Introduction

This is designed to be your first Scala book, but not your last. We're only showing you what you need to know to get to the end of the book -- to become familiar and comfortable with the language. Competent, but not expert. You'll be able to write useful Scala code, but you won't necessarily be able to read all the Scala code you come across.

At the end, you'll be ready for more complex Scala books, such as *Programming in Scala* by Odersky, Spoon and Venners.

This book is designed for a dedicated novice. "Novice" because you don't need prior programming knowledge, but "dedicated" because we want to ground you in programming and Scala without burying you in verbiage or swamping you with detail. Beginning programmers should think of it as a game: you *can* get through, but you'll need to tease out some puzzles along the way.

Experienced programmers can rapidly progress through the book and easily find the place where they need to slow down and start paying attention.

We won't make a business case for moving to Scala -- we assume you have your reasons, and that you'll see the value of it as you learn the language. For those who seek a better way, Scala makes its own argument.

Atomic Concepts

All programming languages consist of features that you apply to produce results. Scala is powerful: not only does it have more features, but you can usually express those features in numerous different ways. The combination of more features and more ways to express them can, if everything is dumped on you too quickly, make you run away, declaring that Scala is "too complicated."

It's not.

If you know the features and their expressions, you can look at any Scala code and tease out its meaning. Indeed, it's often easier to understand one page of Scala that produces the same effect as many pages of code in some other language, simply because you can see all the Scala code in one place.

But because it's easy to get overwhelmed, we must teach you the language carefully and deliberately, using the following principles:

1. **Baby steps and small wins**. We cast off the tyranny of the chapter. Instead, we present each small step as an *atomic concept* or simply *atom*, which will look like a tiny chapter. The

typical concept begins with a small, runnable piece of code and the output it produces. Then we describe what's new and different. Ideally we only present one new concept per atom.

2. **No forward references**. It often helps book authors to say, "We'll use these features now that will be explained in a later chapter." But this just confuses the reader, so we won't do it. We also won't make references to other languages. We don't know what languages you've learned (if any), and if we make an analogy to a feature in a language you don't understand, it just frustrates you.

3. **Show don't tell**. Instead of verbally describing a feature, we prefer examples and output that demonstrate what the feature does. It's much easier and more concrete to see it in code.

4. **Practice before theory**. We show code first, then describe it. We get you used to the mechanics of the language first, then tell you why those features exist. This is backwards from "traditional" teaching, but it often seems to work better.

This is an eBook

This book was designed to be an eBook. An eBook doesn't need to follow the same rules as a print book. In particular, if we need to reference some other part of the book, we don't need to use a print-book reference like "see section 14 in chapter 21." eBook readers and PDF documents understand hyperlinks, so we simply use those and dispense with the intermediate notation.

Free and Not Free

As you'll see, we've worked very hard on this book in order to make your learning experience the best it can be. In order to introduce you to the book and get you going in Scala, we've released the first volume of this book, [Title here], as a free distribution. Please post it on your website for download, and give it away to everyone you think might benefit.

The remaining volumes are for sale, in order for us to benefit financially from our efforts so that we can continue working on this and other projects. If you like what we've done in the free volume, please support us and help us continue our work by paying for the volumes you use. We hope what we've done helps and we greatly appreciate your sponsorship.

We also know that sometimes people can't pay for things. You might live in a country where the exchange rate makes the book too expensive. There are any number of reasons why you might honestly be unable to afford the book. We don't want you to *not* learn Scala because you can't pay for the book.

In the age of the Internet, it doesn't seem possible to control any piece of information. You'll probably be able to find the electronic version of this book in a number of places. If you are unable to pay for the book right now and you do download it from one of these places, please "pay it forward" by, for example, helping someone else learn the language once you've learned it. Or just help someone in any way that they need it. And perhaps sometime in the future you'll be better off, and you can come and buy something, or just make a donation to our tip jar:
http://www.AtomicScala.com/tipjar

About Us

Dianne Marsh is the co-founder of SRT Solutions in Ann Arbor, Michigan. Her expertise in software programming and technology includes manufacturing, genomics decision support and real-time processing applications. Dianne works with Unix, Windows, Java, C#, and C++ in enterprise-level applications, and has deep experience with a variety of graphical user interface libraries. A member of Women Presidents Organization, Dianne is also active in CodeMash and various Java user groups. She earned her Master of Science degree in computer science from Michigan Technological University. She has a husband and two lovely children and she talked Bruce into doing this book.

Bruce Eckel is the author of *Thinking in Java* and *Thinking in C++*, and a number of other books on computer programming. He's been in the computer industry for 30 years, periodically gets frustrated and tries to quit, then something like Scala comes along and offers hope and sucks him back in. He's given hundreds of presentations around the world and enjoys putting on alternative conferences and events like *The Java Posse Roundup*. He lives in Crested Butte, Colorado where he often acts in the community theatre. Although he will probably never be more than an intermediate-level skier or mountain biker, he finds these very enjoyable pursuits and considers them among his stable of life-projects. He is currently studying organizational dynamics, trying to find a new way to organize companies so that working together becomes a joy; you can read about his struggles in this arena at www.Reinventing-Business.com.

Installation

Scala runs on top of Java, so you must first install Java. You can check to see if it's already installed at:
http://java.com/en/download/installed.jsp
If you need to install it (you only need basic Java, not the development kit) go to:
http://java.com/en/download/index.jsp
and follow the instructions to install Java on your computer.

Next, you must include the Java directory in your computer's execution path, so that Scala can find it. On Windows, go to the control panel, select "System," then "Advanced System Settings," then "Environment Variables." Under "System variables," open "Path," then add to the end of the "Variable value" string:
;C:\Program Files (x86)\Java\jre6\bin
(This assumes the default location for the installation of Java. If you put it somewhere else, use the appropriate path). Close all the open windows by pressing "OK." Open a command prompt (press the Start button and start typing "command"; in Windows 7 you can click on any folder in the explorer while depressing the Shift key and select "command window here"). You should now be able to type "java" at the prompt (regardless of the subdirectory you're in) and get the list of command-line options for Java.

Next, download and install Scala from:
http://www.scala-lang.org/downloads
In this book, we're using version 2.9.

To install Scala, simply unpack the download into a directory. For simplicity, we used a directory called **Scala**, and after unpacking it contained:
bin doc lib meta misc src

In order to be able to run Scala from inside any directory on your machine, you must add the **bin** directory to your path, following the same directions as above. In our case, we add:
;C:\Scala\bin
To the end of the path. To test it, open a command prompt and type "scala" and you should get a welcome message and the Scala interpreter prompt.

For Mac and Linux, first make sure you have Java installed by opening a terminal window and typing **java** at the command prompt (again, it should work regardless of what directory you're in). Then go to the above Scala URL and download the version indicated (with the **.tgz** extension), and unpack it with the **tar** program that is part of the operating system. For example:
tar xvfz scala-2.9.0.RC1.tgz
This will produce the same subdirectories as above, as well as the Unix manual pages.

Just as for Windows, you must put the **bin** directory in your execution path. To do this, first move to your home directory:
cd ~
Now you need to discover where your **PATH** variable is defined, because it can be different depending on how the system is configured. The configuration files all start with '**.**' so they don't show up during a normal listing. One way to discover where your **PATH** is defined is by typing:

grep PATH .*
This will print all the lines containing the word “PATH” in all the files whose names begin with a ‘**.**’ My (Bruce’s) PATH is defined in the **.bash_profile** file, but yours might also be defined in **.bashrc** or elsewhere. Now you’ll need to edit that file using your favorite editor (probably the same one that you’ll use for editing Scala programs in future chapters). If you don’t already have an editor that you use, two that typically come with all Unix-based systems (including the Mac) are **vi** (or the more graphical **vim**) and **emacs**. Many a bloody flame war has been fought on the Internet between proponents of these two editors; suffice it to say you’ll need to decide which one to use. If you’re on the Mac, there are a number of Mac-based editors such as **textmate** that might be more to your taste.

Suppose you choose **emacs** for your editing, then type:
emacs .bash_profile
which will start up the editor on that file. At the end of the file, add:
PATH="~/Scala/bin/:${PATH}"
export PATH
(This assumes you unpacked the Scala files in a directory called **Scala** underneath your home directory; if not, use the path where you unpacked them). Note that the **:${PATH}** includes the existing **PATH** information; without it you will overwrite the old **PATH** (erasing it) and this will cause problems. The **export** statement makes the new **PATH** information available to your shell.

Once you’ve saved the file and exited your editor, you can include the new information in your current shell incarnation by running:
source .bash_profile
(Assuming that was the file where your **PATH** was defined). Now you should be able to type **scala** at the command prompt -- and in any shell you start up from now on -- and you’ll (eventually; startup times can be slow) see the Scala interpreter prompt.

The Scala interpreter is also known as the REPL (for *Read-Evaluate-Print-Loop*). You get the REPL when type "scala" by itself on the command line. You should see something like the following:

```
C:\Scala> scala
Welcome to Scala version 2.9.0.RC1 (Java HotSpot(TM) Client VM, Java
1.6.0_24).
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

The exact version numbers will vary depending on the version of Scala and Java you install.

The REPL gives you immediate interactive feedback with Scala. For example, you can do arithmetic:

```
scala> 42 * 11.3
res0: Double = 474.6
```

The “res0” is a “result” identifier and “Double” indicates the type of the result.

You can find out more about the REPL by typing **:help** at the Scala prompt.

To exit the REPL, type:

```
scala> sys.exit()
```

Actually, it works to just type **exit** at the prompt, but Scala will complain with a *deprecation warning*.

Comments

Comments allow you to insert illuminating text that is ignored by Scala. This way, you can tell anyone reading your code what is going on.

There are two forms of comment. The *//* begins a comment that goes to the end of a line:

```
47 * 42  // Perform multiplication
47 + 42
```

Scala will evaluate the multiplication, but will ignore the *//* and everything after it, until the end of the line. On the following line, it will pay attention again and perform the sum.

The multiline comment begins with a */** and continues -- including line breaks (usually called *newlines*) -- until a ***/* ends the comment:

```
47 + 42 /* A multiline comment
doesn't care
about newlines */ 47 * 42
```

It's possible to have code on the same line as the closing ***/* of a comment but it's confusing so people don't usually do it. In practice, you'll see the *//* comment used a lot more than the multiline comment.

Comments should add new information that isn't obvious from reading the code. If the comments just repeat what the code says, it becomes annoying. When the code changes, programmers often forget to update comments, so it's a good practice to use comments judiciously. However, because this book will be teaching you what the code does, we *will* use comments to explain the behavior of code.

Scripting

A *script* is a file filled with Scala code that you can run directly through the REPL. Suppose you have a file named **myfile-Script.scala** (we will use the "**-Script**" appellation on all our script files so you know how to run them). To execute that script from your operating system command prompt, you enter:

```
C:\Scala> scala myfile-Script.scala
```

Scala will then execute all the lines in your script.

Scripting makes it easy to quickly put together simple programs, so we will use it in the early part of this book. You can use scripting to solve basic problems, such as making utilities for your computer. However, more sophisticated programs require the use of the *compiler*, which we'll explore when the time comes.

To create a script, you need to use an *editor*. There are numerous editors available that are especially for programmers and ultimately you will have to decide which one suits you best. There are many free options: The Mac OSX and most versions of Linux include the **vi** and **emacs** editors, both of which have a long history as programming editors. On Windows you can find lots of editors; an excellent evaluation of free programming editors can be found here.

Once you've chose your editor and have become comfortable with it, create a file containing the following script:

```
// Basics/ScriptDemo-Script.scala
println("Hello, Scala!")
```

We will always begin a code file with a comment that contains the name of the file and the *relative path* of that file, based on where you unpacked the source code (You can get all the source code from this book at **http://AtomicScala.com/code**). When you unzip the file you find there, you will find fhis book's examples in a subdirectory called **code**.

This script has a single executable line of code. **`println("Hello, Scala!")`** tells Scala to display the string *"Hello, Scala!"* on the console. Run this script:

```
C:\Scala> scala ScriptDemo-Script.scala
```

And you should see:

```
Hello, Scala!
```

Values and Data Types

```
1   // Basics/Values-Script.scala
2   val b0b: Byte = 7        // Range: -2^7  to 2^7  -1
3   val small: Short = 8     // Range: -2^15 to 2^15 -1
4   val big_ger: Int = 11    // Range: -2^31 to 2^31 -1
5   val biggest: Long = 12L  // Range: -2^63 to 2^63 -1
6   val high: Float = 1.3f   // Must say '1.3f' or '1.3F'
7   val wide: Double = 1.4   // Optional '1.4d' or '1.4D'
8   val oneDimensional: Boolean = true // true or false
9   val a: Char = 'c'        // Range: 0 to 2^16 -1
10  val words: String = "A value"
11  val lines: String = """Triple quotes allow you
12  to have many lines
13  in your string"""
14
15  println(b0b, small, big_ger, biggest, high, wide,
16    oneDimensional, a, words)
17  println(lines)
18
19  /* Output:
20  (7,8,11,12,1.3,1.4,true,c,A value)
21  Triple quotes allow you
22  to have many lines
23  in your string
24  */
```

A *value* is a named piece of storage that holds a particular type of information. You define a value like this:

val *name*: *type* **=** *initialization*

That is, the **val** keyword followed by the name (that you make up), a colon, the type of the value, and the initialization value. The name must begin with a character, but can include things like numbers and underscores. Choosing a descriptive name is a good idea. It makes things easier for people who read your code and often reduces the need for comments.

By giving the type of your value, the compiler knows how much storage to allocate and it can verify that you put the right kind of data into the value.

You can see Scala's basic types in the example. The first four data types are *integral*, which means they only hold whole numbers. You can see from the comments that the different integral types hold different sizes of numbers. The '^' means "to the power of," so these numbers are powers of two, and two is used because computers count in binary (two states: on and off). Don't worry if you don't thoroughly understand the numbers; the basic idea is that different integral types hold different ranges of numbers.

Ordinarily, you'll just use an **Int** for integral values; the only time you need to worry about the others is if your numbers are too big for an **Int**, in which case you use a **Long**, or you are concerned with the amount of storage being used, in which case you use one of the smaller types.

The default type for floating-point numbers is a **Double**, but if you get in a situation where you're worried about the size of storage, you can use a **Float** (which has a smaller range and precision than a **Double**).

Left to its own devices, Scala will automatically choose **Int** to hold whole numbers and **Double** to hold fractional numbers, while **Byte**, **Short**, **Long** and **Float** are for special cases.

A **Boolean** can only hold the two special values **true** and **false**, which are predefined in Scala.

A **Char** is a special integral type designed to hold single characters (thus it doesn't hold negative values).

A **String** holds a sequence of characters. You can initialize it with a double-quoted string, or if you have many lines and/or special characters, you can surround them with triple-double-quotes.

In this book, we'll show the output at the end of a listing, inside a multiline comment. Note that **println()** will take a comma-separated sequence of values, or a single value.

Functions

```
1   // Basics/Functions-Script.scala
2   // Define a function:
3   def function1(x: Int): Int = {
4     println("Inside function1")
5     x * 2 // Return value
6   }
7
8   var r1: Int = function1(5) // Call the function
9   println(r1)
10
11  def function2(x: Int, y: Double, s: String): Double = {
12    println(s)
13    (x + y) * 2.1
14  }
15
16  var r2: Double = function2(7, 9, "Inside function2")
17  println(r2)
18
19  def function3(): Unit = {
20    println("Doesn't return anything")
21  }
22
23  function3()
24
25  /* Output:
26  Inside function1
27  10
28  Inside function2
29  33.6
30  Doesn't return anything
31  */
```

A *function* is some lines of code packaged under a name. When you call the name, that code is executed. The function allows you to summarize what you mean to do, as the function name, and is the most basic way to reuse code and structure programs.

Ordinarily, you pass information into a function, and the function uses that information to calculate a result, which it then returns to you. The basic form of a function in Scala is:

```
def functionName(arg1:Type1, arg2:Type2, …) : returnType = {
 lines of code
 result
}
```

All function definitions begin with the keyword **def**, followed by the function name and the

argument list in parentheses. The arguments are the information that you pass into the function, and each one has a name followed by a colon and the type of that argument. The closing parenthesis of the argument list is followed by a colon and the type of value that the function returns when you call it. Finally, there's an equal sign, to say "here's the function body itself." The lines of code in the function body are enclosed in curly braces, and the last line is the result that the function returns to you when it's finished. You don't have to say anything special to produce the result; it's just whatever is on the last line in the function.

On line 3, you'll see **function1**'s definition: the **def** keyword, the function name, and an argument list consisting of a single argument. Note that declaring the arguments is just like declaring **val**s: the argument name, a colon, and the type. This function takes an **Int** and returns an **Int**. Lines 4 and 5 are the body of the function. Note that line 5 just performs a calculation and since it's the last line, the result of that calculation becomes the result of the function.

Line 8 runs the function by *calling* it with an appropriate argument, and then captures the result into the variable **r1**. You can see how the function call mimics the form of its declaration: the function name, followed by arguments inside parentheses. Observe that **println()** is also a function call -- it just happens to be a function that is defined by Scala.

All the lines of code in a function (and you can put a lot of code inside) are now executed by a single call, so the function call becomes an abbreviation for that code. This is why functions are the most basic form of simplification and code reuse in programming.

On line 11, you see **function2** with 3 arguments of 3 different types. It returns a **Double** value and prints its 3rd argument, a **String**.

On line 19 is **function3**, which takes no arguments (the empty parentheses) and returns nothing, which Scala denotes with **Unit**. A function that doesn't return a result is called for its *side effects* -- whatever it does *other* than returning something that you can use.

If you read other Scala code you'll see many different ways to write functions than the above form. Scala is very expressive this way and it saves effort when writing and reading code. However, it can be confusing to see all these forms right away, when you're just learning the language, so for now we'll use this form and introduce the others after you've gotten more comfort with Scala. And while we've used the names **function1**, **function2**, and **function3** here, you should choose descriptive names to make reading the code easier and to reduce the need for code comments.

For Loops

```
1   // Basics/For-Script.scala
2
3   for(i <- 0 until 10) {
4     print(i + " ")
5   }
6   println()
7
8   for(i <- 0 until 20 by 2) {
9     print(i + " ")
10  }
11  println()
12
13  for(i <- Range(0, 10)) {
14    print(i + " ")
15  }
16  println()
17
18  for(i <- Range(0, 20, 2)) {
19    print(i + " ")
20  }
21  println()
22
23  /* Output:
24  0 1 2 3 4 5 6 7 8 9
25  0 2 4 6 8 10 12 14 16 18
26  0 1 2 3 4 5 6 7 8 9
27  0 2 4 6 8 10 12 14 16 18
28  */
```

A **for** loop steps through a sequence of items so you can perform operations using each item. On line 3, you see one of the simplest versions of a **for** loop: the symbol **<-** says that the variable **i** "gets" the values from 0 up to -- but not including -- 10, one at a time. The block of code in curly braces, following the **for**-expression, is executed for each value that **i** gets. You can put multiple lines of code inside the curly braces.

The call to **print()** displays information to the console, just like **println()** except that it doesn't output a newline at the end. In order to emit a newline, you can call **println()** with no arguments.

The code starting on line 8 does the same thing, but the **by 2** steps the sequence by a value of two instead of one (try different step values to see what happens).

The subsequent two **for** loops accomplish the same effect as the previous two, except they use **Range()** to produce the sequence of items. Scala often has multiple ways to produce the same result.

Here's something you might not have noticed: the value **i** was not defined anywhere! Scala does something very convenient called *type inference*. If possible, Scala figures out what a type should be, based on how you are using it, and it defines the variable for you. This doesn't mean that **i** isn't safe -- Scala still determines that it should be an **Int** and ensures that it follows all the rules for **Int**s. But it does this relatively trivial bit of work for you, so you can focus on the more meaningful code.

Scala does a lot of type inference for you, as part of its strategy of doing work for the programmer (rather than making the programmer work for the language). You can often just try leaving out the definition and see whether Scala will pick up the slack -- if not, it will give you an error message. We'll see more of this as we go.

Vectors and Objects

```
1   // VectorsAndObjects-Script.scala
2
3   val v1 = Vector(1, 3, 5, 7, 11, 13)  // A Vector holds other things
4   println(v1)
5
6   val n1 = v1(4) // "indexing" into a Vector
7   println(n1)
8
9   for(i <- v1) { // A for loop takes each element of the Vector
10    print(i + " ") // Add a space after each number
11  }
12  println() // Add a newline after all the numbers
13
14  val v3 = Vector(1.1, 2.2, 3.3, 4.4)
15  println(v3.reverse) // reverse is an "operation" on the Vector
16
17  var v4 = Vector("Twas", "Brillig", "And", "The", "Slithy", "Toves")
18  println(v4)
19  println(v4.sorted) // Only if it can be sorted
20  println(v4.head)
21  println(v4.tail)
22
23  /* Output:
24  Vector(1, 3, 5, 7, 11, 13)
25  11
26  1 3 5 7 11 13
27  Vector(4.4, 3.3, 2.2, 1.1)
28  Vector(Twas, Brillig, And, The, Slithy, Toves)
29  Vector(And, Brillig, Slithy, The, Toves, Twas)
30  Twas
31  Vector(Brillig, And, The, Slithy, Toves)
32  */
```

Now we'll learn about a tool provided by Scala: the **Vector**. A **Vector** is a *container* that holds any number of other things. On line 3, we create a **Vector** populated with **Int**s by simply stating the **Vector** name and handing it the initialization values. On line 24 you can see that when you **println()** a **Vector**, it produces the output in the same form as the initialization expression, making it easy to understand.

Notice something interesting about **v1**: we haven't told Scala what type it is by following **v1** with a colon and the type information (here, it would be something involving **Vector**). This is because Scala figures it out for you. This is called *type inference*. One of the nice things about Scala is that it will use type inference anywhere it can, which reduces the amount of code you must type and "visual noise" when reading Scala code. We could use type inference in this book a lot more than we do, but it's often easier to learn when things are explicit. We'll use type inference

when it makes things easier to understand, as it does in this case.

On line 6 parentheses are used to *index* into the **Vector**. A **Vector** keeps its elements in the order they were initialized, and you can select them individually by number. Most programming languages start indexing at element zero, which in this case would produce the value **1**. Thus, the index of **4** produces a value of **11**, as you can see on line 25. Forgetting that indexing starts at zero is responsible for the so-called *off-by-one* error.

In a language like Scala we often don't select elements one at a time, but instead *iterate* through a whole container -- an approach which eliminates off-by-one errors. On line 9, you can see that **for** loops work very well with **Vector**s: **for(i <- v1)** means "**i** gets each value in **v1**." This is another example of Scala helping you: you don't even have to declare **val i** or give its type -- Scala knows from the context that this is a **for** loop variable, and takes care of that for you. Many other programming languages will force you to do the extra work; that can be annoying because, in the back of your mind, you know that the language *can* figure it out and it seems like it's making you do the extra work out of spite. For this and many other reasons, programmers from other languages find Scala to be a breath of fresh air -- it seems to be saying, "How can I serve you?" instead of cracking a whip and forcing you to jump through hoops.

A **Vector** can hold all different types; on line 14 we create a **Vector** of **Double**. And on line 15 this is printed in reverse order, as you can see in the output on line 27. This is accomplished with the **reverse** *operation*. We say that a **Vector** is a kind of *object*, and the defining characteristic of objects is that you can perform operations on them. Sometimes we call this *sending a message to an object* or *calling a method on an object*, but an operation is really just a function that is associated with that object.

You call an operation on an object by giving the value name, then a dot, then the name of the operation. On line 15, the **reverse** operation is called for **v3** by saying **v3.reverse**.

A programming language that is *object-oriented* (OO) has a straightforward meaning: it is a language that is oriented towards the creation and use of objects.

Many objects have quite a number of operations available for them. In Scala, it's often easiest to explore such things using the REPL, which has the valuable characteristic of *code completion*. This means that if you start typing something and then hit the TAB button, the REPL will attempt to complete what you're typing. If it can't complete it, it will give you a list of options. So we can find the possible operations on a **Vector** like this (the REPL will give lots of information -- you can ignore the things you see here that we haven't talked about yet):

```
scala> val v = Vector()
v: scala.collection.immutable.Vector[Nothing] = Vector()

scala> v.(PRESS THE TAB KEY)
++                  ++:                 +:                  /:
/:\                 :+                  :\                  addString
aggregate           andThen             apply               asInstanceOf
canEqual            collect             collectFirst        combinations
companion           compose             contains            containsSlice
copyToArray         copyToBuffer        corresponds         count
diff                distinct            drop                dropRight
```

```
dropWhile            elements             endsWith             equalsWith
exists               filter               filterNot            find
findIndexOf          findLastIndexOf      first                firstOption
flatMap              flatten              fold                 foldLeft
foldRight            forall               foreach              foreachFast
genericBuilder       groupBy              grouped
             hasDefiniteSize
head                 headOption           indexOf              indexOfSlice
indexWhere           indices              init                 inits
intersect            isDefinedAt          isEmpty              isInstanceOf
isTraversableAgain   iterator             last                 lastIndexOf
lastIndexOfSlice     lastIndexWhere       lastOption           length
lengthCompare        lift                 map                  mapFast
max                  maxBy                min                  minBy
mkString             nonEmpty             orElse               padTo
par                  partition            patch                permutations
prefixLength         product              projection           reduce
reduceLeft           reduceLeftOption     reduceOption         reduceRight
reduceRightOption    repr                 reverse
             reverseIterator
reverseMap           reversedElements     sameElements         scan
scanLeft             scanRight            segmentLength        seq
size                 slice                sliding              sortBy
sortWith             sorted               span                 splitAt
startsWith           stringPrefix         sum                  tail
tails                take                 takeRight            takeWhile
toArray              toBuffer             toIndexedSeq         toIterable
toIterator           toList               toMap                toSeq
toSet                toStream             toString             toTraversable
transpose            union                unzip                unzip3
updated              view                 withFilter           zip
zipAll               zipWithIndex
```

Wow! There are lots of operations on a **Vector**, and many of them seem quite fancy. Some are simple and obvious, like **reverse**, and others seem like they might require more learning before you can use them. If you try to call some of these, you'll note that the REPL will tell you that you need some arguments for them. To find out enough to call those operations, you'll need to look them up in the Scala documentation, which you can either download or find online at ***http://www.scala-lang.org/api/current/index.html***
Try going to the link and typing **Vector** into the upper-left search box to see the results. From there, select **Vector** and then type one of the operations above into **Vector**'s search box, and scroll down to see the results. Although you won't understand most of it at this time, it's very helpful to begin getting used to the Scala documentation so that you can become comfortable looking things up.

In fact, while you're programming in Scala, it's good to keep a window open with the REPL running so you can do quick experiments when you have a question, and another window with the documentation so you can rapidly look things up.

The rest of the program experiments with a few other operations. Note the use of the word

sorted instead of just “sort.” When you call **sorted** it *produces* a new **Vector** containing the same elements as the old, in sorted order -- but it leaves the original **Vector** alone. If they had instead chosen to say “sort,” it would imply that the original **Vector** was changed directly (a.k.a *sorted in place*). Throughout Scala you will see this tendency of “leaving the original thing alone and producing a new thing.” For example, the **head** operation produces the first element of the **Vector** but leaves the original alone, and the **tail** operation produces a new **Vector** containing all but the first elements -- and leaves the original alone.

Discussion

1. **List** and **Set** are similar to **Vector**. Use the REPL to discover their operations and compare them to those of **Vector**.
2. Use the REPL to create several **Vector**s, each populated by a different type of data. Look at how the REPL responds and try to guess what it means.
3. Use the REPL to see if you can make a **Vector** containing other **Vector**s. How might you use such a thing?

Exercises

1. Write a script that creates a **Vector** and populates it with words (which are **String**s). Add a **for** loop that prints each element in the **Vector** *in reverse order*.
2. Create and initialize a **List** and **Set** with words, then print each one. Try the **reverse** and **sort** operations and see what happens.
3. Create and initialize two **Vector**s, one containing **Int**s and one containing **Double**s. Call the **sum**, **min** and **max** operations on each one and see what happens. Now create a **Vector** containing **String**s and call the same operations on that and explain the results.
4. Create a **Vector** called **v** containing 3 elements, then index **v(3)**. Does the result seem to make sense? We’ll eventually learn how to program for these situations.

Testing

```
1   // TestingExample.scala
2   import com.atomicscala.AtomicTest._
3
4   // Scala allows a "natural" alternative syntax
5   val v1 = Vector(1, 3, 5, 7, 11, 13)
6   val v2 = Vector(7, 9, 11, 13, 15, 17)
7   val x1 = v1.union(v2) // Remember your Venn diagrams?
8   val x2 = v1 union v2  // Alternative Scala syntax
9   println(x1)
10  println(x2)
11  val x3 = v1.intersect(v2)
12  val x4 = v1 intersect v2  // Alternative Scala syntax
13  println(x3)
14  println(x4)
15
16  // We can use the "natural" syntax for test expressions:
17  v1 is "Vector(1, 3, 5, 7, 11, 13)"
18  v1(4) is 11
19
20  val sb = new StringBuilder
21  for(i <- v1) {
22    sb.append(i + " ")
23  }
24
25  sb is "1 3 5 7 11 13 "
26
27  val v3 = Vector(1.1, 2.2, 3.3, 4.4)
28  v3.reverse is "Vector(4.4, 3.3, 2.2, 1.1)"
29  v3.reverse is Vector(4.4, 3.3, 2.2, 1.1)
30
31  var v4 = Vector("Twas", "Brillig", "And", "The", "Slithy", "Toves")
32  v4.sorted is "Vector(And, Brillig, Slithy, The, Toves, Twas)"
33  v4.head is "Twas"
34  v4.tail is "Vector(Brillig, And, The, Slithy, Toves)"
35  v4 is "Vector(Twas, Brillig, And, The, Slithy, Toves)"
36
37  "foo" is "bar" // Produces a simple error message
38
39  /* Output:
40  Vector(1, 3, 5, 7, 11, 13, 7, 9, 11, 13, 15, 17)
41  Vector(1, 3, 5, 7, 11, 13, 7, 9, 11, 13, 15, 17)
42  Vector(7, 11, 13)
43  Vector(7, 11, 13)
44  Vector(1, 3, 5, 7, 11, 13)
45  11
```

```
46  1 3 5 7 11 13
47  Vector(4.4, 3.3, 2.2, 1.1)
48  Vector(4.4, 3.3, 2.2, 1.1)
49  Vector(And, Brillig, Slithy, The, Toves, Twas)
50  Twas
51  Vector(Brillig, And, The, Slithy, Toves)
52  Vector(Twas, Brillig, And, The, Slithy, Toves)
53  foo [Error] expected:
54  bar
55  */
```

So far, we've been using **println()** to show results in our programs. By looking at the output we are able to verify that the code is working correctly. This is a rather weak way to test code because it requires that you look at it every time. Also, when you're initially creating your code you tend to write lots of **println()** statements to test various aspects of the program, but later these become distracting and you will often remove them -- thus losing the benefit of that test.

It turns out that, for code to be robust, it must be tested constantly -- every time you make any changes. This way, if you make a change in one part of your code that unexpectedly has an effect somewhere else, you know immediately, as soon as you make the change. Thus, you know which change caused things to break. If you don't find out immediately, then changes accumulate and you don't know which change caused the problem -- which means you'll spend a *lot* longer tracking it down. Constant testing is therefore essential for rapid program development.

Because testing is a critical practice, we decided to introduce it early in this book and use it throughout the rest of the book. This way, you'll become accustomed to testing as a standard part of the programming process.

We created our own tiny testing system primarily to simplify your experience using this book. The goal of this system is to be a completely minimalist testing approach that:

1. Allows you to see the expected result of expressions right next to those expressions, thus making comprehension easier while reading the examples.
2. Shows some output so you can see the program is running, even when all the tests succeed.
3. Introduces the concept of testing so it becomes ingrained early in your practice.
4. Requires no extra downloads or installations to work.

Although useful, this is *not* intended to be a testing system you'd use in the workplace. Others have worked long and hard to create such test systems -- in particular, Bill Venners' *ScalaTest* has become the defacto standard for testing, and you should reach for that when you start producing real Scala code.

Line 2 introduces a new feature: **import**, which allows you to use code from other files. In this case, we are using the **AtomicTest** *library* (essentially, an associated collection of code; a library is usually designed to solve a particular kind of problem) that we created for this book, which resides in the *package* called **com.atomicscala**. Notice that there's a URL http://www.AtomicScala.com, which means that (as long as everyone sticks to using their own domain names) **com.atomicscala** can be considered unique to this book, and this means we can use it to avoid name collisions with other libraries that might also use the name **AtomicTest**. This reversed-domain package name is thus standard Scala practice.

The '**_**' at the end of **import com.atomicscala.AtomicTest._** means "include everything."

We don't intend that you understand the code for **com.atomicscala.AtomicTest** because it uses some tricks that are beyond the scope of this book. However, if you'd like to see the code it is in Appendix A.

In order to produce a clean, comfortable appearance, **AtomicTest** uses a Scala feature that you haven't seen before: The ability to write a function call **a.function(b)** in the text-like form **a function b**. For example, **Vector** has an operation called **union** that combines all the elements from two **Vector**s. You can see an ordinary call on line 7. And then on line 8 the text-like call produces exactly the same effect (Scala actually turns it into the call on line 7). You see a second example using **Vector**'s **intersect** ("take only the items common between two **Vector**s") on lines 11 and 12.

On lines 17 and 18, you see how we use this feature in **AtomicTest**: by defining an **is** function so you can say:

```
expression is expected
```

If *expected* is a string, then *expression* is converted to a string and the two strings are compared. Otherwise, *expression* and *expected* are just compared directly (without converting them first). In either case, *expression* is printed to the console so that you can see something happening when the program runs. And, if *expression* and *expected* are not equivalent, **AtomicTest** prints an error message when the program runs.

That's all there is to it. The **is** function is the only operation defined for **AtomicTest** -- it truly is a minimal testing system.

Now you can put "**is**" expressions anywhere you want in a script to produce both a test and some console output. In this example, we've still included a commented output block, but in most examples from now on we won't need it because the testing code will do everything we need (and better, because you can see the results right there rather than scrolling to the bottom and detecting which line of output corresponds to a particular **println()** statement).

Ideally, by seeing the benefits of using testing throughout the rest of the book, you'll become addicted to the idea of testing and will feel uncomfortable when you see code that doesn't have tests. You will probably start feeling like code without tests is broken by definition.

Note the flexibility of the system -- almost anything works. For example, line 28, the *expected* is a string but on line 29 an actual **Vector** is created and compared, but both of them work fine.

Line 37 intentionally fails, to show you the output for a failure on lines 53 and 54. It's very simple but it does the job. Anytime you run a program that uses **AtomicTest**, you'll automatically verify the correctness of that program.

There's one other thing about writing programs with testing in mind -- it often changes the way you think about and design your code. For example, in lines 20-23 we could have just printed the results to the console. But the test mindset makes you think, "How will I test this?" In this case, we capture the strings into another object -- a **StringBuilder**, which lets you **append()** one string at a time to create a bigger string. At the end, you have a string that you can test,

rather than just some pieces that you've thrown onto the console and then lost. With this mindset, when you create a function you begin thinking you should return something from that function, if for no other reason than to test. It turns out that creating functions to take one thing and transform that into something else tends to produce better designs, as well.

Pattern Matching

```
1   // Basics/PatternMatching-Script.scala
2
3   def printColor(color: String) =  {
4
5      color match {
6        case "red" => println("RED")
7        case "blue" => println("BLUE")
8        case "green" => println("GREEN")
9        case _ => println("UNKNOWN COLOR: " + color)
10     }
11  }
12
13  printColor("white")
14  printColor("blue")
15
16  def getColorShade(shade: Int) = {
17     shade match {
18       case 1  => "light"
19       case 2  => "medium"
20       case 3  => "dark"
21       case _  => "unknown"
22     }
23  }
24
25  println (getColorShade(1) )
26  println (getColorShade(3) )
27  println (getColorShade(65) )
28
29  /* Output:
30  UNKNOWN COLOR: white
31  BLUE
32  light
33  dark
34  unknown
35  */
```

A *match expression* lets you match what is stored in a value against a variety of other things. When the 2 things match, the code on the right side of the arrow (=>) is executed.

All match expressions begin a value that you would like to compare, followed by the keyword **match**. Follow this with a set of possible matches and their associated actions. Each expression to compare begins with the keyword **case** followed by a statement. The statements are executed and if they match the value, the code to the right of the arrow is executed. The lines of code in the match expression are enclosed in curly braces.

On line 5, you'll see *the match expression's* definition: the value name **color** followed by the **match** keyword and then a set of statements in curly braces, representing things to match against. On line 6, we compare the value **color** to "red". On line 7, we compare against "blue" and then on line 8, "green".

Line 9 is a special case. It uses an "_" (pronounced "underscore") to match whatever isn't matched above. In other words, if we set the value of color to "white" on line 13, it wouldn't match red, blue, or green, but would match the statement on line 7, and print "UNKNOWN COLOR: white." In Scala, we call this a *wildcard pattern*.

On line 14, we set the value of **color** to "blue" so the code to the right of the arrow on line 7 is executed, and "BLUE" is printed.

Matches always result in a value in Scala, but since the last statement in each of the match patterns is a function that returns nothing (**Unit**), printColor's match expression does this as well.

Match expressions can return a more interesting value. On line 16, we have defined a function that will return a value set by the match expression, instead of printing it. If you're confused about why the **getColorShade** function returns a value, review Functions.

Discussion

Exercises

Case Classes

```
//Basics/CaseClasses-Script.scala
1   case class Dog (name: String)
4
5   val dog1 = Dog("Henry")
6   val dog2 = Dog("Cleo")
7
8   val dogs = Vector(dog1, dog2)
9   println(dogs)

case class Cat(name: String, age: Int)
val cats = Vector(Cat("Miffy", 3), Cat("Rags", 2))
println(cats)
/* Output
println(dogs)
Vector(Dog(Henry), Dog(Cleo))
println(cats)
Vector(Cat(Miffy,3), Cat(Rags,2))
*/
```

In the last atom we introduced the idea of an object. You might wonder whether you can create your own types of objects. Yes, you can, and that's what most of object-oriented programming is about. You create your own types of objects by defining *classes*, and the most convenient way to do this in Scala is with a *case class*. You define a case class like this:

> **case class** *TypeName***(***arg1: Type, arg2: Type, ...***)**

That is, the **case class** keywords followed by the name (that you make up), a left parenthesis followed by something that you want to describe about the object, followed by a right parenthesis. The name must begin with a character, but can include things like numbers and underscores.

The thing that you want to describe about the object is contained in parenthesis and follows the same rules that we saw previously in Values and Data Types. That is, use a name (that you make up), a colon, and the type of the value. You do not need to include the keyword **val** here, because the **case class** assumes that's what you want to do.

Notice the two different **Dog**s in this example. Case classes provide you with a way to print what you've created. You see both the name of the case class (in this case, **Dog**) and the interesting thing that you wanted to keep with the **Dog** (in this case, **name**). The class **Cat** contains two pieces of information: **name** and **age**. The **Cat**s go directly into the **Vector** without any intermediate identifiers.

Discussion

Case classes are easy to create and give you a ready-made object that can be easily created. They are also easy to stuff into Collections (such as Vectors) and to print.

Exercises

1. Create a case class that represents a Person in an Address Book, complete with Name and Contact Information.
2. Create some “Person” objects.
3. Put the “Person” objects in a Vector.
4. Print out the list of “Person” objects.
5. Update the Contact Information with a new email address (for example). Did you have to change your definition of the case class to make this possible?
6. Print out the list of “Person” objects again. Did the list reflect the updated address? Why or why not?

If Statements

```
1   // If.scala
2   import com.atomicscala.AtomicTest._
3
4   def simple(exp: Boolean): String = {
5     if(exp) {
6       return "It's true!" // 'return' necessary
7     }
8     "It's false"
9   }
10
11  simple(true) is "It's true!"
12  simple(false) is "It's false"
13
14  def withElse(exp: Boolean): String = {
15    if(exp) {
16      "It's true!" // No 'return'
17    } else {
18      "It's false"
19    }
20  }
21
22  val v = Vector(1)
23  val v2 = Vector(3, 4)
24  withElse(v == v.reverse) is "It's true!"
25  withElse(v2 == v2.reverse) is "It's false"
26
27  def elseIf(exp1: Boolean, exp2: Boolean): String = {
28    if(exp1 && exp2) {
29      "Both are true"
30    } else if(!exp1 && !exp2) {
31      "Both are false"
32    } else if(exp1) {
33      "First: true, second: false"
34    } else {
35      "First: false, second: true"
36    }
37  }
38
39  elseIf(true || false, true) is "Both are true"
40  elseIf(1 > 0 && -1 < 0, 1 == 2) is "First: true, second: false"
41  elseIf(1 >= 2, 1 >= 1) is "First: false, second: true"
42  elseIf(true && false, false && true) is "Both are false"
43
44  def assignResult(arg: Boolean): Int = {
45    val result = if(arg) 42 else 47
```

```
46    result
47  }
48
49  assignResult(true) is 42
50  assignResult(false) is 47
```

An **If** statement makes a choice. It tests an expression to see whether it's **true** or **false** and does something based on the result. A true-or-false expression is called a *Boolean*, after the mathematician George Boole who invented the logic behind such expressions. All the examples in this demonstration code are cast in the form of functions that take one or more **Boolean** arguments.

The simplest form of the **if** is seen in lines 4-9. The **Boolean** expression **exp** is tested and if it is **true**, the lines inside the curly braces are executed. There's something new here, however: the **return** statement. **return** is a Scala keyword that says: "Leave this function and return this value." Normally, the last statement in a Scala function produces the value returned from that function, so we don't usually need the **return** keyword and you won't see it very often. In **simple()**, if we just gave the **String** "It's true" without the **return**, nothing would happen, the function would continue and "It's false" would always be returned from **simple()** (try it -- remove the **return** and see what happens).

A more typical way of coding this is seen in **withElse()**, which introduces the **else** keyword. The **else** keyword is only used in conjunction with **if**, and **withElse()** now consists of a single statement, instead of the two statements in **simple()**. The result of that statement -- the last statement in the first set of braces if **exp** is true, the last in the second set of braces if **exp** is false -- becomes the returned value, so the **return** keyword is no longer necessary (some people feel strongly that **return** should never be used to exit a function in the middle, but we remain neutral on the subject).

The tests for **withElse()** show that if a **Vector** of length one is reversed it is always equal to the original, but if it is longer than one the reverse typically *isn't* equal to the original.

You are not limited to a single test. You can test multiple combinations by combining **else** and **if** as seen in **elseIf()**. The typical pattern is to start with an **if**, followed by as many **else if** clauses as you need, and ending with an **else** for anything that doesn't match all the previous tests. Note that when an **if** statement reaches a certain size and complexity you might use pattern matching instead if that is simpler.

In the body of **elseIf()** and in its tests, we introduce more *Boolean Algebra*: the **&&** means "and" and it requires both **Boolean** expressions to be **true** in order to produce a **true** result. The **!** means "not" and it takes the **Boolean** expression to its right and inverts the result. The **||** means "or" and produces a **true** result if either the expression on the left or right is **true** (or if both are **true**). Study the code to deepen your understanding.

The last function, **assignResult()**, demonstrates that, because an **if** statement produces a result, you can "assign an **if** statement to a value." Well, that's what it looks like, but what you're really doing is just assigning the *result* of the statement. Line 45 also shows an abbreviated form of **if**, all on the same line with no curly braces. This is sometimes useful when an **if** is extremely simple, and you're assigning the result.

Discussion

1. Under what conditions would a **Vector** of length greater than one be equal to its reverse?

Exercises

1. Rewrite **elself()** as a **match** statement. (I don't know how hard this is)
2. Write a function that generates **Vector** objects that are equal to their reverse. (This might be too hard).

Parameterized Types

```
1   // ParameterizedTypes.scala
2   import com.atomicscala.AtomicTest._
3
4   // Type is inferred:
5   val v1 = Vector(1,2,3)
6   val v2 = Vector("one", "two", "three")
7   // Exactly the same, but explcitly typed:
8   val p1: Vector[Int] = Vector(1,2,3)
9   val p2: Vector[String] = Vector("one", "two", "three")
10
11  v1 is p1
12  v2 is p2
13
14  // Return type is inferred:
15  def f(c1: Char, c2: Char, c3: Char) = {
16    Vector(c1, c2, c3)
17  }
18
19  // Explicit return type:
20  def g(c1: Char, c2: Char, c3: Char): Vector[Char] = {
21    Vector(c1, c2, c3)
22  }
23
24  f('a', 'b', 'c') is g('a', 'b', 'c')
```

We've been making heavy use of Scala's type inference, and this is good -- let the language do the work for you. Sometimes, however, it can't figure out what type to use, and we have to help it along (Scala will always complain when it needs such help).

There are a number of cases requiring explicit type declarations, but in this book we need to address the issue of containers. That is, we know we are dealing with, for example, a **Vector**, but sometimes we need to tell Scala what type is contained in that **Vector**. Often, Scala can figure this out. Lines 5 and 6 are what we've usually been seeing -- a simple **val** on the left and a simple **Vector** on the right. Scala looks at the initialization values and detects that the **Vector** on line 5 contains **Int**s and the **Vector** on line 6 contains **String**s.

We consider it a "best practice" to let Scala infer types whenever possible. It tends to make the code cleaner and easier to read. However, as an example, we'll rewrite lines 5 and 6 using explicit typing. Line 8 is the rewrite of line 5. The part on the right side of the equals sign is the same, but on the left we've added the colon and the type declaration, which is **Vector[Int]** . The square brackets are new here; they are how we denote a type parameter in Scala. For a container the type parameter tells the kind of object the container holds. You typically pronounce **Vector[Int]** "vector of int," and the same for other types of container: "list of int," "set of int" and so forth.

Type parameters are useful for other things than containers, but you'll usually see them with container-like objects, and in this book we'll generally stick to using **Vector** as our container.

On line 15 we allow Scala to infer the return type of the function, and on line 17 we specify the function return type. You can't just say that it returns a **Vector**; Scala will complain, so you have to give the type parameter as well.

Discussion

Exercises

1. Modify **g()** (starting on line 20) so that it creates and returns a **Vector** of **Double**.
2. Modify **g()** to create and return a **List**. Make another version that creates and returns a **Set**.

For Expressions

```
1   // ForExpressions.scala
2   import com.atomicscala.AtomicTest._
3
4   def filter(v: Vector[Int]): Vector[Int] = {
5     var result = Vector[Int]() // 'var' so we can change it
6     for(n <- v
7         if n > 5
8         if n % 2 == 0) {
9           result = result :+ n
10    }
11    result
12  }
13
14  val v = Vector(1,2,3,5,6,7,8,10,13,14,17)
15  filter(v) is Vector(6,8,10,14)
16
17  def filterWithYield(v: Vector[Int]): Vector[Int] = {
18    val result = for(n <- v
19        if n < 10
20        if n % 2 != 0)
21          yield n
22    result
23  }
24
25  filterWithYield(v) is Vector(1,3,5,7)
```

Now that you know about type parameters, you're ready to learn about a powerful combination of **for** and **if** called the *for expression*.

*For Expression*s combine *generators*, *filters*, and *definitions* in Scala. The for loop that you saw in For Loops was a *for expression* with one *generator*, and that's also what we see on line 18 as:

```
for (n <- v)
```

But **for** expressions can be more complex. The **filter()** function that begins on line 4 takes and returns **Vector**s containing **Int**s. It selects **Int**s satisfying a particular criteria from the input **Vector** and puts those in the **result Vector**. Notice something new: **result** is not the usual **val**, but rather a **var**, for "variable," which means we can modify **result**. Ordinarily, we always try to use a **val** because **val**s can't be changed and that gives them a lot of safety benefits (which are beyond what we can talk about just yet). Sometimes, however, you just can't seem to achieve your goal without changing an object, and for that you need **var**. Here, we are building up the **result** vector by assembling it, so **result** must be changeable. By initializing it with **Vector[Int]()** we establish the type parameter as **Int** and create an empty **Vector**.

On line 6, the **for** expression begins in typical fashion: **n** gets all the values from **v**. But instead

of stopping there as usual, we see two **if** statements before the closing parenthesis. Each of these selects the value of **n** that will make it through the **for** expression. First, each **n** that we're looking for must be greater than 5. But also, an **n** of interest must be such that **n % 2 == 0**. The '**%**' the *modulo* operator and determines whether there is any remainder. So **n % x** asks, "What is the remainder when **n** is divided by **x**?" Thus, **n % 2 == 0** selects all the even numbers, and the **for** expression says, "Find all the numbers in **v** that are greater than 5 and also are even."

Next, we want to append all those numbers to **result**. Because **result** is a **var**, we can assign to it. However, the nature of **Vector** is that it can't be changed (the aforementioned safety effect again), so how do we "add to" our **Vector**? **Vector** has an operator '**:+**' which creates a new **Vector** by taking an existing one (but *not* changing it) and combining it with the element to the right of the operator. So **result = result :+ n** produces a new **Vector** by appending **n** to the old one, and then assigns this new **Vector** to **result** (the old **Vector** is thrown away and Scala automatically cleans it up). By the time we finish the **for** loop, we've created a new **Vector** filled with the desired values.

It turns out there *is* a way to use **val** for **result**, and that's to build **result** "in place" rather than creating it piece-by-piece. Scala provides a special keyword called **yield** to help with this. When you say **yield n**, it "yields up" the value **n** to become part of the **result**. You can see **yield** used inside **filterWithYield()** beginning on line 17 (we'll let you figure out what values this function is looking for).

You always use **yield** to fill a container. But we haven't declared the type of **result** on line 18, so how does Scala know what kind of container to create? It infers the type from the container that the **for** loop is traversing -- **v** is a **Vector[Int]**, so **yield** creates a **Vector[Int]**. Now, with a **for** expression and **yield**, we can create the entire **Vector** before assigning it to **result**, so **result** can be a **val** instead of a **var**.

Discussion

Exercises

Pattern Matching with Types

```
1   // AtomicScala/Basics/PatternMatchingOfTypes-script.scala
2
3   def doSomethingInteresting(x : Any) =
4   x match {
5     case s: String => println("It's a String and its value is " + s)
6     case i: Int => println("It's an Int and its value is " + i)
7     case p: Person => println("It's a person named " + p.name)
8     case _ => println("I don't know what that is!")
9   }
```

```
10
11  doSomethingInteresting(5)
12  doSomethingInteresting("Some text")
13
14  case class Person(name: String)
15  val bob = Person("Bob")
16  doSomethingInteresting(bob)
17
18  doSomethingInteresting(Vector(1, 2, 5))
19
20  /* Output:
21  It's an Int and its value is 5
22  It's a String and its value is Some text
23  It's a person named Bob
24  I don't know what that is!
25  */
```

We already showed you how to do pattern matching on values, but pattern matching can do a lot more. You can also match against the *type* of a value, and do something different based on what type you passed in. We created a function called “doSomethingInteresting” that doesn’t care what type of value is passed in. We tell the function that it can take any type of value by using the type **Any** for **x**, on line 2. Then, we compare against the types that we’re interested in, using pattern matching. In this case, we have added comparisons for Int and String and print a message about the match. We’ve also added a comparison for a value of our own type, Person. If the type of the value that we pass in is something else, we use the underscore (**_**), and we print a different message.

We’ve included a value declaration for each type, as “s: String”, “i: Int”, and “p: Person” as part of the match expression. By doing this, we can use the value that we’re comparing against in a calculation or in the simple print statement. We can use the **name** of a **Person** as easily as we can use an **Int** or a **String**.

Pattern Matching with Case Classes

```
1   // CaseClassPatternMatching.scala
2   import com.atomicscala.AtomicTest._
3
4   case class Color(hue: Int)
5   case class Part(id: String)
6   case class Location(x: Int, y: Int)
7
8   def detectAndUnpack(thing: Any): String = {
9     thing match {
10      case Color(hue)  => "Color " + hue
11      case p: Part => "Part " + p
12      case loc: Location => loc + " " + loc.x + ", " + loc.y
13      case x => "Something else " + x
14    }
15  }
16  detectAndUnpack(Color(33)) is "Color 33"
17  detectAndUnpack(Part("Widget")) is "Part Part(Widget)"
18  detectAndUnpack(Location(20,42)) is "Location(20,42) 20, 42"
19  detectAndUnpack(3.14) is "Something else 3.14"
20
21  case class Passenger(first: String, last: String)
22  case class Train(passengers: Vector[Passenger], line: String)
23  case class Plane(passengers: Vector[Passenger], plane: String)
24  case class Bus(passengers: Vector[Passenger], capacity: Int)
25
26  def travel(transport: Any): String = {
27    transport match {
28      case Train(trvlrs, line) =>
29        "Train line " + line + " " + trvlrs
30      case Plane(trvlrs, equip) =>
31        "Plane " + equip + " " + trvlrs
32      case Bus(trvlrs, cap) =>
33        "Bus size " + cap + " " + trvlrs
34    }
35  }
36
37  val travelers = Vector(Passenger("Harvey", "Rabbit"),
38                         Passenger("Dorothy", "Gale"))
39
```

```
40  val trip = Vector(Train(travelers, "Reading"),
41                    Plane(travelers, "B757"),
42                    Bus(travelers, 100))
43
44  travel(trip(0)) is "Train line Reading " +
45    "Vector(Passenger(Harvey,Rabbit), Passenger(Dorothy,Gale))"
46  travel(trip(1)) is "Plane B757 " +
47    "Vector(Passenger(Harvey,Rabbit), Passenger(Dorothy,Gale))"
48  travel(trip(2)) is "Bus size 100 " +
49    "Vector(Passenger(Harvey,Rabbit), Passenger(Dorothy,Gale))"
```

So far, we've seen how **case** classes conveniently add useful functionality. However, they were originally designed for use in **match** expressions, thus they are ideally suited for that task. Not only can a **match** expression easily detect a **case** class, but it can also extract the argument fields from a **case** class.

In lines 4-6 we create some typical case classes, and lines 8-15 define a function containing a single pattern match. The argument type for **detectAndUnpack()** is something we haven't seen before: **Any**. As the name implies, **Any** allows any type to be passed as this argument. We need **Any** in this situation because we want to apply **detectAndUnpack()** to all the **case** classes we've defined above, and they have nothing in common. An **Any** argument accepts everything, and so solves the problem (we'll explore this issue of commonality later in the book).

Line 10 shows how a **case** class can be matched -- including the argument(s) used to create the matched object. Here, the argument is named **hue**, just like in the class definition, but you can use any name. When a match happens, the identifier **hue** is created and gets the value used as the argument when the **Color** object was created, so it can be used in the expression on the right side of the "rocket" symbol.

You aren't forced to unpack the **case** class arguments. Lines 11 and 12 simply match the type, without the arguments. But once you have the value, (**p** or **loc**), you can treat it like a regular object and access its properties.

Line 13 matches an identifier (**x**) without a type, which means it matches anything else that the **case** statements above it missed. Here you still have an identifier, which we use as part of the resulting string on the right of the "rocket." If you don't want to use the matched value, you can use the special character '**_**' as the identifier (this could be called the "whatever" character).

Now let's look at a slightly more interesting example: a basic description of a trip taken by travelers using various modes of transportation. Line 21 is a **Passenger** class containing the name of the passenger, and lines 22-24 show different modes of transportation along with varying details about each mode. However, all transportation types have something in common: they carry passengers. The simplest "passenger list" we can make is a **Vector[Passenger]**. Notice how easy it is to include this in the class: just put it in the class argument list.

The **travel()** method contains a single **match** statement, as before, and each **case** extracts the arguments. On line 37 we create a **Vector[Passenger]** and on line 40 we create a **Vector** of the different types of transportation. Each type of transportation carries our travelers and also has details about the transportation. The point of this second example is not to introduce any new features, but rather to show the power of Scala -- how easy it is to build a model that represents your system. As you learn more about Scala, you'll discover that it contains numerous ways to keep representations simple, even as your systems get more complex.

Objects, Fields, and Methods

```
1   // ObjectsFieldsMethods.scala
2   import com.atomicscala.AtomicTest._
3
4   class Simple()
5   val s1 = new Simple()
6   val s2 = new Simple()
7   println(s1)
8   println(s2)
9
10  case class AutoField1(x: Int)
11  val af1 = AutoField1(11)
12  af1.x is 11
13
14  class AutoField2(val x: Int)
15  val af2 = new AutoField2(22)
16  af2.x is 22
17
18  class ExplicitField(x: Int) {
19    val y: Int = x + 10
20    val z = x * 10  // Type inference
21    val q = y * z + 100
22  }
23  val ef1 = new ExplicitField(7)
24  val ef2 = new ExplicitField(12)
25  ef1.y is 17; ef1.z is 70;  ef1.q is 1290
26  ef2.y is 22; ef2.z is 120; ef2.q is 2740
27
28  class Celsius(var temp: Double) {
29    def set(newTemp: Double): Unit = {
30      temp = newTemp
31    }
32    def fahrenheit(): Double = {
33      temp * 9/5 + 32
34    }
35  }
36  val c = new Celsius(0)
37  c.fahrenheit() is 32
38  c.set(100)
39  c.fahrenheit() is 212
```

```
40  c.set(-40)
41  c.fahrenheit() is -40
```

So far we've been waving our hands around the term "object," and now it's time to get more precise. Objects are the foundation for numerous modern languages, but only *half* the foundation for Scala, which is an *object-functional* language. We'll study the functional aspect of Scala in Volume 2 of this book.

Unfortunately, there is no universal meaning for the term object across programming languages, so we'll talk about what it means to Scala. While some of these concepts will apply to other languages, keep in mind that there are differences.

In the most general (and vague) sense of the term, an object is *a region of storage*. This means that an object must occupy some unique space in memory, which means each object has a unique address. If we create a class and make two objects of that class, printing those objects will produce something that involves the address in memory of that object. On line 4, we create **class Simple**.

Notice that we don't try to use the "**is**" testing mechanism in this case, but print the objects to the console. You should see *something* like this for the **println()** statements:

```
Main$$anon$1$Simple@60de1b8a
Main$$anon$1$Simple@15e232b5
```

However, if you run the program multiple times, the number after the '**@**' sign should change each time (thus, trying to use "**is**" would fail). Yes, that's actually a number, even though it contains letters -- this is *hexadecimal notation*, which is base-16 and uses the digits 0-9 as well as the letters a-e. That's really all you need to know about hexadecimal for now, but if you're interested, look it up in Wikipedia.

The number after the '**@**' is the address in memory where the object begins. You can see that each object starts at a different address, thus each object is unique. If you change one object, it doesn't affect the other.

When we use the term "class," it is a shorthand for "a class of objects," in the same way that Greek philosophers described "a class of fishes and a class of birds." A class represents objects that have common characteristics and behaviors. We express characteristics as *fields* and behaviors as *methods*.

A field stores data in an object. We've been using Scala's shortcut notation to produce fields. In a **case** class, any arguments automatically become **val** fields, as shown in lines 10-12. In an ordinary class, we must put **val** before a class argument in order to have it automatically stored as a field, as you can see in lines 14-16.

You can also explicitly create fields in a class, as seen in lines 18-22. Here, we create a class with a *body* which begins after the argument list and is surrounded by curly braces. Inside the body we simply define **val**s and their initialization, but now these values are *part of the class*.

This means that, every time you create a new object of class **ExplicitField**, that object will get its own storage for **y**, **z** and **q**. You can see this in the creation and comparison of **ef1** and **ef2** (the use of '**;**' allows us to put multiple **is** statements on the same line).

ExplicitField does not store the value of **x** directly, but instead uses **x** to calculate values for **y**, **z**, and **q**. It's also possible to add fields to **case** classes, but if you put a **case** before **class** in **ExplicitField**, you'd also automatically store **x** as a field.

So characteristics are represented as fields. What about the behavior of objects? We implement behavior using *methods*, which are simply functions that are part of a class. Once again, all we do is nest inside the class definition, as seen in **Celsius** on lines 28-35. In this case we make the class argument a **var** so we can demonstrate how to change a field (which *mutates* the object), which we do using **set()**. The **set()** method only changes the object but doesn't return anything, so we use Scala's **Unit** (which means "nothing") as the return type.

Look closer at **set()** and **fahrenheit()**. Both of them access the field **temp**, and yet there's no obvious connection -- other than the fact that the methods belong to the class. Scala invisibly makes this connection between fields and methods (as well as methods to methods) within the same class by giving them special access to each other. A class is its own little world.

Exercises

1. Make a class that has an integer field **height** and an integer field **width** that can be both retrieved and set externally.
2. Make a class that has a String field **name** that can be retrieved externally and a String field **description** that can be both updated and retrieved.
3. Challenge Goal: Modify **Celsius** and its tests to eliminate the mutability of the class. We have provided some steps that you can use as hints.

Step 1. Get rid of set method. Create a new object each time you want to convert.
Discussion: Does this seem inconvenient? Why call a constructor with a value if all you can ever do is display in fahrenheit?

Step 2. Call fahrenheit with the value to convert, like this:
fahrenheit(100)
Discussion: What else might you want to do with this class? Since all you have is a method to convert a number from celsius to fahrenheit, let's add a method to convert fahrenheit to celsius.

Step 3: Make a more generic class that can convert values from celsius to fahrenheit and from fahrenheit to celsius.
Discussion: Do you sacrifice anything by doing this? Have we accomplished our goal of making the class immutable?

Brevity & Style

```
1   // BrevityAndStyle.scala
2   import com.atomicscala.AtomicTest._
3
4  // Infer return type, but '=' still required:
5  def filterWithYield2(v: Vector[Int]) = {
6     val result = for(n <- v
7         if n < 10
8         if n % 2 != 0)
9           yield n
10    result
11  }
12
13  // Single statement, no braces needed:
14  def filterWithYield3(v: Vector[Int]) =
15    for(n <- v
16        if n < 10
17        if n % 2 != 0)
18    yield n
19
20  // Semicolons allow for-expression on a single line:
21  def filterWithYield4(v: Vector[Int]) =
22    for(n <- v; if n < 10; if n % 2 != 0) yield n
23
24  val v = Vector(1,2,3,5,6,7,8,10,13,14,17)
25  filterWithYield2(v) is Vector(1,3,5,7)
26  filterWithYield3(v) is Vector(1,3,5,7)
27  filterWithYield4(v) is Vector(1,3,5,7)
28
29  // Return type necessary, otherwise Unit:
30  def thingsA(thing: String): Int = {
31    thing match {
32      case "one" => 1
33      case "two" => 2
34    }
35  }
36
37  def thingsB(thing: String) = thing match {
38    case "one" => 1
39    case "two" => 2
```

```
40  }
41
42  thingsA("two") is 2
43  thingsB("one") is 1
44
45  class Simple(val s: String) {
46    def getA() = s
47    def getB = s
48  }
49
50  val simple = new Simple("Hi")
51  simple.getA() is "Hi"
52  simple.getA is "Hi"
53  simple.getB is "Hi"
54  // simple.getB() is "Hi" // Rejected
```

One of the notable things about Scala is its succinctness. In reaction to many other languages that require the programmer to write a lot of code (often called “boilerplate” or “jumping through hoops”) to do simple things, Scala allows you to express concepts very briefly. Sometimes, arguably, *too* briefly. Indeed, as you learn more about the language you will understand that, more than anything, it’s this powerful brevity that can be responsible for the mistaken idea that Scala is “too complicated.” On the contrary, your experience during this book should be that Scala is clear and straightforward (if we’re doing our job).

Up until now we’ve stuck to a very consistent form and have not introduced any of these syntactic shorteners. However, we don’t want you to be too surprised when you see real Scala code, so we are going to introduce a few of the coding short-forms. This way you’ll get used to the idea that they exist, and you’ll can start to get comfortable with them.

Also, there are one or two style issues that are worth introducing at this point.

Let’s start by revisiting the **filterWithYield()** function from For Expressions. The first thing we can do, as seen on line 5, is to let Scala infer the return type of the function by leaving off the return type. However, the ‘**=**’ sign is still necessary between the function argument list and the function body. If you leave off the ‘**=**’ Scala will decide that you mean the function returns nothing, expressed as **Unit** or ‘**()**’. Try removing the ‘**=**’ and you’ll see the error message produced by the test on line 25.

If a function consists of a single statement, the curly braces around the function are unnecessary. **filterWithYield2()** is effectively only one statement: create and return **result**. By removing the intermediate **result** value, we produce **filterWithYield3()** on lines 14 through 18, which is a single statement that doesn’t need the surrounding curly braces.

We don’t have to stop there. Note that lines 15-17 are distinct expressions within the

parentheses of the **for** expression. In that case the line breaks tell Scala the end of each expression. However, we can put them all on the same line by using semicolons, as you can see on lines 21-22. In fact, you can put those two lines onto a single line (try it).

Although a **match** expression is typically placed on multiple lines, it is still a single expression. Look at **thingsA()** on lines 30-35. It contains one expression, so we can lose the curly braces as in **thingsB()** on lines 37-40. Something interesting happens here, though. On line 37 we use type inference for the return type of the function, and it works fine. But on line 30, *with* the additional curly braces, type inference doesn't work (Scala assumes a **Unit** return value). Try it -- leave off the '**: Int**' on line 30 and you'll see the error message from the test code on line 42.

In general, Scala will figure out what you mean whenever it can. The safest way to approach Scala brevity is to start by being completely explicit, and then slowly pare down by removing extraneous pieces. When you go too far, either Scala will produce an error message or (as in the case of trying to imply a return type on **thingA()**) you'll get the wrong result. Of course, you need to test everything as you go to make sure you don't miss something.

Style

Most programming languages develop style guides as they mature. In some cases these guides tell you how to format the code on the page for greater readability. Fortunately, the code-formatting style of Scala has been established from the inception of the language (and, in some cases, is enforced by the language syntax). Most editing tools that support a Scala mode will automatically format your code as you create it, so you don't need to think very hard about that.

There are higher-level style issues, as well. Most of these help make your code more meaningful to the reader. An important guideline involves the use of parentheses on methods that take no arguments.

Lines 45-48 show a class that stores its argument **String s**. Neither method **getA()** nor **getB** takes arguments. Note that both methods are as succinct as they can possibly be: single statements (so no curly braces are needed) that return the value of **s**, so Scala infers that they each produce a **String**.

The first thing you see is that a method without arguments can leave off the parentheses in the definition, as shown in **getB** on line 47. And in the test code on lines 51 and 52, you can see that even though **getA** is defined with parentheses, it can be called with or without them. Finally, because **getB** is defined without parentheses, it can *only* be called without parentheses.

Here's the style question: Since Scala is flexible about the way you call a method that doesn't have arguments, does it matter? Yes: parentheses have stylistic meaning in the Scala community. If calling a method modifies the internal state of the object -- that is, if there are internal variables that get changed when the method is called -- then you should leave the parentheses *on* in the method definition. This signals to the reader that this is a *mutating* method (it causes the object to change). Ideally, when you call the method you also include the

parentheses (although, as we've seen, the compiler doesn't require it).

On the other hand, if calling the method simply produces a result without changing the state of the object, the convention is to leave the parentheses *off* of the method definition, to indicate to the reader that this method reads date from the object without mutating it. Since both methods simply return the stored value of **s**, **getB** is the preferred form.

There are certainly other style issues that you will discover as you learn more about Scala, but these are the ones we need for this book, so we'll stop there for now.

Exercises

1. Modify the **Celsius** example from the last chapter to conform to the brevity and style guidelines shown in this chapter.

toString

```
1   // StringRepresentations.scala
2   import com.atomicscala.AtomicTest._
3
4   case class Bicycle(riders: Int)
5   val forTwo = Bicycle(2)
6   forTwo is "Bicycle(2)" // Nice
7
8   class Surrey(val adornment: String)
9   val fancy = new Surrey("fringe on top")
10  println(fancy) // Ugly
11
12  class Surrey2(val adornment: String) {
13    override def toString = "Surrey with the " + adornment
14  }
15
16  val fancy2 = new Surrey2("fringe on top")
17  fancy2 is "Surrey with the fringe on top"
```

One of the conveniences provided by a **case** class is that it formats an object nicely for display, including its arguments, as you can see on line 6. The reason this happens is that there's a method called **toString** that is automatically defined when you create the **case** class. Whenever you do anything with that object that expects a **String**, Scala silently calls **toString** to produce a **String** representation for an object.

Interestingly, if you create a regular, non-**case** class, you still get a **toString**, as you see on line 10. But this is the default **toString** and isn't usually very useful. To get a better one, you can define your own **toString** as on line 13.

The first thing you see is the keyword **override**. This is necessary (Scala insists on it) because **toString** is already defined (the definition that produces the ugly result). We must tell Scala that, yes, we do actually want to replace it with our own definition. Note that we use the brief forms described in the previous atom: no parentheses because this method doesn't change the object, return type inference, and a single-line method.

A good **toString** is useful when debugging a program; sometimes all you need is to see what's in the object to know what's going wrong.

Exercises

1. You can override **toString** in a **case** class, too. Modify **Bicycle** so that its **toString**

says “Bicycle built for 2” for the above example.

Satisfy the following test:

```
val forTwo = Bicycle(2)
forTwo is “Bicycle built for 2”
```

1. The **toString** method doesn’t have to be as simple as a single line method.

   Step 1: Change the class name to **Cycle** and pass the number of wheels to the constructor.
   Step 2: Use a match expression to display “Unicycle” for a single wheeled cycle, “Bicycle” for 2 wheels, “Tricycle” for 3 wheels, “Quadricycle” for 4 wheels, and “Cycle with n wheels” for numbers greater than 4, replacing the n with the number passed in.

   Satisfy the following tests:

```
val c1 = Cycle(1)
c1 is "Unicycle"
val c2 = Cycle(2)
c2 is "Bicycle"
val cn = Cycle(5)
cn is "Cycle with 5 wheels"
```

Regular Classes & Companion Objects

```
1   // RegularClasses.scala
2   import com.atomicscala.AtomicTest._
3
4   // Things that case classes give you for free:
5   case class CaseClass(n: Int, s: String)
6   val cc1 = CaseClass(1, "hi") // Simple construction
7   val cc2 = CaseClass(1, "hi")
8   val cc3 = CaseClass(2, "hi")
9   cc1 == cc2 is true  // Automatic creation of '=='
10  cc1 == cc3 is false
11  cc2 != cc3 is true  // ... and '!='
12  cc3.n is 2 // Arguments stored and accessible
13  cc1 is "CaseClass(1,hi)" // Nice String output
14
15  def matchCaseClass(cc: CaseClass): String = {
16    cc match {
17      // Automatic unpacking/matching of arguments:
18      case CaseClass(n, s) => "n = " + n + " s = " + s
19      case _ => "Everything Else"
20    }
21  }
22  matchCaseClass(cc1) is "n = 1 s = hi"
23
24  class Regular(n: Int, s: String)
25  val r1 = new Regular(3, "hi") // Must say 'new'
26  // n and s are not members of Regular!
27
28  class Regular2(val n: Int, val s: String)
29  val r2a = new Regular2(4, "hello")
30  val r2b = new Regular2(4, "hello")
31  r2a.n is 4
32  r2b.s is "hello"
33  r2a == r2b is false // '==' not created correctly
34  println(r2a) // String representation is ugly!
35
36  class Regular3(val d: Double) {
37    override def toString = "Regular3 contains " + d
38  }
39  object Regular3 {
```

```
40    def apply(d: Double) = new Regular3(d)
41  }
42  val r3 = Regular3(3.14) // No 'new' needed
43  r3 is "Regular3 contains 3.14"
44
45  case class WithCompanion(c: Char)
46  object WithCompanion {
47    def show(wc: WithCompanion) = "Char: " + wc.c
48  }
49  val wc = WithCompanion('x') // No 'new' needed
50  wc is "WithCompanion(x)" // Nice String representation
51  WithCompanion.show(wc) is "Char: x"
52  import WithCompanion._ // Now you don't have to qualify:
53  show(wc) is "Char: x"
```

Until now we've only been looking at **case** classes, because in most situations these are the simplest to create and most convenient to use. For one thing, **case** classes can easily be used in pattern matching expressions, which makes them quite powerful. Indeed, the primary motivation for **case** classes is pattern matching.

Lines 5-22 summarize the basics of what we get from **case** classes. Line 5 shows how simple it is to create a **case** class, and lines 6-8 create some objects using the simplified construction syntax that **case** classes provide (the use of the word "simplified" will make more sense in a moment). Lines 9-11 show that sensible '**==**' and '**!=**' operators are automatically provided by the **case** class. Line 12 demonstrates how the arguments of a **case** class are automatically stored and accessible as fields of each object. On Line 13, you can see how a **case** class produces a nice, readable **String** output.

Lines 15-21 review how convenient **case** classes are when used in **match** expressions, making them trivial to use.

Sometimes you don't need **case** classes, and their use can be restrictive in other ways (as we'll discover later in the book). In these situations you can just leave off the **case** keyword and you'll get an ordinary class, as you can see on line 24. The first effect this has is seen on line 25: you can't just give the name of the class in order to create a new object. Instead, you must use the **new** keyword.

At this point we can't get much further, because in ordinary classes, the class arguments are *not* automatically stored. To store them, we must add the **val** keyword in the class argument list as seen on lines 28-32.

On line 33, we see how an ordinary class doesn't have a properly-defined '**==**' (and by implication, '**!=**'). Defining these by hand is beyond the scope of this book; we'll just use **case** classes when we need them.

Line 34 produces the **String** representation of a non-**case** class, which you'll see from the output is rather ugly (we couldn't do an inline test for this because of the inconsistency of the output, but it looks something like **Main$$anon1Regular2@62facf0b**). To clean it up, on lines 36-38 we define a class that has its own **toString** method.

On line 39 you encounter a keyword you haven't seen before: **object**. We've talked about objects before, but only as "things you create from classes" or more precisely, "instances of classes." A class is like a mold, and from it you can create many objects that look like the class and follow the rules established by that class.

When you use the **object** keyword you're saying, "I only want to make a single one of these," versus the **class** keyword which suggests you're going to make many objects. The **object** keyword skips the process of making a class, and goes directly to making that one object. There's a fancy name for what the **object** keyword does, which is *singleton*. The meaning is exactly as it sounds -- there's only a single one.

Here's another interesting case in the example: If an **object** has the same name as a **class**, it's called the *companion object* and it works in concert with the class. So **object Regular3** is the companion object for **class Regular3**.

Notice the **apply()** method on line 40. This is a special method that is called when you take an object and put parentheses on it. For example, when you say **Regular3(3.14)** then **apply()** is automatically called. This is how we define the "**new**-less" object creation, and this is also what a **case** class automatically generates.

case classes are very flexible, and will fill in anything you don't explicitly create. This is shown in lines 45-53. We create a simple **case** class and then a companion object with a method **show()** . But explicitly creating a companion object doesn't stop the mechanism of the **case** class from adding in its convenience methods -- as you can see on lines 49 and 50, the usual **apply()** and **toString** methods are still automatically created.

On line 51 you see the explicit call to **WithCompanion.show()**. Without the qualification (that is, the '**WithCompanion.**' part), Scala wouldn't know what function to call. If this becomes too unwieldy, however, you can use an **import** statement as you see on line 52, which basically says, "bring everything inside **WithCompanion** into the local scope." Now, on line 53 you can just say **show()** and the compiler knows what you mean.

Constructors

```
1   // Constructors.scala
2
3   class Simple(msg: String) {
4     println("Simple: " + msg) // Argument available
5     println("Any amount of code can be executed")
6     def f = "Methods and fields can be"
7     val s = "mixed in"
8     println(f + " " + s)
9     val n = { println("initializing n"); s.length }
10    println("length of s == " + n)
11    println("Execution is in definition order")
12  }
13
14  val s1 = new Simple("Hi there!")
15
16  class Simple2(msg: String, num: Int) {
17    println("Simple2: " + msg + " " + num)
18    def this(msg: String) = {
19      // println("Try uncommenting this")
20      this(msg, 0)
21      println("Call to default must appear first")
22    }
23    def this(num: Int) = this("Empty", num)
24    def this() = this("Nought", 0)
25    println("primary constructor completed")
26  }
27
28  val s2 = new Simple2("Howdy!", 92)
29  val s3 = new Simple2("Konichiwa!")
30  val s4 = new Simple2(47)
31  val s5 = new Simple2
32
33  case class Simple3(num: Int) {
34    println("Simple3: " + num)
35    def this() = this(0)
36  }
37
38  val s6 = Simple3(42)
39  val s7 = new Simple3
```

One of the benefits of using objects is that you can determine how all objects of a particular class will be automatically initialized as they are created. You perform this initialization by defining a *constructor* for that class.

Any code you define inside the body of a class (but not within function definitions) becomes part of the constructor. This code will be executed as the object is created, before the object is handed back to the person asking for that object. This way you can guarantee proper initialization of your object.

Lines 3-12 show the behavior of a basic constructor. Notice that construction code is sprinkled throughout the whole body of the class. Also, since constructor code is executed during the creation of an object, we don't have a chance to compare the results with anything. Thus, our normal testing procedure (using **AtomicTest** and **is**) doesn't work here so we'll just use **println** for this example.

After a couple of **println** statements, the method **f** is defined. All it does is return a **String**, so it follows the convention of leaving off the parentheses because it doesn't modify the internal state of the object.

Line 9 introduces something new: An initialization expression for a **val** or **var** can be *compound*, which means it can exist of multiple statements surrounded by curly braces (because the multiple statements are on the same line, they are separated by a semicolon). The last expression in the block will be produced as the resulting initialization value. Notice also that other variables outside this block can be used within the compound initialization expression. Thus, a compound initialization expression can be quite complex, but without requiring you to define a separate function if you don't need one.

The output for line 14 is:

```
Simple: Hi there!
Any amount of code can be executed
Methods and fields can be mixed in
initializing n
length of s == 8
Execution is in definition order
```

The class argument **msg** is passed in and each line of initialization code -- including the initialization code for the **val**s -- is executed in the order that those lines were written in the class definition. Values like **s** are only available *after* they've been defined (try moving line 9 *before* line 7 and see what happens).

The only way to create a **Simple** object is to provide a single **String** argument to the constructor, as seen in line 14. What if you would like to have other ways to create an object? It's possible to *overload* the constructor and create multiple forms with different types of

arguments using the **this** keyword. If you define a method which has **this** as its name, that becomes a secondary constructor. It must have a different argument list than the primary constructor (also called the *default constructor*).

Class **Simple2** starting on line 16 demonstrates overloaded constructors. The primary constructor takes a **String** and an **Int**, and the overloaded constructors on lines 18, 23 and 24 take just a **String**, just an **Int**, and no arguments, respectively. We learn two things about overloaded constructors: first, that you *must* call the primary constructor (by calling **this** with the appropriate arguments) inside the overloaded constructor, and second, that this call must be the first thing you do inside the overloaded constructor. Any deviations result in an error message from Scala.

The constructor requirements are not arbitrary. They ensure that there is a standard way to create an object, and, in overloaded constructors, that the object has been correctly created (via the call to the primary constructor) before you can make any modifications to it. Without these requirements it would be possible to put the object into an unknown state.

The output for lines 28-31 is:

```
Simple2: Howdy! 92
primary constructor completed
Simple2: Konichiwa! 0
primary constructor completed
Call to default must appear first
Simple2: Empty 47
primary constructor completed
Simple2: Nought 0
primary constructor completed
```

The constructor of lines 18-22 shows that the primary constructor runs to completion -- thus creating a valid object -- before allowing any modifications by the secondary constructor.

It's also possible to add construction code and overloaded constructors to **case** classes, as seen in lines 33-36. Things get rather strange here, because the primary constructor is called without using **new**, which is what we expect from a **case** class. However, the overloaded constructors must be called using **new**, a rather inconsistent result. A **case** class is designed to make life easier by reducing the boilerplate you're forced to write for certain types of common classes. If you try to do too much with a **case** class it doesn't seem to work so well, so you should just drop back to using a normal class instead.

Discussion

1. Remove the curly braces on line 9. What's happening?
2. Uncomment line 19 and observe the result.
3. Comment line 20 and see what happens.

Inheritance

```
1   // Inheritance.scala
2   import com.atomicscala.AtomicTest._
3
4   class Base {
5     def f = 1
6     def g = 2
7     def h = 3
8   }
9
10  class Derived extends Base
11
12  val d = new Derived
13  d.f is 1
14  d.g is 2
15  d.h is 3
16
17  class Derived2 extends Derived {
18    var n = 0 // There's a better way ...
19  }
20
21  class More(arg: Int) extends Derived2 {
22    n = arg * 2
23    override def g = 42
24    override def h = super.h * n + g
25    def u = 3.14
26  }
27
28  val m = new More(3)
29  m.f is 1
30  m.g is 42
31  m.h is 60
32  m.u is 3.14
```

To review: Objects remember things and do stuff. They remember things via fields and they do stuff with operations (typically called methods). Each object occupies a unique place in storage so one object's fields can have different values from every other object.

An object also belongs to a particular category, which we call a class. The class determines the form or template for its objects: the fields and the methods. Thus, all objects look like the class

that created them (via its constructor).

Creating and debugging a class can require a lot of work. What if you want to make a class that is like an existing class but with some variations? It seems like a pity to build a new class from scratch, which is why most object-oriented languages provide a mechanism for reuse called *inheritance*.

Inheritance (which follows the concept of biological inheritance) allows you to say: “I want to make a new class from an existing class, but with some additions and modifications.”

Lines 4-8 define a simple class called **Base**. Note that since none of the methods modify the internal state of the class, we follow the standard style and leave off the parentheses.

To inherit a new class based on an existing class we use the **extends** keyword as seen in Line 10. Here we simply extend the class but don’t actually make any changes, just to show, in lines 13-15, that the new class **Derived** really does inherit all the methods in **Base** (the terms “base class” and “derived class” are often used to describe the inheritance relationship).

Derived2 on lines 17-19 shows that you can add fields during inheritance. In this case we use a **var** because we want to change it during further inheritance, but it turns out that Scala provides a better way that allows us to avoid using **var**, which we shall see later in the book.

In **More** on lines 21-26 the **n** introduced in the **Derived2** is changed, and we also modify the definitions of **f** and **g**. These definition changes require the use of the **override** keyword that we saw in the **toString** atom. Scala insists that you use **override** not only to verify that you aren’t accidentally overriding a method, but also to show anyone reading the code that the method is being overridden. If you accidentally write a method that has the same name as a method in the base class, Scala will emit a message saying that you forgot the **override** keyword, and this will tell you that you were accidentally overriding.

Look closely at line 24. Both the field **n** and the method **g** are accessible without any qualification, but if the overridden method **h** needs to call the base-class version of **h**, there’s a conundrum. If you simply call **h** you’ll be calling the same method you’re currently inside (this is called *recursion*). In order to specify that you want to call the base-class version of **h**, you must use the **super** keyword, as you see in the call to **super.h**. This way the overridden method can reuse the code in the base class. We’ll see later in the book that the **super** keyword has several other uses.

On line 25, the addition of the new method **u** shows that you can add new methods during inheritance; indeed, the keyword **extends** suggests that the class might “get bigger” during inheritance.

Discussion

Inheritance lets you use methods and data from a parent class as if they were defined in this class.

Exercises

1. Create a class that represents a Trip, including departure and arrival times.
2. Create second class that represents an Airplane Trip, including the name of an inflight movie. Create a third class that represents a Car Trip, including a list of cities that you will drive through. Does inheritance simplify the implementation?
3. Can you think of other ways to design these classes?

Base Class Initialization

```
1   // BaseClassInitialization.scala
2   import com.atomicscala.AtomicTest._
3
4   class Base(val f: Int, val g: Int)
5
6   class Derived(f: Int, g: Int, val s: String) extends Base(f, g)
7
8   val d = new Derived(1, 2, "Blue")
9   d.f is 1
10  d.g is 2
11  d.s is "Blue"
12
13  class Base2(val s: String, val d: Double) {
14    def this(s: String) = this(s, 0)
15    def this(d: Double) = this("", d)
16    override def toString = s + ", " + d
17  }
18
19  new Base2("Yay", 11.11) is "Yay, 11.11"
20  new Base2("Ha") is "Ha, 0.0"
21  new Base2(99.99) is ", 99.99"
22
23  class Derived2(s: String, d: Double) extends Base2(s, d)
24  new Derived2("Hey", 2.22) is "Hey, 2.22"
25
26  class Derived3(s: String) extends Base2(s)
27  new Derived3("Now") is "Now, 0.0"
28
29  class Derived4(d: Double) extends Base2(d)
30  new Derived4(47) is ", 47.0"
```

An important aspect of an object-oriented language is the ability to guarantee correct object creation. In the previous atom, we looked at inheritance, but none of the base classes had constructor arguments. If a base class has constructor arguments, then any class that inherits from that base must provide those arguments during construction. This is essential because it ensures that all parts of an object are properly constructed.

If a base class has constructor arguments, the derived-class constructor form is straightforward: you simply include the arguments in the **extends** expression. You can see a very simple form of

this on lines 4 and 6. The class **Base** automatically stores its arguments via the **val** keyword in the argument list. That's all it does so it doesn't need a body. When **Derived** inherits from **Base** it simply passes its appropriate constructor arguments as arguments to the **Base** constructor. Notice that it also adds its own **val** argument -- you aren't limited to the number, type or order of the arguments in the base class. Your only responsibility is to provide the correct base-class arguments.

You've also learned how to make multiple overloaded constructors using the **this** keyword. In a derived class, you can call any of the overloaded base-class constructors via the derived-class primary constructor simply by providing the necessary constructor arguments in the base-class constructor call. You can see this in the definitions of **Base2**, **Derived2**, **Derived3** and **Derived4**. Each of the derived classes exercises a different base-class constructor.

You can't, however, call base-class constructors inside of overloaded derived-class constructors. As before, the primary constructor is the "gateway" for all the overloaded constructors.

Abstract Classes

```
1   // AbstractClasses.scala
2   import com.atomicscala.AtomicTest._
3
4   abstract class Animal {
5     def templateMethod = "The " + animal + " goes " + sound
6     def animal: String // Abstract (no function body)
7     def sound: String  // Abstract
8   }
9
10  // Error: abstract class cannot be instantiated
11  // val a = new Animal
12
13  class Duck extends Animal {
14    def animal = "Duck"
15    override def sound = "Quack" // "override" optional
16  }
17
18  class Cow extends Animal {
19    def animal = "Cow"
20    def sound = "Moo"
21  }
22
23  (new Duck).templateMethod is "The Duck goes Quack"
24  (new Cow).templateMethod is "The Cow goes Moo"
25
26  abstract class Adder(val x: Int) {
27    def add(y: Int): Int
28  }
```

An *abstract class* is just like an ordinary class except that one or more methods has no definition. If you have a class containing methods without definitions, Scala insists that you declare that class using the **abstract** keyword.

Declaring methods without defining them allows you to describe structure without specifying form. The most common use for this is the *template method pattern*. A template method captures common behavior in the base class and relegates details that vary to derived classes.

Suppose we are creating a children's book describing animals and what they say. Each time we're going to do the same thing: produce a statement of the form "The <animal> goes

<sound>." We could easily create a new method in each specific animal class to do this, but that would be duplicated effort *and* if we wanted to change the phrase somehow we'd have to duplicate all the changes (and we might miss some).

The **templateMethod** in class **Animal** captures the common code into one place. Notice that it's completely legal for **templateMethod** to call the **animal** and **sound** methods, even if those methods *haven't been defined yet*. This is safe because Scala will not allow you to make an instance (which is another way of saying "create an object") of an abstract class, as you see on line 11 (try removing the '*//*' and see what happens).

When you define an abstract method, it's important to specify the return type; if you rely on Scala to infer the return type, it will guess "Unit" (that is, "nothing," denoted '**()**') because it doesn't have any better information.

We define **Duck** and **Cow** by extending **Animal** and *only specifying the behavior that varies*. The common behavior has been captured in the base class, in **templateMethod**. Notice that **Duck** and **Cow** are not **abstract** because all their methods now have definitions -- we call such classes *concrete*.

When you provide a definition for an abstract method from a base class, the use of the keyword **override** is optional. Technically, you're *not* overriding because there's no definition to override. In general, when something is optional in Scala, we leave it out because that results in less visual noise.

Since **Duck** and **Cow** are concrete, they can be instantiated, as you can see on lines 23 and 24. Because we are only creating the objects in order to call **templateMethod** on them, we use a shortcut: we don't assign the objects to an identifier. Instead, we surround the **new** expression with parentheses and call **templateMethod** on the resulting (unnamed) object.

Abstract classes can also have parameters, just like ordinary classes. This is demonstrated in **class Adder** on line 26. Since **Adder** is **abstract**, it cannot be instantiated, but any class that inherits from **Adder** can now perform base-class initialization by calling the **Adder** constructor (as you'll see in the exercises).

Exercises

1. Modify **Animal** and its subclasses to also indicate what each animal eats. Add and test **Chicken** and **Pig**.
2. Inherit from the **Adder** class to make it operational, and test the result.

Traits

```
1   // Traits.scala
2   import com.atomicscala.AtomicTest._
3
4   trait Color {
5     def hue: Int
6   }
7
8   trait Texture {
9     def feel: String
10  }
11
12  trait Dimensions {
13    val width: Double
14    val height: Double
15    val depth: Double = 1.0
16    def volume = width * height * depth
17  }
18
19  trait Description {
20    val id: String
21    def hue: Int
22    def feel: String
23    def volume: Double
24    override def toString = id + " " + hue + " " + feel + " " + volume
25  }
26
27  class Toy(val id: String, val hue: Int, val feel: String,
28            val width: Double, val height: Double,
29            override val depth: Double)
30        extends Color with Texture with Dimensions with Description
31
32  new Toy("Unicorn", 93, "Plush", 9, 10, 11) is
33    "Unicorn 93 Plush 990.0"
34
35  class Furniture(val id: String, w: Double, h: Double, d: Double, c: Int)
36        extends Color with Texture with Dimensions with Description {
37    val width = w
38    val height = h
39    override val depth = d
40    def feel = "Hard"
41    def hue = c
42  }
43
```

```
44  new Furniture("End Table", 22, 34, 16, 42) is
45    "End Table 42 Hard 11968.0"
46
47  abstract class PaintingDimensions extends Dimensions {
48    override val depth = 1.5 // Pretend all paintings are this thick
49  }
50
51  trait PaintingTexture extends Texture {
52    def feel = "Dry"
53  }
54
55  class Painting(val width: Double, val height: Double, val id: String,
56                 val hue: Int)
57        extends PaintingDimensions with Color with PaintingTexture
58        with Description
59
60  new Painting(64, 80, "Starry", 111) is "Starry 111 Dry 7680.0"
```

Inheritance looks pretty useful. You can create a new class by modifying an existing class, so you don't have to rewrite everything from scratch.

Inheritance is limited, however: You can only inherit from one class. But it also seems like it could be valuable to assemble a new class from a number of pieces rather than just one. That's where *traits* come in.

Because traits are not limited to single inheritance as classes are, you're able to break them down into pieces as small as you need. Typically, each trait only manages a single aspect of description and/or functionality. For example, you might have one trait that describes color. Because you can put as many traits together as you need, there's no reason to add anything but things about color to the **Color** trait.

A trait definition looks quite similar to an abstract class. Instead of the **class** keyword we use **trait**. Like an abstract class, fields and methods can have definitions or they can be left abstract. One difference is that traits cannot have constructors -- as you'll see, a trait is more of a description of a piece of a class rather than a physical thing that is instantiated.

On line 4, we see a simple trait **Color** which contains a method *declaration* for **hue** (a declaration describes something without providing a *definition* to create storage for a value or code for a method). On line 8 we have a **Texture** trait containing a declaration for a method **feel** returning a **String** to describe that texture.

Dimensions on line 12 is different: we have three **val**s and only two of these have initialization values. Traits cannot have constructors so there is no way to initialize those values. What **Dimensions** is saying is that, after the class has been pasted together, those values must

somehow *exist* in the finished class.

However, we see on lines 15 and 16 that a trait can also provide definitions, both for values and for methods. When you provide definitions within a trait, those definitions are inserted into the class where the trait is included.

Even though **width** and **height** don't exist yet, **volume** can still use them in its calculation.

The goal of **Description** on line 19 is to create a **toString** method that can be used with all the classes we create with these traits, so we don't have to rewrite it for each of those classes. **Description** says that any class where it is used must have an **id**, **hue**, **feel** and **volume** (so they can be used within **toString**).

Now we're ready to combine all these traits into the class **Toy**. Scala requires that, whenever you inherit from a class *or* combine traits, you must use the **extends** keyword for the first item in the list, regardless of whether it's a base class or a trait. So even though **Color** is a trait, because it's the first item in the list we start by using **extends**. All the rest of the traits are added using the **with** keyword, and **with** only works with traits.

By using the **val** keyword in the constructor argument list, **Toy** automatically stores all the constructor arguments. However, **depth** already has a definition inside **Dimensions** so we must use the **override** keyword to tell Scala that we intend to replace that definition with our own -- but we actually place the **override** keyword directly into the argument list.

There's something subtle going on in the definition of **Toy**: some of the trait requirements that are described as methods are being satisfied by providing **val**s instead. For example, **hue** in **Color** is a **def**, but Scala seems to have no problem when we provide a **val** for **hue** in **Toy**. Is this a problem? Well, whenever we call the method **hue**, as in line 24, it's treated as a method that produces an **Int**. And when we reference the **val hue**, it produces an **Int**. It turns out that in Scala, methods without arguments that produce a result can be treated identically to **val**s that produce a result of that same type. There's no harm in treating them the same, and, as in this case, it's often convenient to do so and thus Scala allows it.

Note that all of **Description**'s requirements are fulfilled by **Toy** so its **toString** works and we don't have to write a custom **toString**.

On line 35, the class **Furniture** shows that you are not forced to fulfill the requirements of your traits in the constructor argument list. If you define the appropriate names within the class body (*and* they are the correct type), Scala will still be satisfied. This can, for example, allow for more complex initialization expressions. You can see that **depth** still requires an **override**.

An **abstract** class can inherit from a trait, typically to specify one or more parts of that trait as seen in **PaintingDimensions** on line 47 (**PaintingDimensions** could still be a trait in this case; we're just doing it this way as an example). Also, a trait can inherit from another trait as

with **PaintingTexture** on line 51, which inserts a concrete version of the formerly-abstract **feel**. Because **feel** and **depth** have definitions we don't need to provide anything more for them when we combine all the traits into **Painting**.

This has only been an introduction to traits; there is more depth to the subject than we can cover here. You'll find further details about traits in more advanced books.

Discussion

Inheritance lets you use features from a parent class. For example, the Ordered trait has an abstract method **compare** that you implement to specify the sort order for elements in a Collection, such as a **Vector**.

Exercises

1. Create a case class that represents a Person in an Address Book, complete with Name and Contact Information.
2. Create some "Person" objects and put them in a Vector.
3. Decide what Field in the Person object you might want to sort on and produce a sorted list. Hint:
4. Choose a different Field to sort on. What did you need to change?
5. What if you want to sort on one field (e.g., last name) and resolve any "ties" with a second field (e.g., first name)? What do you have to do differently?

Polymorphism

```
1   // Polymorphism.scala
2   import com.atomicscala.AtomicTest._
3
4   class Element {
5     val id = getClass.getSimpleName.split('$').last
6     def interact(other: Element) = id + " interact " + other.id
7   }
8
9   class Inert extends Element
10  class Wall extends Inert
11
12  trait Material {
13    def resilience
14  }
15  trait Wood extends Material {
16    override def resilience = "Breakable"
17  }
18  trait Rock extends Material {
19    override def resilience = "Hard"
20  }
21  class RockWall extends Wall with Rock
22  class WoodWall extends Wall with Wood
23
24  trait Skill {
25    val id: String
26  }
27  trait Fighting extends Skill {
28    def fight = "Fight!"
29  }
30  trait Digging extends Skill {
31    def dig = "Dig!"
32  }
33  trait Magic extends Skill {
34    def castSpell = "Spell!"
35  }
36  trait Flight extends Skill {
37    def fly = "Fly!"
38  }
39
```

```
40  class Character(var player: String = "None") extends Element
41  class Fairy extends Character with Magic
42  class Viking extends Character with Fighting
43  class Dwarf extends Character with Digging with Fighting
44  class Wizard extends Character with Magic
45  class Dragon extends Character with Magic with Flight
46
47  val d = new Dragon
48  d.player = "Puff"
49  d.interact(new Wall) is "Dragon interact Wall"
50
51  def battle(fighter: Fighting) = fighter.id + ", " + fighter.fight
52  battle(new Viking) is "Viking, Fight!"
53  battle(new Dwarf) is "Dwarf, Fight!"
54  battle(new Fairy with Fighting) is "1, Fight!" // Name: $anon$1
55
56  def fly(flyer: Element with Flight, opponent: Element) = {
57    flyer.id + ", " + flyer.fly + ", " + opponent.interact(flyer)
58  }
59  fly(d, new Fairy) is "Dragon, Fly!, Fairy interact Dragon"
```

Polymorphism is an ancient Greek term that means *many forms*. It turns out that base classes and traits are useful for more than just assembling classes. If we create a class using a subclass **A** and traits **B** and **C**, we can choose to treat that class as if it were only an **A** or only a **B** or only a **C**. For example, if both animals and vehicles can move and you mix in a **Mobile** trait for both of them, you can write a function that takes a **Mobile** argument, and that function will automatically work for both animals and vehicles.

Let's see how this works in Scala. Suppose we want to create some kind of fantasy game. Each game element will draw itself on the screen based on its location in the game world, and when two elements get within proximity of each other, they'll interact. We can sketch out a rough draft that makes use of polymorphism and give you a general idea of how you might design such a game, although we'll leave out the vast majority of implementation details.

Line 5 of **Element** shows a helpful shortcut. Whenever we create a class, we'd like that class to automatically know its own name, so we create an **id** field. We could of course insert the name by hand every time we create a new class, but we'll let Scala do the work for us. However, Scala doesn't have a way to do this, and so we learn a new trick. We can reach down below Scala into Java, because Scala is running on top of Java, and use Java functionality *as if it were just Scala*. Java has a method called **getClass** which produces an object that describes the class. That object has a method called **getSimpleName** which produces a string describing the class name. This string isn't as plain as we'd like; it has some extra stuff separated by '**$**' signs. Without a hitch, we can use Scala's **split** method to break the string across '**$**' signs. The result is an array, and we call **last** to get the end element which is the actual name of the class.

The other method in **Element** is **interact()**, which is how one game element interacts with another when the two are in proximity. **interact()** is going to mean something different depending on the exact types of elements participating in the interaction, and this is an interesting and challenging design problem in itself -- which we shall ignore by just printing out the names of the two interacting elements.

Now we can start creating different types of elements, and traits to mix in to achieve different effects. Note that, just like classes, traits can inherit from each other. When you redefine a base method in a trait, Scala requires the **override** keyword.

Look at the **Skill** trait on line 24. This is designed to be mixed with an **Element** and use the **id** field, so **Skill** contains an abstract **id** which will attach to **Element**'s **id** (We didn't know ahead of time that we wanted to do it this way, but went back and put in the change when we discovered we needed it -- this is often the way design works, as an evolutionary process).

Now we're ready to define some characters which players will actually manipulate. There's something new in the definition of **Character** on line 40: the constructor argument **player** has a *default argument*, specified by the '**=**' after the argument type and the string **"None"**. A default argument is automatically substituted if the caller doesn't provide it. Here, this means that the **player** field always gets **"None"** unless we choose to provide a name. And, since all the arguments for the constructor are defaulted, we don't have to call the base-class constructor during inheritance.

After assembling a number of different **Character** types, we create a **Dragon** on line 47, then on line 48 we change **player** from its default value of **"None"** to **"Puff"**. We can do this because **player** is a **var** rather than our usual **val**, and a **var** can be changed (ordinarily we'd stick with **val** and use base-class constructor calls).

Line 49 is our first example of polymorphism. **Dragon** inherits the **interact** method from **Element**, but **interact** takes an **Element** argument and we're passing it a **Wall**. However, **Wall** ultimately inherits from **Element**, so we can say that "**Wall** is an **Element**" and Scala agrees. The **interact** method takes an **Element** *or anything derived from **Element***. That's polymorphism, and it's very powerful because now any function you write is more general. It can be applied to many more types than just the type you write it for: It can also be applied to anything that inherits from that type. This is transparent and safe because Scala guarantees that a derived class "is a" base class by ensuring that the derived class has (at least) all the methods of the base class.

Line 51 shows a second example, this time using a trait polymorphically. The argument **fighter** happens to be a trait, and this means that any object which happens to incorporate that trait can be safely passed into the **fight** function. The **Fighting** trait has only has one method **fight**, but it also inherits **id** from **Skill** (although **id** is abstract, its "contract" is fulfilled by the **id** in **Element**). This way, **battle** can access both **id** and **fight** in **fighter**, but those are the *only* things it can

access in **fighter** because nothing else is defined in the **Fighting** trait.

Both **Viking** and **Dwarf** include the **Fighting** trait, so they can both be passed into **battle**, which again demonstrates polymorphism. Without polymorphism, you'd have to write specific functions; for example **battle_viking** for **Viking** objects, and **battle_dwarf** for **Dwarf** objects. With polymorphism, you can write one function that not only works with **Viking**s and **Dwarf**s, but with anything else that implements **Fighting**, including types you haven't thought of at the time you write **battle**. Polymorphism is a tool that allows you to write less code and make it more reusable.

As an example of "types you haven't thought of yet," consider line 54, in particular the argument being passed to **battle**:

```
new Fairy with Fighting
```

The type of the object is being created *on the fly!* As we create the object, we combine the existing **Fairy** class with the **Fighting** trait to create a new class, and we immediately make an instance of that class. We didn't give the class a name, so Scala generates one for us: **$anon$1** ("anon" is short for "anonymous") and when **Element**'s **id** encounters this, it produces the '**1**'.

This technique of putting together types just as you need them also works for arguments, as you see on line 56. The first argument, **flyer**, is specified as mixing together **Element with Flight**. Because we already included **id** in the **Skill** mix, **fly** would have worked anyway for **flyer.id** and **flyer.fly**, but for **opponent.interact(flyer)** to work, **flyer** must actually be an **Element**. By saying **Element with Flight**, Scala will ensure that any argument you pass as a **flyer** includes both **Element** and **Flight**, so that **fly** can properly call everything it needs to.

A question often arises: "How did you know to do it this way?" This is the design challenge. Once you decide what you want to build, there are many different ways to actually put it together. You can create a base class and add new methods during inheritance, or you can mix in functionality using traits. These are design decisions that you make using a combination of experience and observing the way your system is being used. You must decide what makes sense, based on the requirements of your system. That's the design process.

Exercises

1. Add a **draw** method to **Element** and ensure that all the objects in the system can **draw**.