Scala Book

Alvin Alexander, et al.

Learn Scala fast
with small, easy lessons
Gratitude: Many thanks to those who have contributed suggestions and editing to this book, including Julien Richard-Foy, Seth Tisue, Marcelo de Oliveira Rosa, and others who I may have missed. All the best, Al.
Contents

1 Introduction 1
2 Prelude: A Taste of Scala 3
3 Preliminaries 15
4 Scala Features 17
5 Hello, World 19
6 Hello, World - Version 2 23
7 The Scala REPL 25
8 Two Types of Variables 27
9 The Type is Optional 31
10 A Few Built-In Types 33
11 Two Notes About Strings 37
12 Command-Line I/O 41
13 Control Structures 43
14 The if/then/else Construct 45
15 for Loops 47
16 for Expressions 51
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>match Expressions</td>
<td>55</td>
</tr>
<tr>
<td>18</td>
<td>try/catch/finally Expressions</td>
<td>61</td>
</tr>
<tr>
<td>19</td>
<td>Scala Classes</td>
<td>63</td>
</tr>
<tr>
<td>20</td>
<td>Auxiliary Class Constructors</td>
<td>69</td>
</tr>
<tr>
<td>21</td>
<td>Supplying Default Values for Constructor Parameters</td>
<td>71</td>
</tr>
<tr>
<td>22</td>
<td>A First Look at Scala Methods</td>
<td>73</td>
</tr>
<tr>
<td>23</td>
<td>Enumerations (and a Complete Pizza Class)</td>
<td>77</td>
</tr>
<tr>
<td>24</td>
<td>Scala Traits and Abstract Classes</td>
<td>83</td>
</tr>
<tr>
<td>25</td>
<td>Using Scala Traits as Interfaces</td>
<td>85</td>
</tr>
<tr>
<td>26</td>
<td>Using Scala Traits Like Abstract Classes</td>
<td>89</td>
</tr>
<tr>
<td>27</td>
<td>Abstract Classes</td>
<td>95</td>
</tr>
<tr>
<td>28</td>
<td>Scala Collections</td>
<td>99</td>
</tr>
<tr>
<td>29</td>
<td>The ArrayBuffer Class</td>
<td>101</td>
</tr>
<tr>
<td>30</td>
<td>The List Class</td>
<td>105</td>
</tr>
<tr>
<td>31</td>
<td>The Vector Class</td>
<td>109</td>
</tr>
<tr>
<td>32</td>
<td>The Map Class</td>
<td>111</td>
</tr>
<tr>
<td>33</td>
<td>The Set Class</td>
<td>115</td>
</tr>
<tr>
<td>34</td>
<td>Anonymous Functions</td>
<td>119</td>
</tr>
<tr>
<td>35</td>
<td>Common Sequence Methods</td>
<td>125</td>
</tr>
<tr>
<td>36</td>
<td>Common Map Methods</td>
<td>133</td>
</tr>
<tr>
<td>CONTENTS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td>---------</td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>A Few Miscellaneous Items 137</td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>Tuples 139</td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>An OOP Example 143</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>SBT and ScalaTest 149</td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>The Scala Build Tool (SBT) 151</td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>Using ScalaTest with SBT 157</td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>Writing BDD Style Tests with ScalaTest and SBT 161</td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>Functional Programming 165</td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>Pure Functions 167</td>
<td></td>
</tr>
<tr>
<td>46</td>
<td>Passing Functions Around 171</td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>No Null Values 175</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>Companion Objects 183</td>
<td></td>
</tr>
<tr>
<td>49</td>
<td>Case Classes 191</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>Case Objects 197</td>
<td></td>
</tr>
<tr>
<td>51</td>
<td>Functional Error Handling in Scala 201</td>
<td></td>
</tr>
<tr>
<td>52</td>
<td>Concurrency 205</td>
<td></td>
</tr>
<tr>
<td>53</td>
<td>Scala Futures 207</td>
<td></td>
</tr>
<tr>
<td>54</td>
<td>Where To Go Next 217</td>
<td></td>
</tr>
</tbody>
</table>
1

Introduction

In these pages, *Scala Book* provides a quick introduction and overview of the Scala programming language. The book is written in an informal style, and consists of more than 50 small lessons. Each lesson is long enough to give you an idea of how the language features in that lesson work, but short enough that you can read it in fifteen minutes or less.

One note before beginning:

- In regards to programming style, most Scala programmers indent their code with two spaces, but we use four spaces because we think it makes the code easier to read, especially in a book format.

To begin reading, click the “next” link, or select the *Prelude: A Taste of Scala* lesson in the table of contents.
Prelude: A Taste of Scala

Our hope in this book is to demonstrate that Scala is a beautiful, modern, expressive programming language. To help demonstrate that, in this first chapter we'll jump right in and provide a whirlwind tour of Scala's main features. After this tour, the book begins with a more traditional “Getting Started” chapter.

In this book we assume that you’ve used a language like Java before, and are ready to see a series of Scala examples to get a feel for what the language looks like. Although it’s not 100% necessary, it will also help if you’ve already downloaded and installed Scala so you can test the examples as you go along. You can also test these examples online with ScalaFiddle.io.

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
- Scala code results in .class files that run on the Java Virtual Machine (JVM)
- It’s easy to use Java libraries in Scala

2.2 Hello, world

Ever since the book, C Programming Language, 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

If you’re coming to Scala from Java, 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

We share more “Hello, world” examples in the lessons that follow, so we’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. We 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:

$ scala
Welcome to Scala 2.13.0 (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. Inside the REPL you type Scala expressions to see how they work:
2.4. TWO TYPES OF VARIABLES

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

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

As those examples show, after you type your expressions in the REPL, it shows the result of each 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:

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

2.5 Declaring variable types

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

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

When you do this, Scala can usually infer the data type for you, as shown in these REPL examples:

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

scala> val s = "a string"
val s: String = a string
```
This feature is known as *type inference*, and it’s a great way to help keep your code concise. You can also *explicitly* declare a variable’s type, but that’s not usually necessary:

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

As you can see, that code looks unnecessarily verbose.

### 2.6 Control structures

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

#### 2.6.1 if/else

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

```scala
if (test1) {
  doA()
} else if (test2) {
  doB()
} else if (test3) {
  doC()
} else {
  doD()
}
```

However, unlike Java and many other languages, the if/else construct returns a value, so, among other things, you can use it as a ternary operator:

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

#### 2.6.2 match expressions

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

```scala
val result = i match {
  case 1 => "one"
  case 2 => "two"
  case _ => "other"
```

```scala
  case _ => "other"
```
case 1 => "one"
case 2 => "two"
case _ => "not 1 or 2"
}

The `match` expression isn't limited to just integers, it can be used with any data type, including booleans:

```
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, and we share more examples of it later in this book.

### 2.6.3 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)
}```
2.6.4 for loops and expressions

Scala for loops — which we generally write in this book as for-loops — look like this:

```scala
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 5:

```scala
val x = for (i <- 1 to 5) yield i * 2
```

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

```scala
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, we’ll imagine that you can guess how this code works, even if you’ve never seen a for-expression or Scala list until now.

2.6.5 while and do/while

Scala also has while and do/while loops. Here’s their general syntax:

```scala
// while loop
while(condition) {
  statement(a)
  statement(b)
}
```
// do-while
do {
    statement(a)
    statement(b)
} while(condition)

2.7 Classes

Here's an example of a Scala class:

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

This is how you use that class:

```scala
val p = new Person("Julia", "Kern")
println(p.firstName)
p.lastName = "Manes"
p.printFullName()
```

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:

```scala
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. The CrustSize, CrustType, and Topping classes aren't shown, but you can probably understand how that code works
without needing to see those classes.

2.8 Scala methods

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

```scala
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:

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

This is how you call those methods:

```scala
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.

2.9 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:

```scala
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")
}
```
2.10. COLLECTIONS CLASSES

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 multiple 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, we explain it in detail later in the book.

2.10 Collections classes

If you’re coming to Scala from Java and you’re ready to really jump in and learn Scala, it's possible to use the Java collections classes in Scala, and some people do so for several weeks or months while getting comfortable with Scala. But it's highly recommended that you learn the basic Scala collections classes — List, ListBuffer, Vector, ArrayBuffer, Map, and Set — as soon as possible. A great benefit of the Scala collections classes is that they offer many powerful methods that you’ll want to start using as soon as possible to simplify your code.

2.10.1 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

2.10.2 Sequence methods

While there are many sequential collections classes you can use — Array, ArrayBuffer, Vector, List, and more — let's look at some examples of what you can do with the 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 without any explanation 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 foldLeft:

```scala
cscala> nums.foldLeft(0)(_ + _)
res0: Int = 55

cscala> nums.foldLeft(1)(_ * _)
res1: Int = 3628800
```

Once you know that the first parameter to foldLeft is a seed value, 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, and many of them will be demonstrated in the collections lessons that follow, but hopefully this gives you an idea of their power.

For more details, jump to the Scala Book collections lessons, or see the Mutable and Immutable collections overview for more details and examples.

### 2.11 Tuples

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

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

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

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

You can access the tuple values by number:

```scala
t._1
t._2
```
Or assign the tuple fields to variables:

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

Tuples are nice for those times when you need to put a little “bag” of things together for a little while.

### 2.12 What we haven’t shown

While that was whirlwind introduction to Scala in about ten printed pages, there are many things we haven’t shown yet, including:

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

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

### 2.13 A bit of background

Scala was created by Martin Odersky, who studied under Niklaus Wirth, 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.
In this book we assume that you’re familiar with another language like Java, so we don’t spend much time on programming basics. That is, we assume that you’ve seen things like for-loops, classes, and methods before, so we generally only write, “This is how you create a class in Scala,” that sort of thing.

That being said, there are a few good things to know before you read this book.

3.1 Installing Scala

First, to run the examples in this book you’ll need to install Scala on your computer. See our general Getting Started page for details on how to use Scala (a) in an IDE and (b) from the command line.

3.2 Comments

One good thing to know up front is that comments in Scala are just like comments in Java (and many other languages):

```
// a single line comment

/*
 * a multiline comment
 */

/**
 * also a multiline comment
 */
```
3.3 IDEs

The three main IDEs (integrated development environments) for Scala are:

- IntelliJ IDEA
- Visual Studio Code
- Scala IDE for Eclipse

3.4 Naming conventions

Another good thing to know is that Scala naming conventions follow the same “camel case” style as Java:

- Class names: Person, StoreEmployee
- Variable names: name, firstName
- Method names: convertToInt, toUpper
Scala Features

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 (the father of javac), and influenced by Java, Ruby, Smalltalk, ML, Haskell, Erlang, and others.
- It’s a high-level language.
- It’s statically typed.
- It has a sophisticated type inference system.
- Its 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.
- 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. Some people say that Scala will change the way you think about programming (and that’s a good thing).
- A great Scala benefit 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*.
Since the release of the book, *C Programming Language*, most programming books have begun with a simple “Hello, world” example, and in keeping with tradition, here’s the source code for a Scala “Hello, world” example:

```scala
total 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
```

5.1 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 single instance of that class
  - If you’re coming to Scala from Java, this means that `main` is just like a static method (We 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:

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

### 5.2 Going deeper: Scala creates .class files

As we 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:

```bash
$ 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.
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:

```scala
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. We’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`.

We haven't mentioned it yet, but 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!)

### 6.1 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`:

```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:

- 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).
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 this:

$ scala
Welcome to Scala 2.13.0 (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.

7.1 Variables created as needed

Note that 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")

In addition to the REPL there are a couple of other, similar tools you can use:

- Scastie is “an interactive playground for Scala” with several nice features, including being able to control build settings and share code snippets
- IntelliJ IDEA has a Worksheet plugin that lets you do the same things inside your IDE
- The Scala IDE for Eclipse also has a Worksheet plugin
- scalafiddle.io lets you run similar experiments in a web browser

For more information on the Scala REPL, see the Scala REPL overview
Two Types of Variables

In Java you declare new variables like this:

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

Each variable declaration is preceded by its type.

By contrast, Scala has 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:

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

val p = new Person("Joel Fleischman")
```

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. We say that the variable's type is *inferred* by the compiler. You can also *explicitly* declare the variable type if you prefer:

```scala
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.

As a practical matter it can help to explicitly show the type when you’re
working with methods in third-party libraries, especially if you don’t use the library often, or if their method names don’t make the type clear.

8.1 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
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
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.

8.2 “Hello, world” with a `val` field

Here’s what a “Hello, world” app looks like with a `val` field:

```scala
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

8.3 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 redefine a val field in the REPL, like this:

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

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

val fields can’t be redefined like that in the real world, but they can be redefined in the REPL playground.
As we showed in the previous lesson, when you create a new variable in Scala you can *explicitly* declare its type, like this:

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

However, you can generally leave the type off and Scala can infer it for you:

```scala
val count = 1
val name = "Alvin"
```

In most cases your code is easier to read when you leave the type off, so this inferred form is preferred.

### 9.1 The explicit form feels verbose

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:

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

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

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

In summary:

```scala
val p = new Person("Candy") // preferred
val p: Person = new Person("Candy") // unnecessarily verbose
```
9.2 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. We show this in the next lesson.
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:

```scala
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:

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

The REPL shows that those examples default to Int and Double:

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

scala> val x = 1.0
x: Double = 1.0
```
Those data types and their ranges are:

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Possible Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>true or false</td>
</tr>
<tr>
<td>Byte</td>
<td>8-bit signed two's complement integer (-2^7 to 2^7-1, inclusive) -128 to 127</td>
</tr>
<tr>
<td>Short</td>
<td>16-bit signed two's complement integer (-2^15 to 2^15-1, inclusive) -32,768 to 32,767</td>
</tr>
<tr>
<td>Int</td>
<td>32-bit two's complement integer (-2^31 to 2^31-1, inclusive) -2,147,483,648 to 2,147,483,647</td>
</tr>
<tr>
<td>Long</td>
<td>64-bit two's complement integer (-2^63 to 2^63-1, inclusive) (-2^63 to 2^63-1, inclusive)</td>
</tr>
<tr>
<td>Float</td>
<td>32-bit IEEE 754 single-precision float 1.40129846432481707e-45 to 3.40282346638528860e+38</td>
</tr>
<tr>
<td>Double</td>
<td>64-bit IEEE 754 double-precision float 4.94065645841246544e-324d to 1.79769313486231570e+308d</td>
</tr>
<tr>
<td>Char</td>
<td>16-bit unsigned Unicode character (0 to 2^16-1, inclusive) 0 to 65,535</td>
</tr>
<tr>
<td>String</td>
<td>a sequence of Char</td>
</tr>
</tbody>
</table>

### 10.1 BigInt and BigDecimal

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

```scala
def main() = {
  var b = BigInt(1234567890)
  var b = BigDecimal(123456.789)

  println(s"A great thing about BigInt and BigDecimal is that they support all the operators you're used to using with numeric types:

  scala> var b = BigInt(1234567890)
b: scala.math.BigInt = 1234567890

  scala> b + b
  res0: scala.math.BigInt = 2469135780

  scala> b * b
  res1: scala.math.BigInt = 1524157875019052100
  
  ""
}
```

```
10.2 String and Char

Scala also has `String` and `Char` data types, which you can generally declare with the implicit form:

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

Though once again, you can use the explicit form, if you prefer:

```scala
val name: String = "Bill"
val c: Char = 'a'
```

As shown, enclose strings in double-quotes and a character in single-quotes.
Two Notes About Strings

Scala strings have a lot of nice features, but we want to take a moment to highlight two features that we’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:

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

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

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

However, Scala provides this more convenient form:

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

This form creates a very readable way to print strings that contain variables:

```scala
println(s"Name: $firstName $mi $lastName")
```

As shown, all you have to do 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.

### 11.0.1 More features

String interpolation in Scala provides many more features. For example, you can also enclose your variable names inside curly braces:

```scala
println(s"Name: ${firstName} ${mi} ${lastName}")
```

For some people that’s easier to read, but an even more important benefit is that you
can put expressions inside the braces, as shown in this REPL example:

```
scala> println(s"1+1 = ${1+1}")
1+1 = 2
```

A few other benefits of string interpolation are:

- You can precede strings with the letter `f`, which lets you use `printf` style formatting inside strings
- The `raw` interpolator performs no escaping of literals (such as `\n`) within the string
- You can create your own string interpolators

See the string interpolation documentation for more details.

### 11.1 Multiline strings

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

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

12.1 Writing output

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

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

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

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

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

```scala
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 data types like String, Int, Float, etc.

12.2 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 use it, you need to first import it, like this:

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

To demonstrate how this 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

12.2.1 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.
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 unique constructs, including:

- match expressions
- for expressions

We'll demonstrate these in the following lessons.
The if/then/else Construct

A basic Scala if statement looks like this:

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

You can also write that statement like this:

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

The if/else construct looks like this:

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

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

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

14.1 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 we did in the previous examples, but a more common approach —
especially in functional programming — is to assign the result to a variable:

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

This is cool for several reasons, including the fact that it means that Scala doesn't require a special “ternary” operator.

### 14.2 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:

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

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

```scala
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. The differences between expressions and statements will also become more apparent.
15

for 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:

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

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

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

This is what the result looks like in the Scala REPL:

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

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

That example uses a sequence of integers, which has the data type Seq[Int]. Here’s a list of strings which has the data type List[String]:

```scala
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)

Seq and List are two types of linear collections. In Scala these collection classes are preferred over Array. (More on this later.)

15.1 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)

foreach is available on most collections classes, including sequences, maps, and sets.

15.2 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:

```scala
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 movie names and ratings using for like this:

```scala
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 that’s assigned to each value in the map.
You can also print the ratings with `foreach` like this:

```scala
ratings.foreach {
    case(movie, rating) => println(s"key: $movie, value: $rating")
}
```
If you recall what we wrote about Expression-Oriented Programming (EOP) and the difference between expressions and statements, you’ll notice that in the previous lesson we used the for keyword and foreach method as tools for side effects. We used them to print the values in the collections to STDOUT using println. Java has similar keywords, and many programmers used them for years without ever giving much thought to how they could be improved.

Once you start working with Scala you’ll see that in functional programming languages you can use more powerful “for expressions” in addition to “for loops.” In Scala, a for expression — which we’ll write as 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 new collections from existing collections.

For example, given this list of integers:

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

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

```scala
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:

```scala
List(2,4,6)
```
In summary, 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`.

### 16.1 Capitalizing a list of strings

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

```scala
def cap(s: String) = s.capitalize
val names = List("adam", "david", "frank")
```

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

```scala
def cap(s: String) = s.capitalize
val ucNames = for (name <- names) yield cap(name)
```

The REPL shows how this works:

```scala
caps> val ucNames = for (name <- names) yield cap(name)
ucNames: List[String] = List(Adam, David, Frank)
```

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

### 16.2 The `yield` keyword

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

```scala
def cap(s: String) = s.capitalize
val doubledNums = for (n <- nums) yield n * 2
val ucNames = for (name <- names) yield cap(name)
```

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.”
16.3 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:

```scala
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 `drop(1)` on each `String`. After you do that, you call the `capitalize` method on each string. Here's how you can use a for-expression to solve this problem:

```scala
val capNames = for (name <- names) yield {
    val nameWithoutUnderscore = name.drop(1)
    val capName = nameWithoutUnderscore.capitalize
    capName
}
```

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

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

16.3.1 A shorter version of the solution

We 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:

```scala
val capNames = for (name <- names) yield name.drop(1).capitalize
```

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

```scala
val capNames = for (name <- names) yield { name.drop(1).capitalize }
```
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:

```scala
// 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 the default with a variable so you can print it
  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 we 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:

```scala
val monthName = i match {
  case 1 => "January"
  case 2 => "February"
  case 3 => "March"
  // catch the default with a variable so you can return it
  case _ => "Invalid month"
}
```
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.

17.1 Aside: A quick look at Scala methods

Scala also makes it easy to use a match expression as the body of a method. We haven’t shown how to write Scala methods yet, so as a brief introduction, here’s a method named `convertBooleanToStringMessage` that takes a `Boolean` value and returns a `String`:

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

Hopefully you can see how that method works, even though we won’t go into its details. These examples show how it works when you give it the `Boolean` values `true` and `false`:

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

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

17.2 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 as an input parameter and returning
17.3. HANDLING ALTERNATE CASES

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

```scala
def convertBooleanToStringMessage(bool: Boolean): String = bool match {
  case true => "you said true"
  case false => "you said false"
}
```

The body of that method is just 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.

This is how you call that method and then print its result:

```scala
val result = convertBooleanToStringMessage(true)
println(result)
```

Using a match expression as the body of a method is also a common use.

### 17.3 Handling alternate cases

Match expressions are extremely powerful, and we'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:

```scala
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 one case statement lets both 0 and the empty string evaluate to false:

\[
\text{case } 0 \lor "" \Rightarrow \text{false}
\]

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

```scala
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:

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

### 17.4 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
17.4. USING IF EXPRESSIONS IN CASE STATEMENTS

statements both use if expressions to match ranges of numbers:

```scala
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:

```scala
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 is easier to read. Here's one example:

```scala
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 curly braces:

```scala
count match {
  case 1 => {
    println("one, a lonely number")
  }
  case x if x == 2 || x == 3 => {
  }
  case x if x == 2 || x == 3 => {
```
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:

```scala
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: " + b)
  case c if 20 to 29 contains c => println("20-29 range: " + c)
  case _ => println("Hmmm...")
}
```

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

```scala
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)
}
```

### 17.5 Even more

Match expressions are very powerful, and there are even more things you can do with them, but hopefully these examples provide a good start towards using them.
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.

### 18.1 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:

```scala
var text = ""
try {
  text = openAndReadAFile(filename)
} catch {
  case e: FileNotFoundException => println("Couldn't find that file.")
  case e: IOException => println("Had an IOException trying to read that file")
}
```

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.

### 18.2 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:

```scala
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 exception")
} finally {
    // your scala code here, such as closing a database connection
    // or file handle
}

18.3 More later

We’ll cover more details about Scala’s try/catch/finally syntax in later lessons, such as in the “Functional Error Handling” lessons, but these examples demonstrate how the syntax works. A great thing about the syntax is that it’s consistent with the match expression syntax. This makes your code consistent and easier to read, and you don’t have to remember a special/different syntax.
In support of object-oriented programming (OOP), Scala provides a `class` construct. The syntax is much more concise than languages like Java and C#, but it’s also still easy to use and read.

### 19.1 Basic class constructor

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

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

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

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

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 roughly 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;
    }

}

19.2 \texttt{val} makes fields read-only

In that first example both fields were defined as \texttt{var} fields:

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

That makes those fields mutable. You can also define them as \texttt{val} fields, which makes
them immutable:

```scala
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
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"
  ^
```

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

### 19.3 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:

```scala
class Person(var firstName: String, var lastName: String) {
  println("the constructor begins")
```
// 'public' access by default
var age = 0

// some class fields
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")

}

This code in the Scala REPL demonstrates how this class 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

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
When you come to Scala from a more verbose language this constructor approach might feel a little unusual at first, but once you understand and write a couple of classes with it, you’ll find it to be logical and convenient.

19.4 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)

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

// 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
)
Auxiliary Class Constructors

You define auxiliary Scala class constructors by defining methods that are named `this`. There are only a few rules to know:

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

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

```scala
val DefaultCrustSize = 12
val DefaultCrustType = "THIN"

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

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

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

  // zero-arg auxiliary constructor
  def this() {
    this(DefaultCrustSize, DefaultCrustType)
  }

  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:

```scala
val p1 = new Pizza(DefaultCrustSize, DefaultCrustType)
val p2 = new Pizza(DefaultCrustSize)
val p3 = new Pizza(DefaultCrustType)
val p4 = new Pizza
```

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

### 20.1 Notes

There are two important notes to make about this example:

- The `DefaultCrustSize` and `DefaultCrustType` variables are not a preferred way to handle this situation, but because we haven't shown how to handle enumerations yet, we use this approach to keep things simple.

- Auxiliary class constructors are a great feature, but because you can use default values for constructor parameters, you won't need to use this feature very often. The next lesson demonstrates how using default parameter values like this often makes auxiliary constructors unnecessary:

```scala
class Pizza(
  var crustSize: Int = DefaultCrustSize,
  var crustType: String = DefaultCrustType
)
```
Scala lets you supply default values for constructor parameters. For example, in previous lessons we showed that you can define a Socket class like this:

```scala
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 better by supplying default values for the timeout and linger parameters:

```scala
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:

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

Here’s what those examples look like in the REPL:

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

scala> new Socket(1000)
res1: Socket = timeout: 1000, linger: 3000

scala> new Socket(4000, 6000)
```
CHAPTER 21. SUPPLYING DEFAULT VALUES FOR CONSTRUCTOR PARAMETERS

res2: Socket = timeout: 4000, linger: 6000

21.0.1 Benefits

Supplying default constructor parameters has at least two benefits:

- You provide preferred, default values for your parameters
- You let consumers of your class override those values for their own needs

As shown in the examples, a third benefit is that it lets consumers construct new Socket instances in at least three different ways, as if it had three class constructors.

21.1 Bonus: Named parameters

Another nice thing about Scala is that you can use named parameters when creating a new instance of a class. For instance, given this class:

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

you can create a new Socket like this:

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

This feature comes in handy from time to time, such as when all of the class constructor parameters have the same type, such as the Int parameters in this example. For example, some people find that this code:

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

is more readable than this code:

```scala
val s = new Socket(2000, 3000)
```
A First Look at Scala 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 will show some examples of methods so you can see what the syntax looks like.

### 22.1 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:

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

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

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

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

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

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

scala> double(10)
```
22.2 Showing the method’s return type

The previous example didn’t show the method’s return type, but you can show it:

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

Writing a method like this explicitly declares the method’s return type. Some people prefer to explicitly declare method return types because it makes the code easier to maintain weeks, months, and years in the future.

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

22.3 Methods with multiple input parameters

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

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

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

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

Here’s a method that takes three input parameters:

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

22.4 Multiline methods

When a method is only one line long you can use the format shown, but when the method body gets longer, you put the multiple lines inside curly braces:

```scala
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
Enumerations (and a Complete Pizza Class)

If we demonstrate enumerations next, we can also show you what an example Pizza class looks like when written in an object-oriented manner. So that’s the path we’ll take.

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 we’re jumping ahead a little bit, we’re not going to explain this syntax too much, but this is how you create an enumeration for the days of a week:

```scala
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
```

As shown, just declare a base trait and then extend that trait with as many case objects as needed.

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

```scala
sealed trait Suit
case object Clubs extends Suit
case object Spades extends Suit
case object Diamonds extends Suit
case object Hearts extends Suit
```
We’ll discuss traits and case objects later in this book, but if you’ll trust us for now that this is how you create enumerations, we can then create a little OOP version of a Pizza class in Scala.

23.1 Pizza-related enumerations

Given that (very brief) introduction to enumerations, we can now create pizza-related enumerations like this:

```scala
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.

23.2 A sample Pizza class

Given those enumerations, we can define a Pizza class like this:

```scala
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’ll see that you can compile it with the usual command:

$ scalac Pizza.scala

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

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

23.3 A complete Pizza class with a main method

If you're ready to have some fun, copy all of the following source code and paste it into a file named Pizza.scala:

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

Now, 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)

That code combines several different concepts — including two things we haven’t discussed yet in the import statement and the ArrayBuffer — but if you have experience with Java and other languages, hopefully it’s not too much to throw at you at one time.

At this point we 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.
CHAPTER 23. ENUMERATIONS (AND A COMPLETE PIZZA CLASS)
Scala traits are a great feature of the language. As you’ll see in the following lessons, you can use them just like a Java interface, and you can also use them like abstract classes that have real methods. Scala classes can also extend and “mix in” multiple traits.

Scala also has the concept of an abstract class, and we’ll show when you should use an abstract class instead of a trait.
25

Using Scala Traits as Interfaces

25.1 Using Scala Traits as Interfaces

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

25.2 A simple example

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

```scala
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:

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

25.3 Extending a trait

Given this trait:

```scala
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")
}

You can also write those methods like this, if you prefer:

class Dog extends TailWagger {
    def startTail() = println("tail is wagging")
    def stopTail() = println("tail is stopped")
}

Notice that in either case, you use the extends keyword to create a class that extends a single trait:

class Dog extends TailWagger {...

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

This demonstrates how you implement a single Scala trait with a class that extends the trait.
25.4 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:

```scala
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")

}

Notice how extends and with are used to create a class from multiple traits:

class Dog extends Speaker with TailWagger with Runner {

```
Key points of this code:

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

From what you’ve seen so far, Scala traits work just like Java interfaces. But there’s more ...
Using Scala Traits Like Abstract Classes

In the previous lesson we showed how to use Scala traits like the original Java interface, but they have much more functionality than that. You can also add real, working methods to them and use them like abstract classes, or more accurately, as mixins.

26.1 A first example

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

```scala
trait Pet {
  def speak = println("Yo") // concrete implementation of a speak method
  def comeToMaster(): Unit // abstract
}
```

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

```scala
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:

```scala
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!

26.2 Overriding an implemented method

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

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

scala> c.speak
meow

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

26.3 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:

```scala
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
26.4 Mixing traits in on the fly

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

```scala
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:

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

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

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

Once again the REPL shows that this works:

```scala
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).
Scala also has a concept of an abstract class that is 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

### 27.1 Scala traits don’t allow constructor parameters

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

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

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

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

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

### 27.2 When Scala code will be called from Java code

Regarding the second point — the second time when you’ll need to use an abstract class — 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.
27.3 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 we defined in the previous lesson:

```scala
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:

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

cscala> d.speak
Woof

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

27.3.1 Notice how `name` was passed along

All of that code is similar to Java, so we 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 `Pet` constructor:

```scala
class Dog(name: String) extends Pet(name) {
}
```

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

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

```scala
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
```

We encourage you to copy and paste that code into the REPL to be sure that it works as expected, and then experiment with it as desired.
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 as they’re intended to be used. As one author of this book has said, “Speaking from personal experience, when I first started working with Scala I tried to use Java collections classes in my Scala code, and all that did was slow down my progress.”

### 28.1 The main Scala collections classes

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

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArrayBuffer</td>
<td>an indexed, mutable sequence</td>
</tr>
<tr>
<td>List</td>
<td>a linear (linked list), immutable sequence</td>
</tr>
<tr>
<td>Vector</td>
<td>an indexed, immutable sequence</td>
</tr>
<tr>
<td>Map</td>
<td>the base <code>Map</code> (key/value pairs) class</td>
</tr>
<tr>
<td>Set</td>
<td>the base <code>Set</code> class</td>
</tr>
</tbody>
</table>

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

We’ll demonstrate the basics of these classes in the following lessons.

In the following lessons on Scala collections classes, whenever we use the word *immutable*, it’s safe to assume that the class is intended for use in a *functional programming* (FP) style. With these classes you don’t modify the collection; you apply functional methods to the collection to create a new result. You’ll see what this means in the examples that follow.
The ArrayBuffer Class

If you’re an OOP developer coming to Scala from Java, the ArrayBuffer class will probably be most comfortable for you, so we’ll demonstrate it first. It’s a mutable sequence, so you can use its methods to modify its contents, and those methods are similar to methods on Java sequences.

To use an ArrayBuffer you must first import it:

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

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

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

Once you have an ArrayBuffer you add elements to it in a variety of ways:

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

The REPL shows how += works:

```scala
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:

```scala
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 multiple elements
nums += 5 += 6

// add multiple elements from another collection
nums ++= List(7, 8, 9)

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

// remove one element
nums -= 9

// remove multiple elements
nums -= 7 -= 8

// remove multiple elements using another collection
nums --= Array(5, 6)

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

scala> import scala.collection.mutable:ArrayBuffer

scala> val nums = ArrayBuffer(1, 2, 3)
val nums: ArrayBuffer[Int] = ArrayBuffer(1, 2, 3)

scala> nums += 4
val res0: ArrayBuffer[Int] = ArrayBuffer(1, 2, 3, 4)

scala> nums += 5 += 6
val res1: ArrayBuffer[Int] = ArrayBuffer(1, 2, 3, 4, 5, 6)

scala> nums ++= List(7, 8, 9)
val res2: ArrayBuffer[Int] = ArrayBuffer(1, 2, 3, 4, 5, 6, 7, 8, 9)

scala> nums -= 9
29.1. **MORE WAYS TO WORK WITH ArrayBuffer**

As a brief overview, here are several methods you can use with an `ArrayBuffer`:

```scala
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.insertAll(0, Vector(4, 5, 6, 7)) // 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')
```
The List class is a linear, immutable sequence. All this means is that it’s a linked-list that you can’t modify. Any time you want to add or remove List elements, you create a new List from an existing List.

30.1 Creating Lists

This is how you create an initial List:

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

You can also declare the List’s type, if you prefer, though it generally isn’t necessary:

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

30.2 Adding elements to a List

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

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

You prepend elements to a List like this:

```scala
val b = 0 :: a
```

and this:

```scala
val b = List(-1, 0) ::: a
```
The REPL shows how this works:

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

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

You can also append elements to a List, but because List is a singly-linked list, you should really only prepend elements to it; appending elements to it is a relatively slow operation, especially when you work with large sequences.

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

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

### 30.3 How to remember the method names

These days, IDEs help us out tremendously, but one way to remember those method names is to think that the `:` character represents the side that the sequence is on, so when you use `+:` you know that the list needs to be on the right, like this:

```
0 +: a
```

Similarly, when you use `:+` you know the list needs to be on the left:

```
a :+: 4
```

There are more technical ways to think about this, this can be a simple way to remember the method names.

One good thing about these method names: they’re consistent. The same method names are used with other immutable sequence classes, such as Seq and Vector.
30.4  How to loop over lists

We showed how to loop over lists earlier in this book, but it’s worth showing the syntax again. Given a List like this:

```scala
val names = List("Joel", "Chris", "Ed")
```

you can print each string like this:

```scala
for (name <- names) println(name)
```

This is what it looks like in the REPL:

```
scala> for (name <- names) println(name)
Joel
Chris
Ed
```

A great thing about this approach is that it works with all sequence classes, including ArrayBuffer, List, Seq, Vector, etc.

30.5  A little bit of history

If you’re interested in a little bit of history, the List class is very similar to the List class from the Lisp programming language. Indeed, in addition to creating a List like this:

```scala
val ints = List(1, 2, 3)
```

you can also create the exact same list this way:

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

The REPL shows how this works:

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

This works because a List is a singly-linked list that ends with the Nil element.
CHAPTER 30. THE LIST CLASS
The Vector class is an indexed, immutable sequence. The “indexed” part of the description means that you can access Vector elements very rapidly by their index value, such as accessing `list0fPeople(999999)`.

In general, except for the difference that Vector is indexed and List is not, the two classes work the same, so we’ll run through these examples quickly.

Here are a few ways you can create a Vector:

```scala
val nums = Vector(1, 2, 3, 4, 5)
val strings = Vector("one", "two")
val peeps = Vector(
  Person("Bert"),
  Person("Ernie"),
  Person("Grover")
)
```

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

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

you *append* elements like this:

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

and this:

```scala
val b = a ++ Vector(4, 5)
```
The REPL shows how this works:

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

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

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

You can also prepend elements like this:

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

and this:

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

Once again the REPL shows how this works:

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

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

Because `Vector` is not a linked-list (like `List`), you can prepend and append elements to it, and the speed of both approaches should be similar.

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

```scala
scala> val names = Vector("Joel", "Chris", "Ed")
val names: Vector[String] = Vector(Joel, Chris, Ed)

scala> for (name <- names) println(name)
Joel
Chris
Ed
```
The Map class documentation describes a Map as an iterable sequence that consists of pairs of keys and values. A simple Map looks like this:

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

Scala has both mutable and immutable Map classes. In this lesson we’ll show how to use the mutable class.

### 32.1 Creating a mutable Map

To use the mutable Map class, first import it:

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

Then you can create a Map like this:

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

### 32.2 Adding elements to a Map

Now you can add a single element to the Map with `+=`, like this:

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

You also add multiple elements using `+=`:

```scala
states += ("AR" -> "Arkansas", "AZ" -> "Arizona")
```
You can add elements from another Map using `++=`:

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

The REPL shows how these examples work:

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

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

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

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

### 32.3 Removing elements from a Map

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

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

The REPL shows how these examples work:

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

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

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

You update Map elements by reassigning their key to a new value:

\[ \text{states("AK") = "Alaska, A Really Big State"} \]

The REPL shows the current Map state:

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

32.5 Traversing a Map

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

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

a nice way to loop over all of the map elements is with this for loop syntax:

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

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

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

The ratings map data in this example comes from the old-but-good book, *Programming Collective Intelligence*. 
32.6 See also

There are other ways to work with Scala Maps, and a nice collection of Map classes for different needs. See the Map class documentation for more information and examples.
The Scala Set class is an iterable collection with no duplicate elements.

Scala has both mutable and immutable Set classes. In this lesson we’ll show how to use the mutable class.

### 33.1 Adding elements to a Set

To use a mutable Set, first import it:

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

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

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

The REPL shows how these examples work:

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

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

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

scala> set ++= Vector(4, 5)
val res2: scala.collection.mutable.Set[Int] = Set(1, 5, 2, 3, 4)
```
CHAPTER 33. THE SET CLASS

Notice that if you try to add a value to a set that’s already in it, the attempt is quietly ignored:

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

Set also has an `add` method that returns `true` if an element is added to a set, and `false` if it wasn’t added. The REPL shows how it works:

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

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

### 33.2 Deleting elements from a Set

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

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

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

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

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

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

```scala
scala> val set = scala.collection.mutable.Set(1, 2, 3, 4, 5)
```
33.3. **MORE SETS**

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

// clear
scala> set.clear()

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

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

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

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

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

### 33.3 More Sets

Scala has several more `Set` classes, including `SortedSet`, `LinkedHashSet`, and more. Please see the `Set` class documentation for more details on those classes.
34

Anonymous Functions

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

```scala
val ints = List(1,2,3)
```

When you want to create a larger list, you can also create them with the `List` class `range` method, like this:

```scala
val ints = List.range(1, 10)
```

That code creates `ints` as a list of integers whose values range from 1 to 10. You can see the result in the REPL:

```scala
scala> val ints = List.range(1, 10)
x: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

In this lesson we’ll use lists like these to demonstrate a feature of functional programming known as *anonymous functions*. It will help to understand how these work before we demonstrate the most common Scala collections methods.

34.1 Examples

An anonymous function is like a little mini-function. For example, given a list like this:

```scala
val ints = List(1,2,3)
```

You can create a new list by doubling each element in `ints`, like this:

```scala
val doubledInts = ints.map(_ * 2)
```

This is what that example looks like in the REPL:

```scala
scala> val doubledInts = ints.map(_ * 2)
```
doubledInts: List[Int] = List(2, 4, 6)

As that shows, `doubledInts` is now the list, `List(2, 4, 6)`. In this example, this code is an anonymous function:

`_ * 2`

This is a shorthand way of saying, “Multiply an element by 2.”

Once you’re comfortable with Scala, this is a common way to write anonymous functions, but if you prefer, you can also write them using longer forms. Besides writing that code like this:

```scala
val doubledInts = ints.map(_ * 2)
```

you can also write it like this:

```scala
val doubledInts = ints.map((i: Int) => i * 2)
val doubledInts = ints.map(i => i * 2)
```

All three lines have exactly the same meaning: Double each element in `ints` to create a new list, `doubledInts`.

The `_` character in Scala is something of a wildcard character. You’ll see it used in several different places. In this case it’s a shorthand way of saying, “An element from the list, `ints`.”

Before going any further, it’s worth mentioning that this `map` example is the equivalent of this Java code:

```java
List<Integer> ints = new ArrayList<>(Arrays.asList(1, 2, 3));

// the `map` process
List<Integer> doubledInts = ints.stream()
    .map(i -> i * 2)
    .collect(Collectors.toList());
```

The `map` example shown is also the same as this Scala code:

```scala
val doubledInts = for (i <- ints) yield i * 2
```
34.2 Anonymous functions with the \texttt{filter} method

Another good way to show anonymous functions is with the \texttt{filter} method of the \texttt{List} class. Given this List again:

\begin{verbatim}
val ints = List.range(1, 10)
\end{verbatim}

This is how you create a new list of all integers whose value is greater than 5:

\begin{verbatim}
val x = ints.filter(_ > 5)
\end{verbatim}

This is how you create a new list whose values are all less than 5:

\begin{verbatim}
val x = ints.filter(_ < 5)
\end{verbatim}

And as a little more complicated example, this is how you create a new list that contains only even values, by using the modulus operator:

\begin{verbatim}
val x = ints.filter(_ % 2 == 0)
\end{verbatim}

If that’s a little confusing, remember that this example can also be written in these other ways:

\begin{verbatim}
val x = ints.filter((i: Int) => i % 2 == 0)
val x = ints.filter(i => i % 2 == 0)
\end{verbatim}

This is what the previous examples look like in the REPL:

\begin{verbatim}
scala> val x = ints.filter(_ > 5)
x: List[Int] = List(6, 7, 8, 9)

scala> val x = ints.filter(_ < 5)
x: List[Int] = List(1, 2, 3, 4)

scala> val x = ints.filter(_ % 2 == 0)
x: List[Int] = List(2, 4, 6, 8)
\end{verbatim}

34.3 Key points

The key points of this lesson are:
• You can write anonymous functions as little snippets of code

• You can use them with methods on the List class like map and filter

• With these little snippets of code and powerful methods like those, you can create a lot of functionality with very little code

The Scala collections classes contain many methods like map and filter, and they’re a powerful way to create very expressive code.

34.4 Bonus: Digging a little deeper

You may be wondering how the map and filter examples work. The short answer is that when map is invoked on a list of integers — a List[Int] to be more precise — map expects to receive a function that transforms one Int value into another Int value. Because map expects a function (or method) that transforms one Int to another Int, this approach also works:

```scala
val ints = List(1, 2, 3)
def double(i: Int): Int = i * 2 // a method that doubles an Int
val doubledInts = ints.map(double)
```

The last two lines of that example are the same as this:

```scala
val doubledInts = ints.map(_ * 2)
```

Similarly, when called on a List[Int], the filter method expects to receive a function that takes an Int and returns a Boolean value. Therefore, given a method that’s defined like this:

```scala
def lessThanFive(i: Int): Boolean = if (i < 5) true else false
```

or more concisely, like this:

```scala
def lessThanFive(i: Int): Boolean = (i < 5)
```

this filter example:

```scala
val ints = List.range(1, 10)
val y = ints.filter(lessThanFive)
```
is the same as this example:

```scala
val y = ints.filter(_ < 5)
```
A great strength of the Scala collections classes is that they come with dozens of pre-built methods. The benefit of this is that you no longer need to write custom for loops every time you need to work on a collection. (If that’s not enough of a benefit, it also means that you no longer have to read custom for loops written by other developers.)

Because there are so many methods available to you, they won’t all be shown here. Instead, just some of the most commonly-used methods will be shown, including:

- map
- filter
- foreach
- head
- tail
- take, takeWhile
- drop, dropWhile
- find
- reduce, fold

The following methods will work on all of the collections “sequence” classes, including Array, ArrayBuffer, List, Vector, etc., but these examples will use a List unless otherwise specified.

35.1 Note: The methods don’t mutate the collection

As a very important note, none of these methods mutate the collection that they’re called on. They all work in a functional style, so they return a new collection with the modified results.
35.2 Sample lists

The following examples will use these lists:

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

This is what those lists look like in the REPL:

```scala
scala> val nums = (1 to 10).toList
nums: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> val names = List("joel", "ed", "chris", "maurice")
names: List[String] = List(joel, ed, chris, maurice)
```

35.3 map

The `map` method steps through each element in the existing list, applying the algorithm you supply to each element, one at a time; it then returns a new list with all of the modified elements.

Here's an example of the `map` method being applied to the `nums` list:

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

As we showed in the lesson on anonymous functions, you can also write the anonymous function like this:

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

However, in this lesson we'll always use the first, shorter form.

With that background, here's an example of the `map` method being applied to the `nums` and `names` lists:

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

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

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

As that last example shows, it's perfectly legal (and very common) to use map to return a list with a different type (List[Boolean]) from the original type (List[Int]).

35.4 filter

The filter method creates a new, filtered list from the given list. Here are a few examples:

scala> val lessThanFive = nums.filter(_ < 5)
lessThanFive: List[Int] = List(1, 2, 3, 4)

scala> val evens = nums.filter(_ % 2 == 0)
evens: List[Int] = List(2, 4, 6, 8, 10)

scala> val shortNames = names.filter(_.length <= 4)
shortNames: List[String] = List(joel, ed)

35.5 foreach

The foreach method is used to loop over all elements in a collection. As we mentioned in a previous lesson, foreach is used for side-effects, such as printing information. Here's an example with the names list:

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

The nums list is a little long, so you may not want to print out all of those elements. But a great thing about Scala's approach is that you can chain methods together to solve
problems like this. For example, this is one way to print the first three elements from nums:

```scala
nums.filter(_ < 4).foreach(println)
```

The REPL shows the result:

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

### 35.6 head

The `head` method comes from Lisp and functional programming languages. It's used to print the first element (the head element) of a list:

```scala
scala> nums.head
res0: Int = 1

scala> names.head
res1: String = joel
```

Because a `String` is a sequence of characters, you can also treat it like a list. This is how `head` works on these strings:

```scala
scala> "foo".head
res2: Char = f

scala> "bar".head
res3: Char = b
```

`head` is a great method to work with, but as a word of caution it can also throw an exception when called on an empty collection:

```scala
scala> val emptyList = List[Int]()
val emptyList: List[Int] = List()

scala> emptyList.head
```
35.7 tail

The tail method also comes from Lisp and functional programming languages. It's used to print every element in a list after the head element. A few examples:

```scala
cscala> nums.tail
res0: List[Int] = List(2, 3, 4, 5, 6, 7, 8, 9, 10)

cscala> names.tail
res1: List[String] = List(ed, chris, maurice)
```

Just like head, tail also works on strings:

```scala
cscala> "foo".tail
res2: String = oo

cscala> "bar".tail
res3: String = ar
```

Note that like head, tail will also throw an exception when called on an empty collection:

```scala
cscala> emptyList.tail
java.lang.UnsupportedOperationException: tail of empty list
```

35.8 take, takeWhile

The take and takeWhile methods give you a nice way of taking the elements out of a list that you want to create a new list. This is take:

```scala
cscala> nums.take(1)
res0: List[Int] = List(1)

cscala> nums.take(2)
res1: List[Int] = List(1, 2)
```
And this is `takeWhile`:

```scala
cscala> nums.takeWhile(_ < 5)
res4: List[Int] = List(1, 2, 3, 4)
cscala> names.takeWhile(_.length < 5)
res5: List[String] = List(joel, ed)
```

### 35.9 `drop`, `dropWhile`

`drop` and `dropWhile` are essentially the opposite of `take` and `takeWhile`. This is `drop`:

```scala
cscala> nums.drop(1)
res0: List[Int] = List(2, 3, 4, 5, 6, 7, 8, 9, 10)
cscala> nums.drop(5)
res1: List[Int] = List(6, 7, 8, 9, 10)
cscala> names.drop(1)
res2: List[String] = List(ed, chris, maurice)
cscala> names.drop(2)
res3: List[String] = List(chris, maurice)
```

And this is `dropWhile`:

```scala
cscala> nums.dropWhile(_ < 5)
res4: List[Int] = List(5, 6, 7, 8, 9, 10)
cscala> names.dropWhile(_ != "chris")
res5: List[String] = List(chris, maurice)
```
35.10  reduce

When you hear the term, “map reduce,” the “reduce” part refers to methods like reduce. It takes a function (or anonymous function) and applies that function to successive elements in the list.

The best way to explain reduce is to create a little helper method you can pass into it. For example, this is an add method that adds two integers together, and also gives us some nice debug output:

```scala
def add(x: Int, y: Int): Int = {
  val theSum = x + y
  println(s"received $x and $y, their sum is $theSum")
  theSum
}
```

Now, given that method and this list:

```scala
val a = List(1,2,3,4)
```

this is what happens when you pass the add method into reduce:

```scala
scala> a.reduce(add)
received 1 and 2, their sum is 3
received 3 and 3, their sum is 6
received 6 and 4, their sum is 10
res0: Int = 10
```

As that result shows, reduce uses add to reduce the list a into a single value, in this case, the sum of the integers in the list.

Once you get used to reduce, you’ll write a “sum” algorithm like this:

```scala
scala> a.reduce(_ + _)
res0: Int = 10
```

Similarly, this is what a “product” algorithm looks like:

```scala
scala> a.reduce(_ * _)
res1: Int = 24
```
That might be a little mind-blowing if you've never seen it before, but after a while you'll get used to it.

Before moving on, an important part to know about reduce is that — as its name implies — it's used to reduce a collection down to a single value.

35.11 Even more!

There are literally dozens of additional methods on the Scala sequence classes that will keep you from ever needing to write another for loop. However, because this is a simple introduction book they won't all be covered here. For more information, see the collections overview of sequence traits.
Common Map Methods

In this lesson we’ll demonstrate some of the most commonly used `Map` methods. In these initial examples we’ll use an immutable `Map`, and Scala also has a mutable `Map` class that you can modify in place, and it’s demonstrated a little later in this lesson.

For these examples we won’t break the `Map` methods down into individual sections; we’ll just provide a brief comment before each method.

Given this immutable `Map`:

```scala
val m = Map(
    1 -> "a",
    2 -> "b",
    3 -> "c",
    4 -> "d"
)
```

Here are some examples of methods available to that `Map`:

// how to iterate over Map elements
scala> for ((k,v) <- m) printf("key: %s, value: %s\n", k, v)
key: 1, value: a
key: 2, value: b
key: 3, value: c
key: 4, value: d

// how to get the keys from a Map
scala> val keys = m.keys
keys: Iterable[Int] = Set(1, 2, 3, 4)

// how to get the values from a Map
scala> val values = m.values
val values: Iterable[String] = MapLike.DefaultValuesIterable(a, b, c, d)
// how to test if a Map contains a value
scala> val contains3 = m.contains(3)
contains3: Boolean = true

// how to transform Map values
scala> val ucMap = m.transform((k,v) => v.toUpperCase)

// how to filter a Map by its keys
scala> val twoAndThree = m.view.filterKeys(Set(2,3)).toMap
twoAndThree: scala.collection.immutable.Map[Int,String] = Map(2 -> b, 3 -> c)

// how to take the first two elements from a Map
scala> val firstTwoElements = m.take(2)
firstTwoElements: scala.collection.immutable.Map[Int,String] = Map(1 -> a, 2 -> b)

Note that the last example probably only makes sense for a sorted Map.

### 36.1 Mutable Map examples

Here are a few examples of methods that are available on the mutable Map class. Given this initial mutable Map:

```scala
define states = scala.collection.mutable.Map(
  "AL" -> "Alabama",
  "AK" -> "Alaska"
)
```

Here are some things you can do with a mutable Map:

```scala
// add elements with +=
states += ("AZ" -> "Arizona")
states += ("CO" -> "Colorado", "KY" -> "Kentucky")

// remove elements with -=
states -= "KY"
states -= ("AZ", "CO")
```
// update elements by reassigning them
states("AK") = "Alaska, The Big State"

// retain elements by supplying a function that operates on
// the keys and/or values
states.retain((k,v) => k == "AK")

36.2 See also

There are many more things you can do with maps. See the Map class documentation for more details and examples.
A Few Miscellaneous Items

In this section we’ll cover a few miscellaneous items about Scala:

- Tuples
- A Scala OOP example of a pizza restaurant order-entry system
A *tuple* is a neat class that gives you a simple way to store heterogeneous (different) items in the same container. For example, assuming that you have a class like this:

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

Instead of having to create an ad-hoc class to store things in, like this:

```scala
class SomeThings(i: Int, s: String, p: Person)
```

you can just create a tuple like this:

```scala
val t = (3, "Three", new Person("Al"))
```

As shown, just put some elements inside parentheses, and you have a tuple. Scala tuples can contain between two and 22 items, and they’re useful for those times when you just need to combine a few things together, and don’t want the baggage of having to define a class, especially when that class feels a little “artificial” or phony.

Technically, Scala 2.x has classes named `Tuple2`, `Tuple3` ... up to `Tuple22`. As a practical matter you rarely need to know this, but it’s also good to know what’s going on under the hood. (And this architecture is being improved in Scala 3.)

### 38.1 A few more tuple details

Here’s a two-element tuple:

```scala
scala> val d = ("Maggie", 30)
d: (String, Int) = (Maggie,30)
```

Notice that it contains two different types. Here’s a three-element tuple:
There are a few ways to access tuple elements. One approach is to access them by element number, where the number is preceded by an underscore:

```
scala> t._1
res1: Int = 3
```

```
scala> t._2
res2: java.lang.String = Three
```

```
scala> t._3
res3: Person = Person(David)
```

Another cool approach is to access them like this:

```
scala> val(x, y, z) = (3, "Three", new Person("David"))
x: Int = 3
y: String = Three
z: Person = Person(David)
```

Technically this approach involves a form of pattern-matching, and it’s a great way to assign tuple elements to variables.

### 38.2 Returning a tuple from a method

A place where this is nice is when you want to return multiple values from a method. For example, here’s a method that returns a tuple:

```
def getStockInfo = {
  // other code here ...
  ("NFLX", 100.00, 101.00) // this is a Tuple3
}
```

Now you can call that method and assign variable names to the return values:
38.3. TUPLES AREN’T COLLECTIONS

val (symbol, currentPrice, bidPrice) = getStockInfo

The REPL demonstrates how this works:

scala> val (symbol, currentPrice, bidPrice) = getStockInfo
symbol: String = NFLX
currentPrice: Double = 100.0
bidPrice: Double = 101.0

For cases like this where it feels like overkill to create a class for the method’s return type, a tuple is very convenient.

38.3 Tuples aren’t collections

Technically, Scala 2.x tuples aren’t collections classes, they’re just a convenient little container. Because they aren’t a collection, they don’t have methods like `map`, `filter`, etc.
An OOP Example

This lesson shares an example of an OOP application written with Scala. The example shows code you might write for an order-entry system for a pizza store.

As shown earlier in the book, you create enumerations in Scala like this:

```scala
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
```

A nice thing about Scala is that even though we haven’t discussed sealed traits or case objects, you can probably still figure out how this code works.

39.1 A few classes

Given those enumerations, you can now start to create a few pizza-related classes for an order-entry system. First, here’s a Pizza class:

```scala
import scala.collection.mutable.ArrayBuffer
```
class Pizza {
    var crustSize: CrustSize,
    var crustType: CrustType,
    var toppings: ArrayBuffer[Topping]
}

Next, here's an Order class, where an Order consists of a mutable list of pizzas and a Customer:

class Order {
    var pizzas: ArrayBuffer[Pizza],
    var customer: Customer
}

Here's a Customer class to work with that code:

class Customer {
    var name: String,
    var phone: String,
    var address: Address
}

Finally, here's an Address class:

class Address {
    var street1: String,
    var street2: String,
    var city: String,
    var state: String,
    var zipCode: String
}

So far those classes just look like data structures — like a struct in C — so let's add a little behavior.
39.2 Adding behavior to Pizza

For the most part an OOP Pizza class needs a few methods to add and remove toppings, and adjust the crust size and type. Here's a Pizza class with a few added methods to handle those behaviors:

```scala
class Pizza {
  var crustSize: CrustSize,
  var crustType: CrustType,
  val toppings: ArrayBuffer[Topping]

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

You can also argue that a pizza should be able to calculate its own price, so here's another method you could add to that class:

```scala
def getPrice(
  toppingsPrices: Map[Topping, Int],
  crustSizePrices: Map[CrustSize, Int],
  crustTypePrices: Map[CrustType, Int]
): Int = ???
```

Note that this is a perfectly legal method. The `??` syntax is often used as a teaching tool, and sometimes you use it as a method-sketching tool to say, “This is what my method signature looks like, but I don’t want to write the method body yet.” A great thing for those times is that this code compiles.

That being said, don’t call that method. If you do, you’ll get a `NotImplementedError`, which is very descriptive of the situation.

39.3 Adding behavior to Order

You should be able to do a few things with an order, including:

- Add and remove pizzas
• Update customer information

• Get the order price

Here's an `Order` class that lets you do those things:

class Order (  
  val pizzas: ArrayBuffer[Pizza],  
  var customer: Customer
) {

  def addPizza(p: Pizza): Unit = pizzas += p
  def removePizza(p: Pizza): Unit = pizzas -= p

  // need to implement these
  def getBasePrice(): Int = ???
  def getTaxes(): Int = ???
  def getTotalPrice(): Int = ???

}

Once again, for the purposes of this example, we're not concerned with how to calculate the price of an order.

### 39.4 Testing those classes

You can use a little “driver” class to test those classes. With the addition of a `printOrder` method on the `Order` class and a `toString` method in the `Pizza` class, you'll find that the code shown works as advertised:

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

object MainDriver extends App {

  val p1 = new Pizza (  
    MediumCrustSize,  
    ThinCrustType,  
    ArrayBuffer(Cheese)
  )
```
val p2 = new Pizza (
  LargeCrustSize,
  ThinCrustType,
  ArrayBuffer(Cheese, Pepperoni, Sausage)
)

val address = new Address (
  "123 Main Street",
  "Apt. 1",
  "Talkeetna",
  "Alaska",
  "99676"
)

val customer = new Customer (
  "Alvin Alexander",
  "907-555-1212",
  address
)

val o = new Order(
  ArrayBuffer(p1, p2),
  customer
)

o.addPizza(
  new Pizza (
    SmallCrustSize,
    ThinCrustType,
    ArrayBuffer(Cheese, Mushrooms)
  )
)

// print the order
o.printOrder
39.5 Experiment with the code yourself

To experiment with this on your own, please see the *PizzaOopExample* project in this book’s GitHub repository, which you can find at this URL:

- github.com/alvinj/HelloScalaExamples

To compile this project it will help to either (a) use IntelliJ IDEA or Eclipse, or (b) know how to use the Scala Build Tool.
In the next few lessons you’ll see a couple of tools that are commonly used in Scala projects:

- The SBT build tool
- ScalaTest, a code testing framework

We’ll start by showing how to use SBT, and then you’ll see how to use ScalaTest and SBT together to build and test your Scala projects.
The Scala Build Tool (SBT)

You can use several different tools to build your Scala projects, including Ant, Maven, Gradle, and more. But a tool named SBT was the first build tool that was specifically created for Scala, and these days it’s supported by Lightbend, the company that was co-founded by Scala creator Martin Odersky that also maintains Akka, the Play web framework, and more.

If you haven’t already installed SBT, here’s a link to its download page.

41.1 The SBT directory structure

Like Maven, SBT uses a standard project directory structure. If you use that standard directory structure you’ll find that it’s relatively simple to build your first projects.

The first thing to know is that underneath your main project directory, SBT expects a directory structure that looks like this:

```
build.sbt
project/
src/
  -- main/
    |-- java/
    |-- resources/
    |-- scala/
  |-- test/
    |-- java/
    |-- resources/
    |-- scala/
target/
```
41.2 Creating a “Hello, world” SBT project directory structure

Creating this directory structure is pretty simple, and you can use a shell script like `sbtmkdirs` to create new projects. But you don't have to use that script; assuming that you're using a Unix/Linux system, you can just use these commands to create your first SBT project directory structure:

```bash
mkdir HelloWorld
cd HelloWorld
mkdir -p src/{main,test}/{java,resources,scala}
mkdir project target
```

If you run a `find .` command after running those commands, you should see this result:

```
$ find .
.
./project
./src
./src/main
./src/main/java
./src/main/resources
./src/main/scala
./src/test
./src/test/java
./src/test/resources
./src/test/scala
./target
```

If you see that, you're in great shape for the next step.

There are other ways to create the files and directories for an SBT project. One way is to use the `sbt new` command, which is documented here on scala-sbt.org. That approach isn't shown here because some of the files it creates are more complicated than necessary for an introduction like this.

41.3 Creating a first `build.sbt` file

At this point you only need two more things to run a “Hello, world” project:
41.4. A “HELLO, WORLD” PROGRAM

- A `build.sbt` file
- A `Hello.scala` file

For a little project like this, the `build.sbt` file only needs to contain a few lines, like this:

```scala
name := "HelloWorld"
version := "1.0"
scalaVersion := "{{ site.scala-version }}"
```

Because SBT projects use a standard directory structure, SBT already knows everything else it needs to know.

Now you just need to add a little “Hello, world” program.

41.4 A “Hello, world” program

In large projects, all of your Scala source code files will go under the `src/main/scala` and `src/test/scala` directories, but for a little sample project like this, you can put your source code file in the root directory of your project. Therefore, create a file named `HelloWorld.scala` in the root directory with these contents:

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

Now you can use SBT to compile your project, where in this example, your project consists of that one file. Use the `sbt run` command to compile and run your project. When you do so, you’ll see output that looks like this:

```
$ sbt run
```

```
Updated file /Users/al/Projects/Scala/Hello/project/build.properties setting sbt.version to: 0.13.15
[warn] Executing in batch mode.
[warn] For better performance, hit [ENTER] to switch to interactive mode, or
[warn] consider launching sbt without any commands, or explicitly passing 'shell'
[info] Loading project definition from /Users/al/Projects/Scala/Hello/project
[info] Updating {file:/Users/al/Projects/Scala/Hello/project/}hello-build...
[info] Resolving org.fusesource.jansi#jansi;1.4 ...
[info] Done updating.
```
The first time you run sbt it needs to download some things and can take a while to run, but after that it gets much faster. As the first comment in that output shows, it’s also faster to run SBT interactively. To do that, first run the sbt command by itself:

> sbt

The execute its run command like this:

> run

There, that’s much faster.

If you type help at the SBT command prompt you’ll see a bunch of other commands you can run. But for now, just type exit to leave the SBT shell. You can also press CTRL-D instead of typing exit.

41.5 See also

Here’s a list of other build tools you can use to build Scala projects are:

- Ant
- Gradle
- Maven
- Fury
41.5. SEE ALSO

- Mill
ScalaTest is one of the main testing libraries for Scala projects, and in this lesson you’ll see how to create a Scala project that uses ScalaTest. You’ll also be able to compile, test, and run the project with SBT.

42.1 Creating the project directory structure

As with the previous lesson, create an SBT project directory structure for a project named HelloScalaTest with the following commands:

```bash
mkdir HelloScalaTest
cd HelloScalaTest
mkdir -p src/{main,test}/{java,resources,scala}
mkdir lib project target
```

42.2 Creating the `build.sbt` file

Next, create a `build.sbt` file in the root directory of your project with these contents:

```scala
name := "HelloScalaTest"
version := "1.0"
scalaVersion := "{{site.scala-version}}"

libraryDependencies ++= Seq(
  "org.scalactic" %% "scalactic" % "3.0.8",
  "org.scalatest" %% "scalatest" % "3.0.8" % "test"
)
```

The first three lines of this file are essentially the same as the first example, and the `libraryDependencies` lines tell SBT to include the dependencies (jar files) that are needed to run ScalaTest:
libraryDependencies ++= Seq(
  "org.scalactic" %% "scalactic" % "3.0.4",
  "org.scalatest" %% "scalatest" % "3.0.4" % "test"
)

The ScalaTest documentation has always been good, and you can always find the up to date information on what those lines should look like on the Installing ScalaTest page.

42.3 Create a Scala file

Next, create a Scala program that you can use to demonstrate ScalaTest. First, from the root directory of your project, create a directory under `src/main/scala` named `simpletest`:

$ mkdir src/main/scala/simpletest

Then, inside that directory, create a file named `Hello.scala` with these contents:

```scala
package simpletest

object Hello extends App {
  val p = new Person("Alvin Alexander")
  println(s"Hello ${p.name}")
}

class Person(var name: String)
```

There isn’t much that can go wrong with that source code, but it provides a simple way to demonstrate ScalaTest. At this point you can run your project with the `sbt run` command, where your output should look like this:

> sbt run

[warn] Executing in batch mode.
[warn] For better performance, hit [ENTER] to switch to interactive mode, or [warn] consider launching sbt without any commands, or explicitly passing 'shell'
...
[info] Compiling 1 Scala source to /Users/al/Projects/Scala/HelloScalaTest/target/scala-2.12/classes...
[info] Running simpletest.Hello
Hello Alvin Alexander
[success] Total time: 4 s

Now let's create a ScalaTest file.

42.4 Your first ScalaTest tests

ScalaTest is very flexible, and there are a lot of different ways to write tests, but a simple way to get started is to write tests using the ScalaTest “FunSuite.” To get started, create a directory named simpletest under the src/test/scala directory, like this:

$ mkdir src/test/scala/simpletest

Next, create a file named HelloTests.scala in that directory with the following contents:

```
package simpletest

import org.scalatest.FunSuite

class HelloTests extends FunSuite {

  // test 1
  test("the name is set correctly in constructor") {
    val p = new Person("Barney Rubble")
    assert(p.name == "Barney Rubble")
  }

  // test 2
  test("a Person's name can be changed") {
    val p = new Person("Chad Johnson")
    p.name = "Ochocinco"
    assert(p.name == "Ochocinco")
  }

}
```

This file demonstrates the ScalaTest FunSuite approach. A few important points:
• Your class should extend FunSuite
• You create tests as shown, by giving each test a unique name
• At the end of each test you should call assert to test that a condition has been satisfied

Using ScalaTest like this is similar to JUnit, so if you’re coming to Scala from Java, hopefully this looks very familiar.

Now you can run these tests with the sbt test command. Skipping the first few lines of output, the result looks like this:

> sbt test
[info] Set current project to HelloScalaTest (in build file:/Users/al/Projects/Scala/HelloScalaTest/)
[info] HelloTests:
[info] - the name is set correctly in constructor
[info] - a Person's name can be changed
[info] Run completed in 277 milliseconds.
[info] Total number of tests run: 2
[info] Suites: completed 1, aborted 0
[info] Tests: succeeded 2, failed 0, canceled 0, ignored 0, pending 0
[info] All tests passed.
[success] Total time: 1 s

42.5 TDD tests

This example demonstrates a Test-Driven Development (TDD) style of testing with ScalaTest. In the next lesson you’ll see how to write Behavior-Driven Development (BDD) tests with ScalaTest and SBT.

Keep the project you just created. You’ll use it again in the next lesson.
43

Writing BDD Style Tests with ScalaTest and SBT

In the previous lesson you saw how to write Test-Driven Development (TDD) tests with ScalaTest. ScalaTest also supports a Behavior-Driven Development (BDD) style of testing, which we’ll demonstrate next.

This lesson uses the same SBT project as the previous lesson, so you don’t have to go through the initial setup work again.

43.1 Creating a Scala class to test

First, create a new Scala class to test. In the src/main/scala/simpletest, create a new file named MathUtils.scala with these contents:

```scala
class MathUtils {
  def double(i: Int) = i * 2
}
```

The BDD tests you’ll write next will test the `double` method in that class.

43.2 Creating ScalaTest BDD-style tests

Next, create a file named `MathUtilsTests.scala` in the `src/test/scala/simpletest` directory, and put these contents in that file:

```scala
package simpletest
```
import org.scalatest.FunSpec

class MathUtilsSpec extends FunSpec {

  describe("MathUtils::double") {

    it("should handle 0 as input") {
      val result = MathUtils.double(0)
      assert(result == 0)
    }

    it("should handle 1") {
      val result = MathUtils.double(1)
      assert(result == 2)
    }

    it("should handle really large integers") (pending)

  }

}

As you can see, this is a very different-looking style than the TDD tests in the previous lesson. If you’ve never used a BDD style of testing before, a main idea is that the tests should be relatively easy to read for one of the “domain experts” who work with the programmers to create the application. A few notes about this code:

- It uses the FunSpec class where the TDD tests used FunSuite
- A set of tests begins with describe
- Each test begins with it. The idea is that the test should read like, “It should do XYZ…,” where “it” is the double function
- This example also shows how to mark a test as “pending”

43.3 Running the tests

With those files in place you can again run sbt test. The important part of the output looks like this:
> sbt test

[info] HelloTests:
[info] - the name is set correctly in constructor
[info] - a Person's name can be changed
[info] MathUtilsSpec:
[info] MathUtils::double
[info] - should handle 0 as input
[info] - should handle 1
[info] - should handle really large integers (pending)
[info] Total number of tests run: 4
[info] Suites: completed 2, aborted 0
[info] Tests: succeeded 4, failed 0, canceled 0, ignored 0, pending 1
[info] All tests passed.
[success] Total time: 4 s, completed Jan 6, 2018 4:58:23 PM

A few notes about that output:

- `sbt test` ran the previous `HelloTests` as well as the new `MathUtilsSpec` tests
- The pending test shows up in the output and is marked “(pending)"
- All of the tests passed

If you want to have a little fun with this, change one or more of the tests so they intentionally fail, and then see what the output looks like.

43.4 Where to go from here

For more information about SBT and ScalaTest, see the following resources:

- The main SBT documentation
- The ScalaTest documentation
CHAPTER 43. WRITING BDD STYLE TESTS WITH SCALATEST AND SBT
Functional Programming

Scala lets you write code in an object-oriented programming (OOP) style, a functional programming (FP) style, and even in a hybrid style, using both approaches in combination. This book assumes that you’re coming to Scala from an OOP language like Java, C++, or C#, so outside of covering Scala classes, there aren’t any special sections about OOP in this book. But because the FP style is still relatively new to many developers, we’ll provide a brief introduction to Scala’s support for FP in the next several lessons.

Functional programming is a style of programming that emphasizes writing applications using only pure functions and immutable values. As Alvin Alexander wrote in Functional Programming, Simplified, rather than using that description, it can be helpful to say that functional programmers have an extremely strong desire to see their code as math — to see the combination of their functions as a series of algebraic equations. In that regard, you could say that functional programmers like to think of themselves as mathematicians. That’s the driving desire that leads them to use only pure functions and immutable values, because that’s what you use in algebra and other forms of math.

Functional programming is a large topic, and there’s no simple way to condense the entire topic into this little book, but in the following lessons we’ll give you a taste of FP, and show some of the tools Scala provides for developers to write functional code.
A first feature Scala offers to help you write functional code is the ability to write pure functions. In *Functional Programming, Simplified*, Alvin Alexander defines a *pure function* like this:

- The function’s output depends *only* on its input variables
- It doesn’t mutate any hidden state
- It doesn’t have any “back doors”: It doesn’t read data from the outside world (including the console, web services, databases, files, etc.), or write data to the outside world

As a result of this definition, any time you call a pure function with the same input value(s), you’ll always get the same result. For example, you can call a `double` function an infinite number of times with the input value 2, and you’ll always get the result 4.

### 45.1 Examples of pure functions

Given that definition of pure functions, as you might imagine, methods like these in the `scala.math._` package are pure functions:

- `abs`
- `ceil`
- `max`
- `min`

These Scala String methods are also pure functions:

- `isEmpty`
- `length`
- `substring`
Many methods on the Scala collections classes also work as pure functions, including `drop`, `filter`, and `map`.

### 45.2 Examples of impure functions

Conversely, the following functions are *impure* because they violate the definition.

The `foreach` method on collections classes is impure because it’s only used for its side effects, such as printing to STDOUT.

A great hint that `foreach` is impure is that its method signature declares that it returns the type `Unit`. Because it returns nothing, logically the only reason you ever call it is to achieve some side effect. Similarly, *any* method that returns `Unit` is going to be an impure function.

Date and time related methods like `getDayOfWeek`, `getHour`, and `getMinute` are all impure because their output depends on something other than their input parameters. Their results rely on some form of hidden I/O, *hidden input* in these examples.

In general, impure functions do one or more of these things:

- Read hidden inputs, i.e., they access variables and data not explicitly passed into the function as input parameters
- Write hidden outputs
- Mutate the parameters they are given
- Perform some sort of I/O with the outside world

### 45.3 But impure functions are needed …

Of course an application isn’t very useful if it can’t read or write to the outside world, so people make this recommendation:

Write the core of your application using pure functions, and then write an impure “wrapper” around that core to interact with the outside world. If you like food analogies, this is like putting a layer of impure icing on top of a pure cake.
There are ways to make impure interactions with the outside world feel a little more pure. For instance, you’ll hear about things like the IO Monad for dealing with user input, files, networks, and databases. But in the end, FP applications have a core of pure functions combined with other functions to interact with the outside world.

### 45.4 Writing pure functions

Writing pure functions in Scala is one of the simpler parts about functional programming: You just write pure functions using Scala’s method syntax. Here’s a pure function that doubles the input value it’s given:

``` scala
def double(i: Int): Int = i * 2
```

Although recursion isn’t covered in this book, if you like a good “challenge” example, here’s a pure function that calculates the sum of a list of integers (`List[Int]`):

``` scala
def sum(list: List[Int]): Int = list match {
  case Nil => 0
  case head :: tail => head + sum(tail)
}
```

Even though we haven’t covered recursion, if you can understand that code, you’ll see that it meets my definition of a pure function.

### 45.5 Key points

The first key point of this lesson is the definition of a pure function:

> A pure function is a function that depends only on its declared inputs and its internal algorithm to produce its output. It does not read any other values from “the outside world” — the world outside of the function’s scope — and it does not modify any values in the outside world.

A second key point is that real-world applications consist of a combination of pure and impure functions. A common recommendation is to write the core of your application using pure functions, and then to use impure functions to communicate with the outside world.
While every programming language ever created probably lets you write pure functions, a second great FP feature of Scala is that *you can create functions as variables*, just like you create `String` and `Int` variables. This feature has many benefits, the most common of which is that it lets you pass functions as parameters into other functions. You saw that earlier in this book when the `map` and `filter` methods were demonstrated:

```scala
val nums = (1 to 10).toList

val doubles = nums.map(_ * 2)
val lessThanFive = nums.filter(_ < 5)
```

In those examples, anonymous functions are passed into `map` and `filter`. In the lesson on anonymous functions we demonstrated that this example:

```scala
val doubles = nums.map(_ * 2)
```

is the same as passing a regular function into `map`:

```scala
def double(i: Int): Int = i * 2 //a method that doubles an Int
def doubles = nums.map(double)
```

As those examples show, Scala clearly lets you pass anonymous functions and regular functions into other methods. This is a powerful feature that good FP languages provide.

If you like technical terms, a function that takes another function as an input parameter is known as a *Higher-Order Function* (HOF). (And if you like humor, as someone once wrote, that’s like saying that a class that takes an instance of another class as a constructor parameter is a Higher-Order Class.)
46.1 Function or method?

Scala has a special “function” syntax, but as a practical matter the def syntax seems to be preferred. This may be because of two reasons:

- The def syntax is more familiar to people coming from a C/Java/C# background
- You can use def methods just like they are val functions

What that second statement means is that when you define a method with def like this:

```scala
def double(i: Int): Int = i * 2
```

you can then pass `double` around as if it were a variable, like this:

```scala
val x = ints.map(double)
```

Even though `double` is defined as a method, Scala lets you treat it as a function.

The ability to pass functions around as variables is a distinguishing feature of functional programming languages. And as you’ve seen in map and filter examples in this book, the ability to pass functions as parameters into other functions helps you create code that is concise and still readable.

46.2 A few examples

If you’re not comfortable with the process of passing functions as parameters into other functions, here are a few more examples you can experiment with in the REPL:

```scala
List("foo", "bar").map(_.toUpperCase)
List("foo", "bar").map(_.capitalize)
List("adam", "scott").map(_.length)
List(1,2,3,4,5).map(_ * 10)
List(1,2,3,4,5).filter(_ > 2)
List(5,1,3,11,7).takeWhile(_ < 6)
```

Remember that any of those anonymous functions can also be written as “regular” functions, so you can write a function like this:
def toUpper(s: String): String = s.toUpperCase

and then pass it into map like this:

List("foo", "bar").map(toUpper)

or this:

List("foo", "bar").map(s => toUpper(s))

Those examples that use a “regular” function are equivalent to these anonymous function examples:

List("foo", "bar").map(s => s.toUpperCase)
List("foo", "bar").map(_.toUpperCase)
Functional programming is like writing a series of algebraic equations, and because you don’t use null values in algebra, you don’t use null values in FP. That brings up an interesting question: In the situations where you might normally use a null value in Java/OOP code, what do you do?

Scala’s solution is to use constructs like the Option/Some/None classes. We’ll provide an introduction to the techniques in this lesson.

### 47.1 A first example

While this first Option/Some/None example doesn’t deal with null values, it’s a good way to demonstrate the Option/Some/None classes, so we’ll start with it.

Imagine that you want to write a method to make it easy to convert strings to integer values, and you want an elegant way to handle the exceptions that can be thrown when your method gets a string like "foo" instead of something that converts to a number, like "1". A first guess at such a function might look like this:

```scala
def toInt(s: String): Int = {
  try {
    Integer.parseInt(s.trim)
  } catch {
    case e: Exception => 0
  }
}
```

The idea of this function is that if a string converts to an integer, you return the converted Int, but if the conversion fails you return 0. This might be okay for some purposes, but it’s not really accurate. For instance, the method might have received "0", but it may have also received "foo" or "bar" or an infinite number of other strings. This creates a real problem: How do you know when the method really received a "0",
or when it received something else? The answer is that with this approach, there's no way to know.

### 47.2 Using Option/Some/None

Scala's solution to this problem is to use a trio of classes known as `Option`, `Some`, and `None`. The `Some` and `None` classes are subclasses of `Option`, so the solution works like this:

- You declare that `toInt` returns an `Option` type
- If `toInt` receives a string it *can* convert to an `Int`, you wrap the `Int` inside of a `Some`
- If `toInt` receives a string it *can't* convert, it returns a `None`

The implementation of the solution looks like this:

```scala
def toInt(s: String): Option[Int] = {
  try {
    Some(Integer.parseInt(s.trim))
  } catch {
    case e: Exception => None
  }
}
```

This code can be read as, “When the given string converts to an integer, return the integer wrapped in a `Some` wrapper, such as `Some(1)`. When the string can't be converted to an integer, return a `None` value.”

Here are two REPL examples that demonstrate `toInt` in action:

```scala
scala> val a = toInt("1")
a: Option[Int] = Some(1)

scala> val a = toInt("foo")
a: Option[Int] = None
```

As shown, the string "1" converts to `Some(1)`, and the string "foo" converts to `None`. This is the essence of the Option/Some/None approach. It's used to handle exceptions
47.3. BEING A CONSUMER OF TOINT

(as in this example), and the same technique works for handling null values.

You’ll find this approach used throughout Scala library classes, and in third-party Scala libraries.

47.3 Being a consumer of toInt

Now imagine that you’re a consumer of the toInt method. You know that the method returns a subclass of Option[Int], so the question becomes, how do you work with these return types?

There are two main answers, depending on your needs:

- Use a match expression
- Use a for-expression

There are other approaches, but these are the two main approaches, especially from an FP standpoint.

47.3.1 Using a match expression

One possibility is to use a match expression, which looks like this:

```scala
toInt(x) match {
  case Some(i) => println(i)
  case None => println("That didn't work.")
}
```

In this example, if x can be converted to an Int, the first case statement is executed; if x can’t be converted to an Int, the second case statement is executed.

47.3.2 Using for/yield

Another common solution is to use a for-expression — i.e., the for/yield combination that was shown earlier in this book. To demonstrate this, imagine that you want to convert three strings to integer values, and then add them together. The for/yield solution looks like this:
val y = for {
  a <- toInt(stringA)
  b <- toInt(stringB)
  c <- toInt(stringC)
} yield a + b + c

When that expression finishes running, \( y \) will be one of two things:

- If all three strings convert to integers, \( y \) will be a `Some[Int]`, i.e., an integer wrapped inside a `Some`
- If any of the three strings can’t be converted to an inside, \( y \) will be a `None`

You can test this for yourself in the Scala REPL. First, paste these three string variables into the REPL:

```scala
val stringA = "1"
val stringB = "2"
val stringC = "3"
```

Next, paste the for-expression into the REPL. When you do that, you’ll see this result:

```scala
scala> val y = for {
               |   a <- toInt(stringA)
               |   b <- toInt(stringB)
               |   c <- toInt(stringC)
               | } yield a + b + c
y: Option[Int] = Some(6)
```

As shown, \( y \) is bound to the value `Some(6)`.

To see the failure case, change any of those strings to something that won’t convert to an integer. When you do that, you’ll see that \( y \) is a `None`:

```scala
y: Option[Int] = None
```

47.4 Options can be thought of as a container of 0 or 1 items

One good way to think about the `Option` classes is that they represent a container, more specifically a container that has either zero or one item inside:
• Some is a container with one item in it
• None is a container, but it has nothing in it

If you prefer to think of the Option classes as being like a box, None is a little like getting an empty box for a birthday gift.

47.5 Using foreach

Because Some and None can be thought of containers, they can be further thought of as being like collections classes. As a result, they have all of the methods you’d expect from a collection class, including map, filter, foreach, etc.

This raises an interesting question: What will these two values print, if anything?

toInt("1").foreach(println)
toInt("x").foreach(println)

The answer is that the first example prints the number 1, and the second example doesn’t print anything. The first example prints 1 because:

• toInt(“1”) evaluates to Some(1)
• The expression evaluates to Some(1).foreach(println)
• The foreach method on the Some class knows how to reach inside the Some container and extract the value (1) that’s inside it, so it passes that value to println

Similarly, the second example prints nothing because:

• toInt("x") evaluates to None
• The foreach method on the None class knows that None doesn’t contain anything, so it does nothing

Again, None is just an empty container.

Somewhere in Scala’s history, someone noted that the first example (the Some) represents the “Happy Path” of Option/Some/None approach, and the second example (the None) represents the “Unhappy Path.” But, despite having two different possible outcomes, the cool thing about the approach is that the code you write to handle an Option looks exactly the same in both cases. The foreach examples look like this:
toInt("1").foreach(println)
toint("x").foreach(println)

And the for-expression looks like this:

```scala
val y = for {
  a <- toInt(stringA)
  b <- toInt(stringB)
  c <- toInt(stringC)
} yield a + b + c
```

You only have to write one piece of code to handle both the Happy and Unhappy Paths, and that simplifies your code. The only time you have to think about whether you got a Some or a None is when you finally handle the result value in a match expression, like this:

```scala
toint(x) match {
  case Some(i) => println(i)
  case None => println("That didn't work.")
}
```

### 47.6 Using Option to replace null values

Another place where a null value can silently creep into your code is with a class like this:

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

While every address on Earth has a street1 value, the street2 value is optional. As a result, that class is subject to this type of abuse:

```scala
val santa = new Address(
  "1 Main Street",
```
47.6. USING OPTION TO REPLACE NULL VALUES

null, // <-- D'oh! A null value!
"North Pole",
"Alaska",
"99705"
)

To handle situations like this, developers tend to use null values or empty strings, both of which are hacks to work around the main problem: `street2` is an optional field. In Scala — and other modern languages — the correct solution is to declare up front that `street2` is optional:

class Address (
    var street1: String,
    var street2: Option[String],
    var city: String,
    var state: String,
    var zip: String
)

With that definition, developers can write more accurate code like this:

val santa = new Address(
    "1 Main Street",
    None,
    "North Pole",
    "Alaska",
    "99705"
)

or this:

val santa = new Address(
    "123 Main Street",
    Some("Apt. 2B"),
    "Talkeetna",
    "Alaska",
    "99676"
)

Once you have an optional field like this, you work with it as shown in the previous
examples: With match expressions, for expressions, and other built-in methods like foreach.

47.7 Option isn’t the only solution

This lesson focused on the Option/Some/None solution, but Scala has a few other alternatives. For example, a trio of classes known as Try/Success/Failure work in the same manner, but a) you primarily use these classes when code can throw exceptions, and b) the Failure class gives you access to the exception message. For example, Try/Success/Failure is commonly used when writing methods that interact with files, databases, and internet services, as those functions can easily throw exceptions. These classes are demonstrated in the Functional Error Handling lesson that follows.

47.8 Key points

This lesson was a little longer than the others, so here’s a quick review of the key points:

- Functional programmers don’t use null values
- A main replacement for null values is to use the Option/Some/None classes
- Common ways to work with Option values are match and for expressions
- Options can be thought of as containers of one item (Some) and no items (None)
- You can also use Options when defining constructor parameters

47.9 See also

- Tony Hoare invented the null reference in 1965, and refers to it as his “billion dollar mistake.”
A companion object in Scala is an object that's declared in the same file as a class, and has the same name as the class. For instance, when the following code is saved in a file named Pizza.scala, the Pizza object is considered to be a companion object to the Pizza class:

```scala
class Pizza {
}

object Pizza {
}
```

This has several benefits. First, a companion object and its class can access each other's private members (fields and methods). This means that the printFilename method in this class will work because it can access the HiddenFilename field in its companion object:

```scala
class SomeClass {
  def printFilename() = {
    println(SomeClass.HiddenFilename)
  }
}

object SomeClass {
  private val HiddenFilename = "/tmp/foo.bar"
}
```

A companion object offers much more functionality than this, and we'll demonstrate a few of its most important features in the rest of this lesson.
48.1 Creating new instances without the `new` keyword

You probably noticed in some examples in this book that you can create new instances of certain classes without having to use the `new` keyword before the class name, as in this example:

```scala
val zenMasters = List(
    Person("Nansen"),
    Person("Joshu")
)
```

This functionality comes from the use of companion objects. What happens is that when you define an `apply` method in a companion object, it has a special meaning to the Scala compiler. There’s a little syntactic sugar baked into Scala that lets you type this code:

```scala
val p = Person("Fred Flinstone")
```

and during the compilation process the compiler turns that code into this code:

```scala
val p = Person.apply("Fred Flinstone")
```

The `apply` method in the companion object acts as a Factory Method, and Scala’s syntactic sugar lets you use the syntax shown, creating new class instances without using the `new` keyword.

48.1.1 Enabling that functionality

To demonstrate how this feature works, here’s a class named `Person` along with an `apply` method in its companion object:

```scala
class Person {
    var name = ""
}

object Person {
    def apply(name: String): Person = {
        var p = new Person
        p.name = name
```
48.1. CREATING NEW INSTANCES WITHOUT THE NEW KEYWORD

```
p
}
}
```

To test this code, paste both the class and the object in the Scala REPL at the same time using this technique:

- Start the Scala REPL from your command line (with the scala command)
- Type `:paste` and press the [Enter] key
- The REPL should respond with this text:

```
// Entering paste mode (ctrl-D to finish)
```

- Now paste both the class and object into the REPL at the same time
- Press Ctrl-D to finish the “paste” process

When that process works you should see this output in the REPL:

```
defined class Person
defined object Person
```

The REPL requires that a class and its companion object be entered at the same time with this technique.

Now you can create a new instance of the `Person` class like this:

```
val p = Person.apply("Fred Flinstone")
```

That code directly calls `apply` in the companion object. More importantly, you can also create a new instance like this:

```
val p = Person("Fred Flinstone")
```

and this:

```
val zenMasters = List(
  Person("Nansen"),
  Person("Joshu")
)
```
To be clear, what happens in this process is:

- You type something like `val p = Person("Fred")`
- The Scala compiler sees that there is no new keyword before `Person`
- The compiler looks for an `apply` method in the companion object of the `Person` class that matches the type signature you entered
- If it finds an `apply` method, it uses it; if it doesn’t, you get a compiler error

### 48.1.2 Creating multiple constructors

You can create multiple `apply` methods in a companion object to provide multiple constructors. The following code shows how to create both one- and two-argument constructors. Because we introduced `Option` values in the previous lesson, this example also shows how to use `Option` in a situation like this:

```scala
class Person {
  var name: Option[String] = None
  var age: Option[Int] = None
  override def toString = s"$name, $age"
}

object Person {

  // a one-arg constructor
  def apply(name: Option[String]): Person = {
    var p = new Person
    p.name = name
    p
  }

  // a two-arg constructor
  def apply(name: Option[String], age: Option[Int]): Person = {
    var p = new Person
    p.name = name
    p.age = age
    p
  }
```
48.2. **ADDING AN UNAPPLY METHOD**

If you paste that code into the REPL as before, you’ll see that you can create new `Person` instances like this:

```scala
val p1 = Person(Some("Fred"))
val p2 = Person(None)

val p3 = Person(Some("Wilma"), Some(33))
val p4 = Person(Some("Wilma"), None)
```

When you print those values you’ll see these results:

```scala
val p1: Person = Some(Fred), None
val p2: Person = None, None
val p3: Person = Some(Wilma), Some(33)
val p4: Person = Some(Wilma), None
```

When running tests like this, it’s best to clear the REPL’s memory. To do this, use the `:reset` command inside the REPL before using the `:paste` command.

### 48.2 Adding an `unapply` method

Just as adding an `apply` method in a companion object lets you *construct* new object instances, adding an `unapply` lets you *de-construct* object instances. We’ll demonstrate this with an example.

Here’s a different version of a `Person` class and a companion object:

```scala
class Person(var name: String, var age: Int)

object Person {
    def unapply(p: Person): String = s"${p.name}, ${p.age}"
}
```

Notice that the companion object defines an `unapply` method. That method takes an input parameter of the type `Person`, and returns a `String`. To test the `unapply` method manually, first create a new `Person` instance:
val p = new Person("Lori", 29)

Then test unapply like this:

val result = Person.unapply(p)

This is what the unapply result looks like in the REPL:

scala> val result = Person.unapply(p)
result: String = Lori, 29

As shown, unapply de-constructs the Person instance it’s given. In Scala, when you put an unapply method in a companion object, it’s said that you’ve created an extractor method, because you’ve created a way to extract the fields out of the object.

48.2.1 unapply can return different types

In that example unapply returns a String, but you can write it to return anything. Here’s an example that returns the two fields in a tuple:

class Person(var name: String, var age: Int)

object Person {
    def unapply(p: Person): Tuple2[String, Int] = (p.name, p.age)
}

Here’s what that method looks like in the REPL:

scala> val result = Person.unapply(p)
result: (String, Int) = (Lori,29)

Because this unapply method returns the class fields as a tuple, you can also do this:

scala> val (name, age) = Person.unapply(p)
name: String = Lori
age: Int = 29
48.3. KEY POINTS

48.2.2 unapply extractors in the real world

A benefit of using unapply to create an extractor is that if you follow the proper Scala conventions, they enable a convenient form of pattern-matching in match expressions.

We'll discuss that more in the next lesson, but as you'll see, the story gets even better: You rarely need to write an unapply method yourself. Instead, what happens is that you get apply and unapply methods for free when you create your classes as case classes rather than as the “regular” Scala classes you’ve seen so far. We’ll dive into case classes in the next lesson.

48.3 Key points

The key points of this lesson are:

- A companion object is an object that’s declared in the same file as a class, and has the same name as the class
- A companion object and its class can access each other’s private members
- A companion object’s apply method lets you create new instances of a class without using the new keyword
- A companion object’s unapply method lets you de-construct an instance of a class into its individual components
Another Scala feature that provides support for functional programming is the *case class*. A case class has all of the functionality of a regular class, and more. When the compiler sees the *case* keyword in front of a *class*, it generates code for you, with the following benefits:

- Case class constructor parameters are public `val` fields by default, so accessor methods are generated for each parameter.
- An `apply` method is created in the companion object of the class, so you don’t need to use the `new` keyword to create a new instance of the class.
- An `unapply` method is generated, which lets you use case classes in more ways in `match` expressions.
- A copy method is generated in the class. You may not use this feature in Scala/OOP code, but it’s used all the time in Scala/FP.
- `equals` and `hashCode` methods are generated, which let you compare objects and easily use them as keys in maps.
- A default `toString` method is generated, which is helpful for debugging.

These features are all demonstrated in the following sections.

### 49.1 With `apply` you don’t need `new`

When you define a class as a case class, you don’t have to use the `new` keyword to create a new instance:

```scala
case class Person(name: String, relation: String)
defined class Person
```

// "new" not needed before Person
```scala
val christina = Person("Christina", "niece")
christina: Person = Person(Christina,niece)
```
As discussed in the previous lesson, this works because a method named `apply` is generated inside `Person`'s companion object.

### 49.2 No mutator methods

Case class constructor parameters are `val` fields by default, so an accessor method is generated for each parameter:

```scala
scala> christina.name
res0: String = Christina
```

But, mutator methods are not generated:

```scala
// can't mutate the `name` field
scala> christina.name = "Fred"
<console>:10: error: reassignment to val
   christina.name = "Fred"
^{
```

Because in FP you never mutate data structures, it makes sense that constructor fields default to `val`.

### 49.3 An `unapply` method

In the previous lesson on companion objects you saw how to write `unapply` methods. A great thing about a case class is that it automatically generates an `unapply` method for your class, so you don't have to write one.

To demonstrate this, imagine that you have this trait:

```scala
trait Person {
    def name: String
}
```

Then, create these case classes to extend that trait:

```scala
case class Student(name: String, year: Int) extends Person
case class Teacher(name: String, specialty: String) extends Person
```
Because those are defined as case classes — and they have built-in `unapply` methods — you can write a match expression like this:

```scala
def getPrintableString(p: Person): String = p match {
  case Student(name, year) =>
    s"$name is a student in Year $year."
  case Teacher(name, whatTheyTeach) =>
    s"$name teaches $whatTheyTeach."
}
```

Notice these two patterns in the case statements:

```scala
case Student(name, year) =>
case Teacher(name, whatTheyTeach) =>
```

Those patterns work because `Student` and `Teacher` are defined as case classes that have `unapply` methods whose type signature conforms to a certain standard. Technically, the specific type of pattern matching shown in these examples is known as a `constructor pattern`.

The Scala standard is that an `unapply` method returns the case class constructor fields in a tuple that's wrapped in an `Option`. The “tuple” part of the solution was shown in the previous lesson.

To show how that code works, create an instance of `Student` and `Teacher`:

```scala
val s = Student("Al", 1)
val t = Teacher("Bob Donnan", "Mathematics")
```

Next, this is what the output looks like in the REPL when you call `getPrintableString` with those two instances:

```scala
scala> getPrintableString(s)
res0: String = Al is a student in Year 1.

scala> getPrintableString(t)
res1: String = Bob Donnan teaches Mathematics.
```

All of this content on `unapply` methods and extractors is a little advanced
for an introductory book like this, but because case classes are an important FP topic, it seems better to cover them, rather than skipping over them.

49.4 copy method

A case class also has an automatically-generated copy method that’s extremely helpful when you need to perform the process of a) cloning an object and b) updating one or more of the fields during the cloning process. As an example, this is what the process looks like in the REPL:

```scala
case class BaseballTeam(name: String, lastWorldSeriesWin: Int)
defined class BaseballTeam
cubs1908 = BaseballTeam("Chicago Cubs", 1908)
cubs1908.copy(lastWorldSeriesWin = 2016)
```

As shown, when you use the copy method, all you have to do is supply the names of the fields you want to modify during the cloning process.

Because you never mutate data structures in FP, this is how you create a new instance of a class from an existing instance. This process can be referred to as, “update as you copy.”

49.5 equals and hashCode methods

Case classes also have automatically-generated equals and hashCode methods, so instances can be compared:

```scala
case class Person(name: String, relation: String)
defined class Person
c christina = Person("Christina", "niece")
```

```scala
c christina.copy(lastWorldSeriesWin = 2016)
```
hannah: Person = Person(Hannah,niece)

scala> christina == hannah
res1: Boolean = false

These methods also let you easily use your objects in collections like sets and maps.

49.6 toString methods

Finally, case classes also have a good default toString method implementation, which at the very least is helpful when debugging code:

scala> christina
res0: Person = Person(Christina,niece)

49.7 The biggest advantage

While all of these features are great benefits to functional programming, as they write in the book, Programming in Scala (Odersky, Spoon, and Venners), “the biggest advantage of case classes is that they support pattern matching.” Pattern matching is a major feature of FP languages, and Scala’s case classes provide a simple way to implement pattern matching in match expressions and other areas.
Before we jump into *case objects*, we should provide a little background on “regular” Scala objects. As we mentioned early in this book, you use a Scala object when you want to create a singleton object. As the documentation states, “Methods and values that aren’t associated with individual instances of a class belong in singleton objects, denoted by using the keyword `object` instead of `class`.”

A common example of this is when you create a “utilities” object, such as this one:

```scala
object PizzaUtils {
    def addTopping(p: Pizza, t: Topping): Pizza = ...
    def removeTopping(p: Pizza, t: Topping): Pizza = ...
    def removeAllToppings(p: Pizza): Pizza = ...
}
```

Or this one:

```scala
object FileUtils {
    def readTextFileAsString(filename: String): Try[String] = ...
    def copyFile(srcFile: File, destFile: File): Try[Boolean] = ...
    def readFromFileToArray(file: File): Try[Array[Byte]] = ...
    def readFromBuffer(file: File): Try[String] = ...
    def readLines(file: File, encoding: String): Try[List[String]] = ...
}
```

This is a common way of using the Scala `object` construct.

### 50.1 Case objects

A case object is like an object, but just like a case class has more features than a regular class, a case object has more features than a regular object. Its features include:
CHAPTER 50. CASE OBJECTS

- It’s serializable
- It has a default `hashCode` implementation
- It has an improved `toString` implementation

Because of these features, case objects are primarily used in two places (instead of regular objects):

- When creating enumerations
- When creating containers for “messages” that you want to pass between other objects (such as with the Akka actors library)

50.2 Creating enumerations with case objects

As we showed earlier in this book, you create enumerations in Scala like this:

```scala
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

Then later in your code you use those enumerations like this:

```scala
case class Pizza (
  crustSize: CrustSize,
  crustType: CrustType,
  toppings: Seq[Topping]
```
50.3. Using case objects as messages

Another place where case objects come in handy is when you want to model the concept of a “message.” For example, imagine that you’re writing an application like Amazon’s Alexa, and you want to be able to pass around “speak” messages like, “speak the enclosed text,” “stop speaking,” “pause,” and “resume.” In Scala you create singleton objects for those messages like this:

```scala
case class StartSpeakingMessage(textToSpeak: String)
case object StopSpeakingMessage
case object PauseSpeakingMessage
case object ResumeSpeakingMessage
```

Notice that `StartSpeakingMessage` is defined as a case class rather than a case object. This is because a case object can’t have any constructor parameters.

Given those messages, if Alexa was written using the Akka library, you’d find code like this in a “speak” class:

```scala
class Speak extends Actor {
  def receive = {
    case StartSpeakingMessage(textToSpeak) =>
      // code to speak the text
    case StopSpeakingMessage =>
      // code to stop speaking
    case PauseSpeakingMessage =>
      // code to pause speaking
    case ResumeSpeakingMessage =>
      // code to resume speaking
  }
}
```

This is a good, safe way to pass messages around in Scala applications.
Because functional programming is like algebra, there are no null values or exceptions. But of course you can still have exceptions when you try to access servers that are down or files that are missing, so what can you do? This lesson demonstrates the techniques of functional error handling in Scala.

51.1 Option/Some/None

We already demonstrated one of the techniques to handle errors in Scala: The trio of classes named Option, Some, and None. Instead of writing a method like toInt to throw an exception or return a null value, you declare that the method returns an Option, in this case an Option[Int]:

```scala
def toInt(s: String): Option[Int] = {
  try {
    Some(Integer.parseInt(s.trim))
  } catch {
    case e: Exception => None
  }
}
```

Later in your code you handle the result from toInt using match and for expressions:

```scala
toInt(x) match {
  case Some(i) => println(i)
  case None => println("That didn't work.")
}
```

```scala
val y = for {
  a <- toInt(stringA)
  b <- toInt(stringB)
  c <- toInt(stringC)
}
CHAPTER 51. FUNCTIONAL ERROR HANDLING IN SCALA

} yield a + b + c

These approaches were discussed in the “No Null Values” lesson, so we won’t repeat that discussion here.

51.2 Try/Success/Failure

Another trio of classes named Try, Success, and Failure work just like Option, Some, and None, but with two nice features:

• Try makes it very simple to catch exceptions
• Failure contains the exception message

Here’s the toInt method re-written to use these classes. First, import the classes into the current scope:

import scala.util.{Try, Success, Failure}

After that, this is what toInt looks like with Try:

def toInt(s: String): Try[Int] = Try {
  Integer.parseInt(s.trim)
}

As you can see, that’s quite a bit shorter than the Option/Some/None approach, and it can further be shortened to this:

def toInt(s: String): Try[Int] = Try(Integer.parseInt(s.trim))

Both of those approaches are much shorter than the Option/Some/None approach.

The REPL demonstrates how this works. First, the success case:

scala> val a = toInt("1")
a: scala.util.Try[Int] = Success(1)

Second, this is what it looks like when Integer.parseInt throws an exception:

scala> val b = toInt("boo")
b: scala.util.Try[Int] = Failure(java.lang.NumberFormatException: For input string: "boo")

As that output shows, the Failure that's returned by toInt contains the reason for the failure, i.e., the exception message.

There are quite a few ways to work with the results of a Try — including the ability to "recover" from the failure — but common approaches still involve using match and for expressions:

```
toint(x) match {
  case Success(i) => println(i)
  case Failure(s) => println(s"Failed. Reason: $s")
}
```

```
val y = for {
  a <- toInt(stringA)
  b <- toInt(stringB)
  c <- toInt(stringC)
} yield a + b + c
```

Note that when using a for-expression and everything works, it returns the value wrapped in a Success:

```
scala.util.Try[Int] = Success(6)
```

Conversely, if it fails, it returns a Failure:

```
scala.util.Try[Int] = Failure(java.lang.NumberFormatException: For input string: "a")
```

### 51.3 Even more ...

There are other classes that work in a similar manner, including Either/Left/Right in the Scala library, and other third-party libraries, but Option/Some/None and Try/Success/Failure are commonly used, and good to learn first.

You can use whatever you like, but Try/Success/Failure is generally used when dealing with code that can throw exceptions — because you almost always want to understand the exception — and Option/Some/None is used in other places, such as to avoid using null values.
In the next lesson you’ll see a primary tool for writing parallel and concurrent applications, the Scala Future.
When you want to write parallel and concurrent applications in Scala, you could still use the native Java Thread — but the Scala Future makes parallel/concurrent programming much simpler, and it’s preferred.

Here’s a description of Future from its Scaladoc:

“A Future represents a value which may or may not currently be available, but will be available at some point, or an exception if that value could not be made available.”

53.0.1 Thinking in futures

To help demonstrate this, in single-threaded programming you bind the result of a function call to a variable like this:

```scala
def aShortRunningTask(): Int = 42
val x = aShortRunningTask
```

With code like that, the value 42 is bound to the variable x immediately.

When you’re working with a Future, the assignment process looks similar:

```scala
def aLongRunningTask(): Future[Int] = ???
val x = aLongRunningTask
```

But because aLongRunningTask takes an indeterminate amount of time to return, the value in x may or may not be currently available, but it will be available at some point (in the future).

Another important point to know about futures is that they’re intended as a one-shot, “Handle this relatively slow computation on some other thread, and call me back with a result when you’re done” construct. (As a point of comparison, Akka actors are in-
tended to run for a long time and respond to many requests during their lifetime, but each future you create is intended to be run only once.)

In this lesson you’ll see how to use futures, including how to run multiple futures in parallel and combine their results in a for-expression, along with other methods that are used to handle the value in a future once it returns.

Tip: If you’re just starting to work with futures and find the name Future to be confusing in the following examples, replace it with the name ConcurrentResult, which might be easier to understand initially.

53.1 Source code

You can find the source code for this lesson at this URL:

- github.com/alvinj/HelloScalaFutures

53.2 An example in the REPL

A Scala Future is used to create a temporary pocket of concurrency that you use for one-shot needs. You typically use it when you need to call an algorithm that runs an indeterminate amount of time — such as calling a web service or executing a long-running algorithm — so you therefore want to run it off of the main thread.

To demonstrate how this works, let’s start with an example of a Future in the Scala REPL. First, paste in these import statements:

import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global
import scala.util.{Failure, Success}

Now, you’re ready to create a future. For example, here’s a future that sleeps for ten seconds and then returns the value 42:

scala> val a = Future { Thread.sleep(10*1000); 42 }
a: scala.concurrent.Future[Int] = Future(<not completed>)

While that’s a simple example, it shows the basic approach: Just construct a new Future with your long-running algorithm.
Because a `Future` has a `map` function, you use it as usual:

```scala
scala> val b = a.map(_ * 2)
b: scala.concurrent.Future[Int] = Future(<not completed>)
```

Initially this shows `Future(<not completed>)`, but if you check `b`’s value you’ll see that it eventually contains the expected result of 84:

```scala
scala> b
res1: scala.concurrent.Future[Int] = Future(Success(84))
```

Notice that the 84 you expected is wrapped in a `Success`, which is further wrapped in a `Future`. This is a key point to know: The value in a `Future` is always an instance of one of the `Try` types: `Success` or `Failure`. Therefore, when working with the result of a future, use the usual `Try`-handling techniques, or one of the other `Future` callback methods.

One commonly used callback method is `onComplete`, which takes a partial function in which you should handle the `Success` and `Failure` cases, like this:

```scala
a.onComplete {
  case Success(value) => println(s"Got the callback, value = $value")
  case Failure(e) => e.printStackTrace
}
```

When you paste that code in the REPL you’ll see the result:

```
Got the callback, value = 42
```

There are other ways to process the results from futures, and the most common methods are listed later in this lesson.

### 53.3 An example application

The following application (`App`) provides an introduction to using multiple futures. It shows several key points about how to work with futures:

- How to create futures
- How to combine multiple futures in a `for` expression to obtain a single result
• How to work with that result once you have it

53.3.1 A potentially slow-running method

First, imagine you have a method that accesses a web service to get the current price of a stock. Because it’s a web service it can be slow to return, and even fail. As a result, you create a method to run as a `Future`. It takes a stock symbol as an input parameter and returns the stock price as a `Double` inside a `Future`, so its signature looks like this:

```scala
```

To keep this tutorial simple we won’t access a real web service, so we’ll mock up a method that has a random run time before returning a result:

```scala
    val r = scala.util.Random
    val randomSleepTime = r.nextInt(3000)
    val randomPrice = r.nextDouble * 1000
    sleep(randomSleepTime)
    randomPrice
}
```

That method sleeps a random time up to 3000 ms, and also returns a random stock price. Notice how simple it is to create a method that runs as a `Future`: Just pass a block of code into the `Future` constructor to create the method body.

Next, imagine that you’re instructed to get three stock prices in parallel, and return their results once all three return. To do so, you write code like this:

```scala
package futures

import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.Future
import scala.util.{Failure, Success}

object MultipleFutures extends App {

    // use this to determine the “delta time” below
    val startTime = currentTime
```
// (a) create three futures
val aaplFuture = getStockPrice("AAPL")
val amznFuture = getStockPrice("AMZN")
val googFuture = getStockPrice("GOOG")

// (b) get a combined result in a for-expression
val result: Future[(Double, Double, Double)] = for {
  aapl <- aaplFuture
  amzn <- amznFuture
  goog <- googFuture
} yield (aapl, amzn, goog)

// (c) do whatever you need to do with the results
result.onComplete {
  case Success(x) => {
    val totalTime = deltaTime(startTime)
    println(s"In Success case, time delta: ${totalTime}")
    println(s"The stock prices are: $x")
  }
  case Failure(e) => e.printStackTrace
}

// important for a short parallel demo: you need to keep
// the jvm's main thread alive
sleep(5000)

def sleep(time: Long): Unit = Thread.sleep(time)

// a simulated web service
  val r = scala.util.Random
  val randomSleepTime = r.nextInt(3000)
  println(s"For $stockSymbol, sleep time is $randomSleepTime")
  val randomPrice = r.nextDouble * 1000
  sleep(randomSleepTime)
  randomPrice
}
def currentTime = System.currentTimeMillis()
def deltaTime(t0: Long) = currentTime - t0

}

Question: If everything truly runs in parallel, can you guess what the maximum value of the totalTime will be?

Answer: Because the three simulated web service calls do run in parallel, the total time should never be much longer than three seconds (3000ms). If they were run in series, the algorithm might run up to nine seconds.

This can be a fun little application to experiment with, so you’re encouraged to clone the Github project and run it before continuing this lesson. When you do so, first run it to make sure it works as expected, then change it as desired. If you run into problems, add println statements to the code so you can completely understand how it works.

Tip: The Github repository for this lesson also contains a class named MultipleFuturesWithDebugOutput that contains the same code with a lot of debug println statements.

53.3.2 Creating the futures

Let’s walk through that code to see how it works. First, we create three futures with these lines of code:

val aaplFuture = getStockPrice("AAPL")
val amznFuture = getStockPrice("AMZN")
val googFuture = getStockPrice("GOOG")

As you saw, getStockPrice is defined like this:

def getStockPrice(stockSymbol: String): Future[Double] = Future { ...

If you remember the lesson on companion objects, the way the body of that method works is that the code in between the curly braces is passed into the apply method of Future’s companion object, so the compiler translates that code to something like this:

def getStockPrice ... = Future.apply { method body here }
An important thing to know about Future is that it immediately begins running the block of code inside the curly braces — it isn't like the Java Thread, where you create an instance and later call its start method. You can see this very clearly in the debug output of the MultipleFuturesWithDebugOutput example, where the debug output in getStockPrice prints three times when the AAPL, AMZN, and GOOG futures are created, almost immediately after the application is started.

The three method calls eventually return the simulated stock prices. In fact, people often use the word eventually with futures because you typically use them when the return time of the algorithm is indeterminate: You don’t know when you’ll get a result back, you just hope to get a successful result back “eventually” (though you may also get an unsuccessful result).

53.3.3 The for expression

The for expression in the application looks like this:

```scala
val result: Future[(Double, Double, Double)] = for {
  aapl <- aaplFuture
  amzn <- amznFuture
  goog <- googFuture
} yield (aapl, amzn, goog)
```

You can read this as, “Whenever aapl, amzn, and goog all return with their values, combine them in a tuple, and assign that value to the variable result.” As shown, result has the type Future[(Double, Double, Double)], which is a tuple that contains three Double values, wrapped in a Future container.

It’s important to know that the application’s main thread doesn’t stop when getStockPrice is called, and it doesn’t stop at this for-expression either. In fact, if you print the result from System.currentTimeMillis() before and after the for-expression, you probably won’t see a difference of more than a few milliseconds. You can see that for yourself in the MultipleFuturesWithDebugOutput example.
53.4 onComplete

The final part of the application looks like this:

```scala
result.onComplete {
  case Success(x) => {
    val totalTime = deltaTime(startTime)
    println(s"In Success case, time delta: \${totalTime}\")
    println(s"The stock prices are: \$x\")
  }
  case Failure(e) => e.printStackTrace
}
```

onComplete is a method that's available on a Future, and you use it to process the future's result as a side effect. In the same way that the foreach method on collections classes returns Unit and is only used for side effects, onComplete returns Unit and you only use it for side effects like printing the results, updating a GUI, updating a database, etc.

You can read that code as, “Whenever result has a final value — i.e., after all of the futures return in the for-expression — come here. If everything returned successfully, run the println statement shown in the Success case. Otherwise, if an exception was thrown, go to the Failure case and print the exception's stack trace.”

As that code implies, it's completely possible that a Future may fail. For example, imagine that you call a web service, but the web service is down. That Future instance will contain an exception, so when you call result.onComplete like this, control will flow to the Failure case.

It's important to note that just as the JVM's main thread didn't stop at the for-expression, it doesn't block here, either. The code inside onComplete doesn't execute until after the for-expression assigns a value to result.

53.4.1 About that sleep call

A final point to note about small examples like this is that you need to have a sleep call at the end of your App:

```scala
sleep(5000)
```
That call keeps the main thread of the JVM alive for five seconds. If you don’t include a call like this, the JVM’s main thread will exit before you get a result from the three futures, which are running on other threads. This isn’t usually a problem in the real world, but it’s needed for little demos like this.

53.4.2 The other code

There are a few println statements in the code that use these methods:

```scala
def currentTime = System.currentTimeMillis()
def deltaTime(t0: Long) = System.currentTimeMillis() - t0
```

There are only a few println statements in this code, so you can focus on how the main parts of the application works. However, as you’ll see in the Github code, there are many more println statements in the `MultipleFuturesWithDebugOutput` example so you can see exactly how futures work.

53.5 Other Future methods

Futures have other methods that you can use. Common callback methods are:

- `onComplete`
- `onSuccess`
- `onFailure`

In addition to those methods, futures have methods that you’ll find on Scala collections classes, including:

- `filter`
- `foreach`
- `map`

Other useful and well-named methods include:

- `andThen`
- `fallbackTo`
• recoverWith

These methods and many more details are discussed on the “Futures and Promises” page.

53.6 Key points

While this was a short introduction, hopefully those examples give you an idea of how Scala futures work. A few key points about futures are:

• You construct futures to run tasks off of the main thread
• Futures are intended for one-shot, potentially long-running concurrent tasks that eventually return a value
• A future starts running as soon as you construct it
• A benefit of futures over threads is that they come with a variety of callback methods that simplify the process of working with concurrent threads, including the handling of exceptions and thread management
• Handle the result of a future with methods like onComplete, or combinator methods like map, flatMap, filter, andThen, etc.
• The value in a Future is always an instance of one of the Try types: Success or Failure
• If you’re using multiple futures to yield a single result, you’ll often want to combine them in a for-expression

53.7 See also

• A small demo GUI application named Future Board was written to accompany this lesson. It works a little like Flipboard, updating a group of news sources simultaneously. You can find the source code for Future Board in this Github repository.
• While futures are intended for one-short, relatively short-lived concurrent processes, Akka is an “actor model” library for Scala, and provides a terrific way to implement long-running parallel processes. (If this term is new to you, an actor is a long-running process that runs in parallel to the main application thread, and responds to messages that are sent to it.)
Where To Go Next

We hope you enjoyed this introduction to the Scala programming language, and we also hope we were able to share some of the beauty of the language.

As you continue working with Scala, you can find many more details at the Guides and Overviews section of our website.