examples, I recommend replacing it with the name `ConcurrentTask`[2], which I personally find easier to understand.

56.1 Source code

You can find the source code for this lesson at this Github URL:

- github.com/alvinj/HelloScalaFutures[3]

56.2 An example in the REPL

A Scala `Future` is used to create a little pocket of concurrency that you use for one-shot needs. You typically use it when you need to call an algorithm that runs an indeterminate amount of time — such as calling a web service or executing a long-running algorithm — so you therefore want to run it off of the main application thread.

[1] https://www.scala-lang.org/api/current/scala/concurrent/Future\protect\char"0024\relax.html
[2] https://alvinalexander.com/scala/scala-future-semantics
[3] https://github.com/alvinj/HelloScalaFutures

To demonstrate how this works, let's start with an example of a `Future` in the Scala REPL. First, paste in these `import` statements:

```
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global
import scala.util.{Failure, Success}
```

Now, you can create a future. Here's a future that sleeps for a few seconds and then returns the value `42`:

```
scala> val a = Future { Thread.sleep(2000); 42 }
a: scala.concurrent.Future[Int] = Future(<not completed>)
```

While that's a simple example, it shows the basic approach: Just construct a new `Future` with your long-running algorithm.

Because `Future` has a `map` function, you use it as usual:

```
scala> val b = a.map(_ * 2)
b: scala.concurrent.Future[Int] = Future(<not completed>)
```

This shows `Future(<not completed>)`, but if you check `b`'s value you'll see that it contains the expected result of `84`:

```
scala> b
res1: scala.concurrent.Future[Int] = Future(Success(84))
```

Notice that the `84` you expected is wrapped in a `Success`, which is further wrapped in a `Future`. This is a key point to know: The value in a `Future` is always an instance of one of the `Try` types: `Success` or `Failure`. Therefore, when working with the result of a future, use the usual `Try`-handling techniques, or one of the other `Future` callback methods.

One commonly used callback method is `onComplete`, which takes a partial function[4], in which you should handle the `Success` and `Failure` cases, like this:

[4] http://kbhr.co/hs-pfuncs

```
a.onComplete {
    case Success(value) => println(s"Got the callback, value = $value")
    case Failure(e) => e.printStackTrace
}
```

When you paste that code in the REPL you'll see the result:

```
Got the callback, value = 42
```

There are other ways to process the results from futures, and I'll list the most common methods later in this lesson.

56.3 An example application

I like to use the following application to introduce futures because it's relatively simple, and it shows several key points about how to work with them:

- How to create futures
- How to combine multiple futures in a `for` expression to obtain a single result
- How to work with that result once you have it

56.3.1 A potentially slow-running method

First, imagine that you need to create a method that accesses a web service to get the current price of a stock. Because it's a web service it can be slow to return, and even fail. As a result, you define the method to run as a `Future`. It takes a stock symbol as an input parameter and returns the stock price as a `Double` inside a `Future`, so its signature looks like this:

```
def getStockPrice(stockSymbol: String): Future[Double] = ???
```

I don't want to write a method that accesses a web service for this tutorial, so I'll mock up a method that has a variable run time:

```
def getStockPrice(stockSymbol: String): Future[Double] = Future {
    val r = scala.util.Random
    val randomSleepTime = r.nextInt(3000)
    val randomPrice = r.nextDouble * 1000
```

```
    sleep(randomSleepTime)
    randomPrice
}
```

That method sleeps a random time up to 3000 ms, and also returns a random stock price. Notice how simple it is to create a method that runs as a `Future`: I just pass a block of code into the `Future` constructor to create the method body.

Next, imagine that you want to get three stock prices in parallel, and return their results once all three return. To do so, you'd write code like this:

```
package futures

import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.Future
import scala.util.{Failure, Success}

object MultipleFutures extends App {

    val startTime = currentTime

    // (a) create three futures
    val aaplFuture = getStockPrice("AAPL")
    val amznFuture = getStockPrice("AMZN")
    val googFuture = getStockPrice("GOOG")

    // (b) get a combined result in a for-expression
    val result: Future[(Double, Double, Double)] = for {
        aapl <- aaplFuture
        amzn <- amznFuture
        goog <- googFuture
    } yield (aapl, amzn, goog)

    // (c) do whatever you need to do with the results
    result.onComplete {
        case Success(x) => {
            val totalTime = deltaTime(startTime)
            println(s"In Success case, time delta: ${totalTime}")
            println(s"The stock prices are: $x")
```

```
        }
        case Failure(e) => e.printStackTrace
    }

    // important for a little parallel demo: need to keep
    // the jvm's main thread alive
    sleep(5000)

    def sleep(time: Long): Unit = Thread.sleep(time)

    // a simulated web service
    def getStockPrice(stockSymbol: String): Future[Double] = Future {
        val r = scala.util.Random
        val randomSleepTime = r.nextInt(3000)
        println(s"For $stockSymbol, sleep time is $randomSleepTime")
        val randomPrice = r.nextDouble * 1000
        sleep(randomSleepTime)
        randomPrice
    }

    def currentTime = System.currentTimeMillis()
    def deltaTime(t0: Long) = currentTime - t0

}
```

I encourage you to clone my Github project and put that code in your favorite IDE before continuing this lesson. When you do so, first run it to make sure it works as expected, then change it as desired. I recommend adding debug `println` statements to the code so you can completely understand it.

> The Github repository for this lesson also contains a class named `MultipleFuturesWithDebugOutput` that contains the same code with a lot of debug `println` statements.

56.3.2 Creating the futures

Let's walk through that code to see how it works. First, I create three futures with these lines of code:

```
val aaplFuture = getStockPrice("AAPL")
val amznFuture = getStockPrice("AMZN")
val googFuture = getStockPrice("GOOG")
```

As you saw, `getStockPrice` is defined like this:

```
def getStockPrice(stockSymbol: String): Future[Double] = Future { ...
```

If you remember the lesson on companion objects, the way the body of that method works is that the code in between the curly braces is passed into the `apply` method of `Future`'s companion object, so the compiler translates that code to something like this:

```
def getStockPrice ... = Future.apply { method body here }
                                -----
```

An important thing to know about `Future` is that it *immediately* begins running the block of code inside the curly braces. It isn't like the Java `Thread`, where you create an instance and later call its `start` method. You can see this very clearly in the debug output of the `MultipleFuturesWithDebugOutput` example, where the debug output in `getStockPrice` immediately prints three times when the AAPL, AMZN, and GOOG futures are created.

The three method calls eventually return the simulated stock prices. In fact, people often use the word *eventually* with futures because you typically use them when the return time of the algorithm is indeterminate: you don't know when you'll get a result back, you just hope to get a successful result back "eventually."

56.3.3 The `for` expression

The `for` expression in the application looks like this:

```
val result: Future[(Double, Double, Double)] = for {
    aapl <- aaplFuture
    amzn <- amznFuture
    goog <- googFuture
} yield (aapl, amzn, goog)
```

You can read this as, "Whenever `aapl`, `amzn`, and `goog` all return with their values, combine them in a tuple, and assign that value to the variable `result`." As shown, `result`

has the type `Future[(Double, Double, Double)]`, which is a tuple that contains three `Double` values, wrapped in a `Future` container.

It's important to know that the application's main thread doesn't stop when `getStockPrice` is called, and it doesn't stop at the for-expression either. In fact, if you print the result from `System.currentTimeMillis()` before and after the `for` expression, you probably won't see a difference of more than a few milliseconds. You can see that for yourself in the `MultipleFuturesWithDebugOutput` example.

56.3.4 onComplete

The final part of the application looks like this:

```
result.onComplete {
    case Success(x) => {
        val totalTime = deltaTime(startTime)
        println(s"In Success case, time delta: ${totalTime}")
        println(s"The stock prices are: $x")
    }
    case Failure(e) => e.printStackTrace
}
```

As I showed before, `onComplete` is a method that's available on a `Future`, and you use it to process a future's result as a side effect. In the same way that the `foreach` method on collections classes returns `Unit` and is only used for side effects, `onComplete` returns `Unit` and you only use it for side effects like printing the results, updating a GUI, updating a database, etc.

You can read that code as, "Whenever `result` has a final value — i.e., after *all* of the futures return in the `for` expression — come here. If everything returned successfully, run the `println` statement shown in the `Success` case. Otherwise, if an exception was thrown, go to the `Failure` case and print the exception's stack trace."

As the code implies, it's completely possible that a `Future` may fail. For example, imagine that you call a web service, but the web service is down. That `Future` instance will contain an exception, and when you call `result.onComplete`, control flows to the `Failure` case.

It's important to note that just as the JVM's main thread didn't stop at the for-expression, it doesn't block here, either. The code inside `onComplete` doesn't execute until after the for-expression assigns a value to `result`.

56.3.5 The `sleep` call

A final point to note about small examples like this is that you need to have a `sleep` call at the end of your `App`:

```
sleep(5000)
```

That call keeps the main thread of the JVM alive for five seconds. If you don't include a call like this, the JVM's main thread will exit before you get a result from the three futures, which are running on other threads. This isn't usually a problem in the real world, but it's a problem for little demos like this.

56.3.6 The other code

There are a few `println` statements in the code that use these methods:

```
def currentTime = System.currentTimeMillis()
def deltaTime(t0: Long) = System.currentTimeMillis() - t0
```

I only added a few `println` statements in this code so you can get an idea of how the application works. But as you'll see in the Github repository, I added many more `println` statements to the `MultipleFuturesWithDebugOutput` example so you can see exactly how futures work.

56.4 Other Future methods

Futures have other methods that you can use. Common callback methods are:

- `onComplete`
- `onSuccess`
- `onFailure`

In addition to those methods, futures have methods that you'll find on Scala collections classes, including:

- `filter`
- `foreach`
- `map`

Other useful and well-named methods include:

- `andThen`
- `fallbackTo`
- `recoverWith`

I discuss these methods in my article, Simple concurrency with Scala Futures[5].

56.5 Key points

While this was a short introduction, I hope those examples give you an idea of how Scala futures work. A few key points about futures are:

- You construct futures to run tasks off of the main thread
- A benefit of futures over threads is that they come with a variety of callback methods that simplify the process of working with concurrent threads, including the handling of exceptions and thread management
- A future starts running as soon as you construct it
- If you're using multiple futures to yield a single result, you'll often want to combine them in a for-expression
- Use `onComplete` and other callback methods to process the final result(s)
- The value in a `Future` is always an instance of one of the `Try` types: `Success` or `Failure`

[5] http://kbhr.co/hs-future1

56.6 See also

- My article, Simple concurrency with Scala Futures[6], provides a little more discussion about future callback methods.
- I wrote a little demo GUI application named Future Board that works a little like Flipboard[7]. You can find an image of the application at this URL[8], and you can find the source code for Future Board in this Github repository[9].
- The "Futures and Promises"[10] on scala-lang.org is a good resource

These are some of the best books I know about programming with Akka, futures, and JVM concurrency in general:

- Learning Concurrent Programming in Scala[11]
- Akka Concurrency[12]
- Java Concurrency in Practice[13]

[6] http://kbhr.co/hs-future1
[7] https://flipboard.com/
[8] http://kbhr.co/hs-flipboard
[9] https://github.com/alvinj/FPFutures
[10] http://docs.scala-lang.org/overviews/core/futures.html
[11] http://amzn.to/2fWn70c
[12] http://amzn.to/2xhUNd4
[13] http://amzn.to/2fdFfSK

57
Summary

I hope you enjoyed this book as a quick, gentle introduction to the Scala programming language, and I hope I was able to share some of the beauty of the language.

57.1 Best Scala books

To help you learn more about Scala, here are some of the best resources I know. First, as a special mention, Programming in Scala[1] is written by Martin Odersky (the creator of Scala), Bill Venners (creator of ScalaTest and more), and Lex Spoon, and I consider it to be *the* reference for the Scala language.

In alphabetical order, I've read these other books, and I can recommend them:

- Akka Concurrency[2]
- Once you know about functional programming, Functional and Reactive Domain Modeling[3] is a good resource
- Functional Programming in Scala[4] is a good resource for learning about FP
- Java Concurrency in Practice[5]
- Learning Concurrent Programming in Scala[6]
- Once you've had an introduction to Scala (such as in this book), Scala for the Impatient[7] is a good quick reference guide

[1] http://kbhr.co/hs-ps
[2] http://kbhr.co/hs-akka-con
[3] http://kbhr.co/hs-frdm
[4] http://kbhr.co/hs-fpis
[5] http://kbhr.co/hs-concurrency
[6] http://kbhr.co/hs-cpis
[7] http://kbhr.co/hs-simp

57.2 My other books

My other books on Scala are:

- Scala Cookbook[8]
- Functional Programming, Simplified[9]

The Cookbook shares the most common recipes for working with Scala, and the second book attempts to make learning functional programming as simple as possible.

Other books I've written include:

- How I Sold My Business: A Personal Diary[10]
- A Survival Guide for New Consultants[11]

57.3 Thank you!

Thank you again for reading this book.

All the best,
Al

[8] http://kbhr.co/hs-cook
[9] http://kbhr.co/hs-fps
[10] http://kbhr.co/hs-hismb
[11] http://kbhr.co/hs-consult

Index