Learn Scala fast
with small, easy lessons

ALVIN ALEXANDER

FREE PREVIEW!

Hello, Scala

Alvin Alexander

Learn Scala fast
with small, easy lessons

Copyright

Hello, Scala

Copyright 2018 Alvin J. Alexander[1]

All rights reserved. No part of this book may be reproduced without prior written permission from the author.

This book is presented solely for educational purposes. While best efforts have been made to prepare this book, the author makes no representations or warranties of any kind and assumes no liabilities of any kind with respect to the accuracy or completeness of the contents, and specifically disclaims any implied warranties of merchantability or fitness of use for a particular purpose. The author shall not be held liable or responsible to any person or entity with respect to any loss or incidental or consequential damages caused, or alleged to have been caused, directly or indirectly, by the information or programs contained herein. Any use of this information is at your own risk.

Version 1.0, published September 3, 2018

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

[1] https://alvinalexander.com
[2] https://alvinalexander.com/hello-scala

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

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.

29.1 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

29.2 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]

29.3 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