CHAPMAN & HALL/CRC
TEXTBOOKS IN COMPUTING

INTRODUCTION TO PROGRAMMING AND PROBLEM-SOLVING USING SCALA

SECOND EDITION

Mark C. Lewis
Lisa L. Lacher

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business

A CHAPMAN & HALL BOOK

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

Printed on acid-free paper
Version Date: 20160607

International Standard Book Number-13: 978-1-4987-3095-2 (Pack - Book and Ebook)

Visit the Taylor & Francis Web site at
http://www.taylorandfrancis.com

and the CRC Press Web site at
http://www.crcpress.com

Contents

Chapter 2

Scala Basics

It is time to begin our journey learning how to program with the Scala language. You can download Scala for free from `http://www.scala-lang.org` to run on Windows, Mac, or Linux (see the inset below for full instructions on how to install). In this book, we will use the command line to run Scala. If you do not have experience with the command line on your machine, you can refer to Appendix A for a brief introduction. Before looking at the language itself, we need to talk a bit about tools so that you can play along.

2.1 Scala Tools

After you have installed Scala on your machine there are several different programs that get installed in the `bin` directory under the Scala installation. To begin with, we will

only concern ourselves with one of these: `scala`.[1] The `scala` command actually runs `scala` programs. There is a second command, `scalac`, that is used to compile `scala` text files into bytecode that is compatible with either the Java or .NET platform. We will only use `scalac` in the last chapter of this book, but we will begin using the `scala` command immediately.

Scala on your Machine

If you only use Scala on a machine in a computer lab, hopefully everything will have been set up for you so that you can simply type the name of a command and it will run. To run Scala on your own machine you can follow the instructions below.

Installation

Scala requires Java® to run so if you do not have Java installed you should go to `http://java.oracle.com` and download then install the most recent version of the Java SE JDK. When you install Java, you can go with the default install locations.

After you have Java installed you can install Scala. To download Scala go to `http://www.scala-lang.org`. On that site download the latest version of Scala. The code in this book was written to work with Scala 2.12.

Dealing with the `PATH`

If you are using Scala on your own machine, it is possible that entering `scala` or `scala.bat` on the command line could produce a message telling you that the command or program `scala` could not be found. This happens because the location of the installed programs are not in your default `PATH`.

The `PATH` is a set of directories that are checked whenever you run a command. The first match that is found for any executable file in a directory in the `PATH` will be run. If none of the programs in the `PATH` match what you entered, you get an error.

When you installed Scala, a lot of different stuff was put into the install directory. That included a subdirectory called "`bin`" with different files in it for the different executables. If you are on a Windows machine, odds are that you installed the program in `C:\Program Files (x86)\scala` so the `scala.bat` file that you want to run is in `C:\Program Files\scala\bin\scala.bat`. You can type in that full command or you can add the `bin` directory to your `PATH`. To do this go to Control Panel, System and Security, Advanced System Settings, Environment Variables, and edit the path to add `C:\Program Files\scala\bin` to the path.

Under Unix/Linux you can do this from the command line. Odds are that Scala was installed in a directory called `scala` in your user space. To add the `bin` directory to your path you can do the following:

```
export PATH=$PATH:/home/username/scala/bin
```

Replace "`username`" with your username. This syntax assumes you are using the bash shell. If it does not work for you, you can do a little searching on the web for how to add directories to your path in whatever shell you are running. To make it so that you do not have to do this every time you open a terminal, add that line to the appropriate configuration file in your home directory. If you are running the Bash shell on Linux this would be `.bashrc`.

[1] On a Windows system this commands should be followed by "`.bat`".

There are three ways in which the `scala` command can be used. If you just type in `scala` and press enter you will be dropped into the Scala REPL (Read-Execute-Print Loop). This is an environment where you can type in single Scala expressions and immediately see their values. This is how we will start off interacting with Scala, and it is something that we will come back to throughout the book because it allows us to easily experiment and play around with the language. The fact that it gives us immediate feedback is also quite helpful.

To see how this works, at the command prompt, type in `scala` and then press enter. It should print out some information for you, including telling you that you can get help by typing in `:help`. It will then give you a prompt of the form `scala>`. You are now in the Scala REPL. If you type in `:help` you will see a number of other commands you could give that begin with a colon. At this time the only one that is significant to us is `:quit` which we will use when we are done with the REPL and want to go back to the normal command prompt.

It is customary for the first program in a language to be Hello World. So as not to break with tradition, we can start by doing this now. Type the following after the `scala>` prompt.

```
println("Hello, World!");
```

If you do this you will see that the next line prints out "Hello, World!". This exercise is less exciting in the REPL because it always prints out the values of things, but it is a reasonable thing to start with. It is worth asking what this really did. `println` is a function in Scala that tells it to print something to standard output and follow that something with a newline character[2] to go to the next line. In this case, the thing that was printed was the string "Hello, World!". You can make it print other things if you wish. One of the advantages of the REPL is that it is easy to play around in. Go ahead and test printing some other things to see what happens.

The second usage of the `scala` command is to run small Scala programs as scripts. The term script is generally used to refer to short programs that perform specific tasks. There are languages that are designed to work well in this type of usage, and they are often called scripting languages. The design of Scala makes it quite usable as a scripting language. Unlike most scripting languages, however, Scala also has many features that make it ideal for developing large software projects as well. To use Scala for scripting, simply type in a little Scala program into a text file that ends with "`.scala`"[3] and run it by putting the file name after the `scala` command on the command line. So you could edit a file called `Hello.scala` and add the line of code from above to it. After you have saved the file, go to the command line and enter "`scala Hello.scala`" to see it run.

2.2 Expressions, Types, and Basic Math

All programming languages are built from certain fundamental parts. In English you put together words into phrases and then combine phrases into sentences. These sentences can be put together to make paragraphs. To help you understand programming, we will make analogies between standard English and programming languages. These analogies are

[2] You will find more information about the and other escape characters in section 2.8

[3] It is not technically required that your file ends with "`.scala`", but there are at least two good reasons you should do this. First, humans benefit from standard file extensions because they have meaning and make it easier to keep track of things. Second, some tools treat things differently based on extensions. For example, some text editors will color code differently based on the file extension.

not perfect. You cannot push them too far. However, they should help you to organize your thinking early in the process. Later on, when your understanding of programming is more mature, you can dispense with these analogies as you will be able to think about programming languages in their own terms.

The smallest piece of a programming language that has meaning is called a TOKEN. A token is like a word or punctuation mark in English. If you break up a token, you change the meaning of that piece, just like breaking up a word is likely to result in something that is no longer a word and does not have any meaning at all. Indeed, many of the tokens in Scala are words. Other tokens are symbols like punctuation. Let's consider the "Hello, World" example from the previous section.

```
println("Hello, World!");
```

This line contains a number of tokens: `println`, `(`, `"Hello, World!"`, and `)`.

When you think of putting words together, you probably think of building sentences with them. A sentence is a grouping of words that stands on its own in written English. The equivalent of a sentence in Scala, and most programming languages, is the STATEMENT. A statement is a complete and coherent instruction that we can give the computer. When you are entering "commands" into the REPL, they are processed as full statements. If you enter something that is not a complete statement in the REPL, instead of the normal prompt, you will get a vertical bar on the next line telling you that you need to continue the statement. The command listed above is a complete statement which is why it worked the way it did.

Note that this statement ends with a semicolon. In English you are used to ending sentences with a period, question mark, or exclamation point. Scala follows many other programming languages in that semicolons denote the end of a statement. Scala also does something called semicolon inference. Put simply, if a line ends in such a way that a semicolon makes sense, Scala will put one there for you. As a result of this, our print statement will work just as well without the semicolon.

```
println("Hello World!")
```

You should try entering this into the REPL to verify that it works. Thanks to the semicolon inference in Scala, we will very rarely have to put semicolons in our code. One of the few times they will really be needed is when we want to put two statements on a single line for formatting reasons.

While you probably think of building sentences from words in English, the reality is that you put words together into phrases and then join phrases into sentences. The equivalent of a phrase in Scala is the EXPRESSION. Expressions have a far more significant impact on programming languages than phrases have in English, or at the least programmers need to be more cognizant of expressions than English writers have to be of phrases. An expression is a group of tokens in the language that has a value and a TYPE.[4] For example, 2 + 2 is an expression which will evaluate to 4 and has an Integer type.

Just like some phrases are made from a single word, some tokens represent things that have values on their own, and, as such, they are expressions themselves. The most basic of these are what are called LITERALS. Our sample line was not only a statement, it was also an expression. In Scala, any valid expression can be used as a statement, but some statements are not expressions. The `"Hello, World!"` part of our statement was also an expression. It is something called a string literal which we will learn more about in section 2.4.

Let us take a bit of time to explore these concepts in the REPL. Run the `scala` command

[4]Type is a construct that specifies a set of values and the operations that can be performed on them. Common types include numeric integer, floating-point, character, and boolean.

without any arguments. This will put you in the REPL with a prompt of `scala>`. In the last chapter we typed in a line that told Scala to print something. This was made from more than one token. We want to start simpler here. Type in a whole number, like `5`, followed by a semicolon and hit enter. You should see something like this:

```
scala> 5;
res0: Int = 5
```

The first line is what you typed in at the prompt. The second line is what the Scala REPL printed out as a response. Recall that REPL stands for Read-Evaluate-Print Loop. When you type something in, the REPL reads what you typed, then evaluates it and prints the result. The term loop implies that this happens over and over. After printing the result, you should have been given a new prompt.

So what does this second line mean? The REPL evaluated the statement that you input. In this case, the statement is just an expression followed by a semicolon and the REPL was printing out the value of the expression you entered. As was mentioned above, the REPL needs you to type in full statements so that it can evaluate it. In this case, we typed in a very simple statement that has an expression called a NUMERIC LITERAL followed by a semicolon. This semicolon will be inferred if you do not add it in. We will take advantage of that and leave them out of statements below.

The end of the output line gives us the value of the expression which is, unsurprisingly, 5. What about the stuff before that? What does `res0: Int` mean? The `res0` part is a name. It is short for "result0". When you type in an expression as a statement in the Scala REPL as we did here, it does not just evaluate it, it gives it a name so that you can refer back to it later. The name `res0` is now associated with the value 5 in this run of the REPL. We will come back to this later. For now we want to focus on the other part of the line, `:Int`. Colons are used in Scala to separate things from their types. We will see a lot more of this through the book, but what matters most to us now is the type, `Int`. This is the type name that Scala uses for basic numeric integers. An integer can be either a positive or negative whole number. You can try typing in a few other integer values to see what happens with them. Most of the time the results will not be all that interesting, but if you push things far enough you might get a surprise or two.

What happens if you type in a number that is not an integer? For example, what if you type in 5.6? Try it, and you should get something like this:

```
scala> 5.6
res1: Double = 5.6
```

We have a different name now because this is a new result. We also get a different type. Instead of `Int`, Scala now tells us that the type is `Double`. In short, `Double` is the type that Scala uses by default for any non-integer numeric values. Even if a value technically is an integer, if it includes a decimal point, Scala will interpret it to be a `Double`. You can type in `5.0` to see this in action. Try typing in some other numeric values that should have a `Double` as the type. See what happens. Once again, the results should be fairly mundane. Double literals can also use scientific notation by putting the letter `e` between a number and the power of ten it is multiplied by. So `5e3` means $5 * 10^3$ or 5000.

So far, all of the expressions we have typed in have been single tokens. Now we will build some more complex expressions. We will begin by doing basic mathematical operations. Try typing in "`5+6`".

```
scala> 5+6
res2: Int = 11
```

This line involves three tokens. Each character in this case is a separate token. If you space things out, it will not change the result. However, if you use a number with multiple digits, all the digits together are a single token and inserting spaces does change the meaning.

There should not be anything too surprising about the result of `5+6`. We get back a value of `11`, and it has a type of `Int`. Try the other basic arithmetic operations of `-`, `*`, and `/`. You'll notice that you keep getting back values of type `Int`. This makes sense for addition, subtraction, and multiplication. However, the result of `5/2` might surprise you a little bit. You normally think of this expression as having the value of `2.5` which would be a `Double`. However, if you ask Scala for the result of `5/2` it will tell you the value is the `Int 2`. Why is this, and what happened to the `0.5`? When both operands are of type `Int`, Scala keeps everything as `Int`s. In the case of division, the decimal answer you might expect is truncated and the fractional part is thrown away. Note that it is not rounded, but truncated. Why is this? It is because in integer arithmetic, the value of `5/2` is not `2.5`. It is `2r1`. That is to say that when you divide five by two, you get two groups of two with one remainder. At some point in your elementary education, when you first learned about division, this is probably how you were told to think about it. At that time you only had integers to work with so this is what made sense.

Scala is just doing what you did when you first learned division. It is giving you the whole number part of the quotient with the fractional part removed. This fractional part is normally expressed as a remainder. There is another operation called modulo that is represented by the percent sign that gives us the remainder after division. Here we can see it in action.

```
scala> 5%2
res3: Int = 1
```

The modulo operator is used quite a bit in computing because it is rather handy for expressing certain ideas. You should take some time to re-familiarize yourself with it. You might be tempted to say that this would be your first time dealing with it, but in reality, this is exactly how you did division yourself before you learned about decimal notation for fractions.

What if you really wanted `2.5` for the division? Well, `2.5` in Scala is a `Double`. We can get this by doing division on `Double`s.

```
scala> 5.0/2.0
res4: Double = 2.5
```

All of our basic numeric operations work for `Double`s as well. Play around with them some and see how they work. You can also build larger expressions. Put in multiple operators, and use some parentheses.

What happens when you combine a `Double` and an `Int` in an expression. Consider this example:

```
scala> 5.0/2
res5: Double = 2.5
```

Here we have a `Double` divided by an `Int`. The result is a `Double`. When you combine numeric values in expressions, Scala will change one to match the other. The choice of which one to change is fairly simple. It changes the one that is more restrictive to the one that is less restrictive. In this case, anything that is an `Int` is also a `Double`, but not all values that are `Double`s are `Int`s. So the logical path is to make the `Int` into a `Double` and do the operation that way.

2.3 Objects and Methods

One of the features of the Scala language is that all the values in Scala are OBJECTs. The term object in reference to programming means something that combines data and the functionality on that data in a single entity. In Scala we refer to the things that an object knows how to do as METHODs. The normal syntax for calling a method on an object is to follow the object by a period (which we normally read as "dot") and the name of the method. Some methods need extra information, which we called arguments. If a method needs ARGUMENTS then those are put after the method name in parentheses.

In Scala, even the most basic literals are treated as objects in our program, and we can therefore call methods on them. An example of when we might do this is when we need to convert one type to another. In the sample below we convert the `Double` value `5.6` into an `Int` by calling the `toInt` method. In this simple context we would generally just use an `Int` literal, but there will be situations we encounter later on where we are given values that are `Double`s and we need to convert them to `Int`s. We will be able to do that with the `toInt` method.

```
scala> 5.6.toInt
res6: Int = 5
```

One thing you should note about this example is that converting a `Double` to an `Int` does not round. Instead, this operation performs a truncation. Any fractional part of the number is cut off and only the whole integer is left.

We saw at the beginning of this chapter that Scala is flexible when it comes to the requirement of putting semicolons at the end of statements. Scala will infer a semicolon at the end of a line if one makes sense. This type of behavior makes code easier to write.

Methods that take one argument can be called using "infix" notation. This notation leaves off the dot and parentheses, and simply places the method between the object it is called on and the argument. If the method name uses letters, spaces will be required on either side of it. This type of flexibility makes certain parts of Scala more coherent and provides the programmer with significant flexibility. Though you did not realize it, you were using "infix" notation in the last section. To see this, go into Scala and type "`5.`" then press tab. The Scala REPL has tab completion just like the command line, so what you see is a list of all the methods that could be called on the `Int`. It should look something like the following.

```
scala> 5.
%   +   >    >>>           isInstanceOf  toDouble  toLong    unary_+  |
&   -   >=   ^             toByte        toFloat   toShort   unary_-
*   /   >>   asInstanceOf  toChar        toInt     toString  unary_~
```

You have already seen and used some of these methods. We just finished using `toInt` on a `Double`. We can call `toDouble` on an `Int` as well. The things that might stand out though are the basic math operations that were used in the previous section. The `+`, `-`, `*`, `/`, and `%` we used above are nothing more than methods on the `Int` type. The expression `5+6` is really `5 .+ (6)` to Scala. In fact, you can type this into Scala and see that you get the same result.

```
scala> 5 .+ (6)
res7: Int = 11
```

The space between the `5` and the `.` is required here because without it Scala thinks you want a `Double`. You could also make this clear using parentheses by entering `(5).+(6)`.

So when you type in `5+6`, Scala sees a call to the method `+` on the object `5` with one argument of `6`. We get to use the short form simply because Scala allows both the dot and the parentheses to be optional in cases like this.

2.4 Other Basic Types

Not everything in Scala is a number. There are other non-numeric types in Scala which also have literals. We will start simple and move up in complexity. Perhaps the simplest type in Scala is the `Boolean` type. Objects of the `Boolean` type are either `true` or `false`, and those are also valid literals for `Boolean`s.

```
scala> true
res8: Boolean = true

scala> false
res9: Boolean = false
```

We will see a lot more on `Boolean`s and what we can do with them in chapter 3 when we introduce Boolean logic.

Another type that is not explicitly numeric is the `Char` type. This type is used to represent single characters. We can make character literals by placing the character inside of single quotes like we see here.

```
scala> 'a'
res10: Char = a
```

The way that computers work, all character data is really numbers, and different numbers correspond to different characters. We can find out what numeric value is associated with a given character by using the `toInt` method. As you can see from the line below, the lowercase "a" has a numeric value of `97`.

```
scala> 'a'.toInt
res11: Int = 97
```

Because characters have numeric values associated with them, we can also do math with them. When we do this, Scala will convert the character to its numeric value as an `Int` and then do the math with the `Int`. The result will be an `Int`, as seen in this example.

```
scala> 'a'+1
res12: Int = 98
```

In the last section you might have noticed that the `Int` type has a method called `toChar`. We can use that to get back from an integer value to a character. You can see from the following example that when you add `1` to `'a'` you get the logical result of `'b'`.

```
scala> ('a'+1).toChar
res13: Char = b
```

An object of the `Char` type can only be a single character. If you try to put more than one character inside of single quotes you will get an error. It is also an error to try to make a `Char` with empty single quotes. However, there are lots of situations when you want to be able to represent many characters, or even zero characters. This includes words, sentences, and many other things. For this there is a different type called a `String`. String literals are formed by putting zero or more characters inside of double quotes like we see in this example.

```
scala> "Scala is a programming language"
res14: String = Scala is a programming language
```

Notice that the type is listed as `String`.[5]

Certain operations that look like mathematical operations are supported for `String`s. For example, when you use `+` with `String`s, it does string concatenation. That is to say it gives back a new string that is the combined characters of the two that are being put together as shown here:

```
scala> "abc"+"def"
res15: java.lang.String = abcdef
```

This type of operation works with other types as well. The next example shows what happens when we concatenate a `String` with an `Int`. The `Int` is converted to a `String`, using the `toString` method, and normal string concatenation is performed.

```
scala> "abc"+123
res16: java.lang.String = abc123
```

This works whether the `String` is the first or second argument of the `+`.

```
scala> 123+"abc"
res17: java.lang.String = 123abc
```

In addition to concatenation, you can multiply a string by an integer, and you will get back a new string that has the original string repeated the specified number of times.

```
scala> "abc"*6
res18: String = abcabcabcabcabcabc
```

This can be helpful for things such as padding values with the proper number of spaces to make a string a specific length. You can do this by "multiplying" the string `" "` by the number of spaces you need.

The infix notation for calling a method was introduced earlier. We can show another example of this using the `String` type and the `substring` method. As the name implies, `substring` returns a portion of a `String`. There are two versions of it. One takes a single `Int` argument and returns everything from that index to the end of the `String`. Here you can see that version being called using both the regular notation and the infix notation.

```
scala> "abcd".substring(2)
res19: String = cd

scala> "abcd" substring 2
res20: String = cd
```

[5]The way you are running this, the real type is a `java.lang.String`. Scala integrates closely with Java and uses some of the Java library elements in standard code. This also allows you to freely call code from the Java libraries, a fact that has been significant in the adoption of Scala.

The indices in Strings begin with zero, so 'a' is at index 0, 'b' is at index 1, and 'c' is at index 2. Calling substring with an argument of 2 gives back everything from the 'c' to the end.

The version of substring that takes two arguments allows us to demonstrate a different syntax where we just leave off the dot. In this case, the first argument is the first index to take and the second one is one after the last index to take. In math terms, the bounds are inclusive on the low end and exclusive on the high end. The fact that there are two arguments means that we have to have parentheses to group together the two arguments, However, we are not required to put the dot, and the method name can just be between the object and the arguments. Here are examples using the normal syntax and the version without the dot.

```
scala> "abcd".substring(1,3)
res21: String = bc

scala> "abcd" substring (1,3)
res22: String = bc
```

The space between the method and the parentheses is not required. Remember that in general Scala does not care about spaces as long as they do not break up a token. In this book, we will typically use the standard method calling notation, but you should be aware that these variations exist.

There are other types that are worth noting before we move on. One is the type Unit. The Unit type in Scala basically represents a value that carries no information.[6] There is a single object of type Unit. It is written in code and prints out as (). We have actually seen an example of code that uses Unit. The first program we saw in this chapter used a function called println. When we called println Scala did something (it directed the string to standard output), but did not give us back a value. This is what happens when we type in an expression that gives us back a value of Unit in the REPL.

Another significant type in Scala is the TUPLE. A tuple is a sequence of a specified number of specific types. Basically, a collection of values that is strict about how many and what type of values it has. We can make tuples in Scala by simply putting values in parentheses and separating them with commas as seen in the following examples.

```
scala> (5,6,7)
res23: (Int, Int, Int) = (5,6,7)

scala> ("book",200)
res24: (String, Int) = (book,200)

scala> (5.7,8,'f',"a string")
res25: (Double, Int, Char, String) = (5.7,8,f,a string)
```

The tuples in Scala provide a simple way of dealing with multiple values in a single package, and they will come up occasionally through the book. Note that the way we express a tuple type in Scala is to put the types of the values of the tuple in parentheses with commas between them, just like we do with the values to make a tuple object.

Tuples with only two elements can have special meanings in some parts of Scala. For that reason, there is an alternate syntax you can use to define these. If you put the token -> between two values, it will produce a 2-tuple with those values. Consider the following example.

[6] The equivalent in many other languages is called void.

```
scala> 3 -> "three"
res26: (Int, String) = (3,three)
```

The -> will only produce tuples with two elements though. If you try using it with more than two elements you can get interesting results.

```
scala> 4 -> 5 -> 6
res27: ((Int, Int), Int) = ((4,5),6)
```

So if you want tuples with more than two elements, stick with the parentheses and comma notation.

Once you have a tuple, there are two ways to get things out of them. The first is to use methods named _1, _2, _3, etc. So using res21 from above we can do the following.

```
scala> res25._1
res28: Double = 5.7

scala> res25._3
res29: Char = f
```

The challenge with this method is that method names like _1 are not very informative and can make code difficult to read. We will see an alternative approach in section 2.7 that requires a bit more typing, but can produce more readable code.

2.5 Back to the Numbers

Depending on how much you played around with the topics in section 2.2 you might or might not have found some interesting surprises where things behaved in ways that you were not expecting. Consider the following:

```
scala> 1500000000+1500000000
res30: Int = -1294967296
```

Mathematicians would consider this to be the wrong answer. It is actually a reflection of the way that numbers are implemented on computers. The details of this implementation can impact how your programs work, so it is worth taking a bit of time to discuss it.

At a fundamental level, all information on computers is represented with numbers. We saw this with the characters being numbers. On modern computers all these numbers are represented in BINARY, or base two which represents numeric values using two different symbols: 0 (zero) and 1 (one). The electronics in the computer alternate between two states that represent 1 and 0 or on and off. Collections of these represent numbers. A single value of either a 0 or a 1 is called a BIT. It is a single digit in a binary number. The term BYTE refers to a grouping of 8 bits which can represent 256 different numbers. In Scala these will be between -128 and 127. To understand this, we need to do a little review of how binary numbers work.

You have likely spent your life working with decimal numbers, or base ten. In this system, there are ten possible values for each digit: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. Digits in different positions represent different power of ten. So the number 365 is really $3*10^2+6*10^1+5*10^0$. There is nothing particularly unique about base ten other than perhaps it relates well to the number of digits on human hands. You can just as well use other bases, in which case

Value	Power of 2	Digit
296	256	1
40	128	0
40	64	0
40	32	1
8	16	0
8	8	1
0	4	0
0	2	0
0	1	0

FIGURE 2.1: Illustration of the conversion from decimal to binary using the subtraction method. This method works from the top down. To get the number in binary just read down the list of digits.

you need an appropriate number of symbols for each digit and each position represents a power of that base.

Binary uses a base of two. In binary we only need two different digits: 0 and 1. This is convenient on computers where the electronics can efficiently represent two states. The different positions represent powers of two: 1, 2, 4, 8, 16, 32, ... So the number $110101 = 1 * 32 + 1 * 16 + 0 * 8 + 1 * 4 + 0 * 2 + 1 * 1 = 53$. This example shows how you convert from binary to decimal. Simply add together the powers of two for which the bits have a value of one. A byte stores eight bits that would represent powers of two from 128 down to 1. The word "would" is used here because there is a significant nuance to this dealing with negative numbers that we will discuss shortly.

There are two basic approaches to converting from decimal to binary. One involves repeated subtraction of powers of two while the other involves repeated division by two. We will start with the first one and use the value 296 in decimal for the conversion. We start by finding the largest power of 2 that is smaller than our value. In this case it is $256 = 2^8$. So we will have a one in the 2^8 position or the 9^{th} digit[7]. Now we subtract and get $296 - 256 = 40$ and repeat. The largest power of 2 smaller than 40 is $32 = 2^5$. So the digits for 2^7 and 2^6 are 0. Subtract again to get $40 - 32 = 8$. We now have $8 = 2^3$ so the final number in binary is 100101000. This procedure is written out the way you might actually do it in figure 2.1.

The other approach is a bit more algorithmic in nature and is probably less prone to error. It works based on the fact that in binary, multiplying and dividing by 2 moves the "binary point" the same way that multiplying or dividing by 10 moves the decimal point in the decimal number system. The way it works is you look at the number and if it is odd you write a 1. If it is even you write a 0. Then you divide the number by 2, throwing away any remainder or fractional part, and repeat with each new digit written to the left of those before it. Do this until you get to 0. You can also think of this as just dividing by two repeatedly and writing the remainder as a bit in the number with the quotient being what you keep working with.

The number 296 is even so we start off by writing a 0 and divide by 2 to get 148. That is also even so write another 0. Divide to get 74. This is also even so write another 0. Divide to get 37. This is odd so write a 1. Divide to get 18, which is even so you write a 0. Divide to get 9 and write a 1. Divide to get 4 and write a 0. Divide to get 2 and write a 0. Divide to get 1 and write that one. The next division gives you zero so you stop. This procedure is illustrated in figure 2.2.

[7]Remember that the first digit is $2^0 = 1$.

```
Value         Digit
   1             1
   2             0
   4             0
   9             1
  18             0
  37             1
  74             0
 148             0
 296             0
```

FIGURE 2.2: Illustration of the conversion from decimal to binary using the repeated division method. This method works from the bottom up so you get the bits in the result starting with the smallest.

2.5.1 Binary Arithmetic

Now that you know how to go from binary to decimal and decimal to binary, let's take a minute to do a little arithmetic with binary numbers. It is certainly possible to do this by converting the binary to decimal, doing the arithmetic in decimal, then converting back to binary. However, this is quite inefficient and not worth it because it really is not hard to work in binary. If anything, it is easier to work in binary than in decimal. Let us begin with the operation of addition. Say we want to add the numbers 110101 and 101110. To do this you do exactly what you would do with long addition in decimal. As with decimal numbers, you start by adding the bits one column, at a time, from right to left. Just as you would do in decimal addition, when the sum in one column is a two-bit number, the least significant part is written down as part of the total and the most significant part is "carried" to the next left column. The biggest difference between decimal and binary addition is that in binary there is a lot more carrying. Here is a problem solved without showing the carries.

```
  110101
+ 101110
 -------
 1100011
```

Here is the same problem, but with numbers written above to show when there is a carry.

```
 1111
  110101
+ 101110
 -------
 1100011
```

Multiplication in binary can also be done just like in decimal, and you have a lot fewer multiplication facts to memorize. Zero times anything is zero and one times anything is that number. That is all we have to know. Let us do multiplication with the same numbers we just worked with. First we will get all the numbers that need to be added up.

```
      110101
    * 101110
 -----------
     1101010
```

```
   11010100
  110101000
11010100000
```

Adding these numbers is best done in pairs. The reason is that as soon as you add together 3 or more numbers in binary you have the capability to have to do something you are not accustomed to doing in decimal: carry a value up two digits. In decimal you would have to have a column sum up to one hundred or more for this to happen. However, in binary you only have to get to four (which is written as 100 in binary). That happens in this particular instance in the 6th digit. To reduce the odds of an error, it is better to add the values two at a time as we have shown here.

```
    1101010
  + 11010100
 -----------
   100111110
 + 110101000
 -----------
  1011100110
+11010100000
 -----------
100110000110
```

You can do division in the same way that you do long division with integers, but we will not cover that here.

2.5.2 Negative Numbers in Binary

We still have not addressed the question of how we represent negative numbers on a computer. The description that we have given so far only deals with positive values. Numbers that are interpreted this way are called UNSIGNED. All the numeric types in Scala are SIGNED, so we should figure out how that works.[8] To do this, there are two things that should be kept in mind. The first is that our values have limited precision. That is to say that they only store a certain number of bits. Anything beyond that is lost. The second is that negative numbers are defined as the additive inverses of their positive counterparts. In other words, $x + (-x) = 0$ for any x.

To demonstrate how we can get negative numbers, let's work with the number 110101 (53 in decimal). Unlike before, we will now limit ourselves to a single byte. So, we have 8 digits to work with, and the top digits are zeros. Our number stored in a byte is really `00110101`. So the question of what should be the negative is answered by figuring out what value we would add to this in order to get zero.

```
  00110101
+ ????????
  --------
  00000000
```

Of course, there is nothing that we can put into the question marks to make this work. However, if we go back to our first fact (i.e. values have limited precision) we can see what we must do. Note that our total below has 9 digits. We do not need the total to be zero, we need eight digits of zero. So in reality, what we are looking for is the following.

[8] The `Char` is actually a 16-bit unsigned numeric value, but the normal numeric types are all signed.

Type	Bits	Min	Max
Byte	8	-128	127
Short	16	-32768	32767
Int	32	-2147483648	2147483647
Long	64	-9223372036854775808	9223372036854775807

TABLE 2.1: Integer types with their sizes and ranges.

```
  00110101
+ ????????
  --------
 100000000
```

This problem is solvable and the most significant `1`, the one on the far left, will be thrown away because we can only store 8 bits in a byte. So the answer is given here.

```
  00110101
+ 11001011
  --------
 100000000
```

Note that the top bit is "on" in the negative value. The top bit is not exactly a sign bit, but if a number is signed, the top bit will tell us quickly whether the number is positive or negative. This style of making negatives is called TWO'S COMPLIMENT. In the early days of digital computing other options were tried, such as adding a sign-bit or a method called ones' compliment where the bits are simply flipped. However, two's compliment is used in machines today because it allows numeric operations to be done with negative numbers using the same circuitry as is used for positive numbers.

This process gives us the correct answer and is based on the proper definition of what a negative number is. Finding negatives using the definition of what a negative value is works and can be a fallback, but there is a simpler method. To get the two's compliment negative of a binary number of any size, simply flip all the bits and add one. You can verify that this approach works for our example above. It is left as an exercise for the student to figure out why this works.

2.5.3 Other Integer Types

There are larger groups of bits beyond the 8-bit bytes that have meaning in Scala. In fact, if you go back to section 2.3 and you look at the different methods on an `Int`, you will see that `toDouble` and `toChar` are not the only conversions we can do. Scala has other integer types called `Byte`, `Short`, and `Long`. A `Byte` in Scala is an 8-bit number. A `Short` is a 16-bit number. The `Int` that we have been using is a 32-bit number. The `Long` type is a 64-bit number. The reason for the odd behavior that was demonstrated at the beginning of section 2.5 is that we added two numbers together whose sum is bigger than what can be stored in the lower 31 bits of an `Int` and the OVERFLOW, as it is called, wrapped it around to a negative value. Table 2.1 shows the minimum and maximum values for each of the different integer types.

Occasionally you will need to use literals that are bigger than what an `Int` can store. You can do this with a `Long`. Making a numeric literal into a `Long` is done by simply adding an `L` to the end. You can see this here.

```
scala> 5000000000L
```

```
res31: Long = 5000000000
```

The value five billion is not a valid `Int`. If you leave off the `L` here you get an error. The `L` can be lower case, but then it looks a lot like the number one so it is better to use the upper case.

We talked about binary above, and Scala has a method that will let you see the binary form of a number. This method works on the four normal numeric types and `Char`. Here we use it to see the binary representation for 83 and -83 for the values as both `Int` and `Long` types.

```
scala> 83.toBinaryString
res32: String = 1010011

scala> -83.toBinaryString
res33: String = 11111111111111111111111110101101

scala> 83L.toBinaryString
res34: String = 1010011

scala> -83L.toBinaryString
res35: String =
  1111111111111111111111111111111111111111111111111111111110101101
```

The `toBinaryString` method does not display leading zeros, so the positive values only show seven digits in both formats. However, the negative form has many leading ones and all of these are printed.

2.5.4 Octal and Hexadecimal

Binary is what the machine uses, but it really is not that useful to humans. This is in large part due to the fact that the number of digits in a binary number is often large, even if the number itself is not what we consider large. There are two other bases that are commonly seen in programming and dealing with computers. They are base 8, OCTAL, and base 16, HEXADECIMAL or HEX. Like decimal, these bases allow you to represent fairly large numbers with relatively few digits. Unlike decimal, converting from octal or hex to binary and back is trivial. The reason for this is that 8 and 16 are powers of two.

To see this, let us start with octal. When working in base 8, the digits can be between `0` and `7` with each subsequent digit being a higher power of 8. The ease of converting to and from binary comes from the fact that 8 is 2^3. In binary the values `0` to `7` are represented with three bits between `000` and `111`. The fourth and subsequent bits represent values that are multiples of eight. Because of this, we can convert a binary number to an octal number by grouping the bits into groups of three, starting with the least significant bit, and converting those groups.[9] So the binary number, `1010011` is `123` in octal. The lowest three bits, `011`, convert to `3`, the next three, `010`, convert to `2`, and the top bit is just a `1`. We can use the `toOctalString` method to confirm this.

```
scala> 83.toOctalString
res36: String = 123
```

To go the other way, from octal to binary, we simply convert the octal digits to three digit

[9]It is very important to start grouping with the ones bit. Starting at the other end will give you the wrong answer if the last group has fewer than three bits.

binary numbers. So the octal value, `3726` converts to `011111010110`. We can emphasize the groupings of bits by spacing them out: `011 111 010 110`. This is `2006` in decimal.

Moving between hexadecimal and binary is similar. The catch is that now a single digit needs to have 16 possible values. So the `0-9` that we are used to will not suffice. It is typical to augment the normal digits with the letters `A-F` where `A` is 10 and `F` is 15. Because $16 = 2^4$, we use groups of 4 bits when converting between hexadecimal and binary. Once again, you start the process with the lower bits and work up. So `1010011` is grouped as `0101 0011` and becomes `53`. We saw that 2006 in decimal is `011111010110`. This groups as `0111 1101 0110` and becomes `7D6` in hex. Again, there is a method called `toHexString` that can be used on the numeric types to quickly get the hexadecimal representation of a number.

While `toHexString` give us hexadecimal representations of numeric values that we have in decimal, it is sometimes helpful to be able to enter values into programs using hexadecimal in a program. This can be done by prefixing a numeric literal with `0x`. The following uses of this confirms the conversion we did for the numbers above.

```
scala> 0x53.toBinaryString
res37: String = 1010011

scala> 0x7D6.toBinaryString
res38: String = 11111010110
```

2.5.5 Non-Integer Numbers

We saw previously that if we type in a numeric value that includes a decimal point Scala tells us that it has type `Double`. The `Double` literal format is more powerful than just including decimal points. It also allows you to use scientific notation to enter very large or very small numbers. Simply follow a number by an `e` and the power of ten it should be multiplied by. So `15000.0` can also be written as `1.5e4`.

The name `Double` is short for double precision floating point number. The full name includes information about the way that these numbers are stored in the memory of a computer. Like all values in a computer, the `Double` is stored as a collection of bits. To be specific, a `Double` uses 64-bits. This size is related to the double precision part. There is another type called `Float` that is a single precision floating point number and only uses 32-bits. In both cases, the internal representation uses floating point format. This is similar to scientific notation, but in binary instead of decimal. The bits in a floating point number are grouped into three different parts. We will call them s, e, and m and the value of the number is given by $(-1)^s * (1 + m) * 2^{(e-bias)}$. The first bit in the number is the sign bit, s. When that bit is on, the number is negative and when it is off it is positive. After the sign bit is a group of bits for the EXPONENT, e. Instead of using two's compliment for determining if the exponent is negative, the exponent is biased by a value that is picked to match with the number of bits in the exponent. Using a bias instead of two's compliment means that comparisons between floating point values can be done with the same logic used for integer values with the same number of bits. All remaining bits are used to store a MANTISSA, m. The stored mantissa is the fractional part of the number in normalized binary. So the highest value bit is $\frac{1}{2}$, the next is $\frac{1}{4}$, and so on. Table 2.2 below gives the number of bits used for e and m, the *bias*, and the range of numbers they can represent in the `Double` and `Float` types. The E notation is short for multiplication by 10 to that power.

As we have seen, floating point literals are considered to be of type `Double` by default. If you specifically need a `Float` you can append an `f` to the end of the literal. There are many other details associated with floating point values, but there is only one main point that will be stressed here. That is the fact that floating point values, whether `Double` or `Float`, are

Type	e Bits	m Bits	bias	Min	Max
Float	8	23	127	-3.4028235E38	3.4028235E38
Double	11	52	1023	-1.7976931348623157E308	1.7976931348623157E308

TABLE 2.2: Floating point types with sizes and ranges.

not Real numbers in the sense you are used to in math with arbitrary precision. Floating point numbers have limited precision. Like the integers, they can be overflowed. Unlike the integers, they are fundamentally imprecise because they represent fractional values with a finite number of bits. The real implications of this are seen in the following example.

```
scala> 1.0-0.9-0.1
res39: Double = -2.7755575615628914E-17
```

To understand why this happens, consider the simple fraction, $\frac{1}{3}$, the decimal representation of which 0.33333... In order to write this fraction accurately in decimal, you need an infinite number of digits. In math we can denote things like this by putting in three dots or putting a line over the digits that are repeated. For floating point values, the digits simply cut off when you get to the end of the mantissa. As such, they are not exact and the circuitry in the computer employs a rounding scheme to deal with this. This imprecision is not visible most of the time, but one immediate implication of it is that you cannot trust two floating point numbers to be equal if they were calculated using arithmetic. It also means that you should not use floating point numbers for programs that involve money. The decimal value `0.1` is a repeating fraction in binary, hence the problem in the example above, and as such, is not perfectly represented. Instead you should use an integer type and store cents instead of dollars.

2.6 The `math` Object

While on the topic of numbers, there are quite a few standard functions that you might want to do with numbers beyond addition, subtraction, multiplication, and division. There are a few other things you can get from operators that we will discuss later. Things like square root, logarithms, and trigonometric functions are not operators. They are found as methods in the `math` object. You can use tab completion in the REPL to see all the different methods that you can call on `math` and values stored in it.

```
scala> math.
BigDecimal                  ScalaNumericConversions max
BigInt                      abs                     min
E                           acos                    package
Equiv                       asin                    pow
Fractional                  atan                    random
IEEEremainder               atan2                   rint
Integral                    cbrt                    round
LowPriorityEquiv            ceil                    signum
LowPriorityOrderingImplicits cos                    sin
Numeric                     cosh                    sinh
Ordered                     exp                     sqrt
Ordering                    expm1                   tan
```

```
PartialOrdering             floor                   tanh
PartiallyOrdered            hypot                   toDegrees
Pi                          log                     toRadians
ScalaNumber                 log10                   ulp
ScalaNumericAnyConversions  log1p
```

Many of these probably do not make sense right now, and you should not worry about them. However, many of them should be identifiable by the name. So if we wanted to take a square root of a number, we could do the following.

```
scala> math.sqrt(9)
res40: Double = 3.0
```

You would use a similar syntax for taking cosines and sines. The functions provided in the math object should be sufficient for the needs of most people. Only two of the contents of math that start with capital letters are worth noting at this point. `Pi` and `E` are numeric constants for π and e.

```
scala> math.Pi
res41: Double = 3.141592653589793
```

Syntax versus Semantics

The terms SYNTAX and SEMANTICS are used very commonly when discussing programming languages. For natural languages, syntax can be defined as "the arrangement of words and phrases to create well-formed sentences in a language". This is a pretty good definition for programming languages other than we are not building sentences, we are building programs. The syntax of a programming language specifies the format or tokens and how tokens have to be put together to form proper expressions and statements as well as how statements must be combined to make proper programs.

As you will see, assuming that you have not yet, programming languages are much more picky about their syntax than natural languages. Indeed, the syntax of programming languages are specified in formal grammars. You do not really get the same type of artistic license in a programming language that you do in a natural language, as deviating from the syntax makes things incorrect and meaningless. Don't worry though, in expressive and flexible languages like Scala, you still have a remarkable amount of freedom in how you express things, and with experience, you can create beautiful solutions that follow the syntax of the language.

The semantics of a program deals with the meaning. Syntax does not have to be attached to meaning. It is just formal rules that are part of a formal system that specify if a program is well formed. It is the semantics of a language that tell us what something that follows the syntax actually means.

2.7 Naming Values and Variables

We have seen enough that you can solve some simple problems. For example, if you were given a number of different grades and asked to find the average, you could type in an expression to add them all up and divide by the number of them to get the average. We basically have the ability to use Scala now to solve anything we could solve with a calculator as well as doing some fairly simple string manipulation. We will develop a lot more over time, but we have to start somewhere. As it stands we are not just limited to solving problems we could do with a calculator, we are solving them the way we would with a calculator. We type in mathematical expressions the way we would write them on paper and get back an answer. Real programming involves tying together multiple lines of instructions to solve larger problems. In order to do this, we need to have a way to give names to values so we can use those values later.

There are two keywords in Scala that give names to values: `val` and `var`. To begin with, let us look at the full syntax of `val` and `var` in two samples. Then we can pull them apart, talk about what they do, see how they are different, and discuss what parts of them are optional.

```
scala> val age:Int = 2015-1996
age: Int = 19

scala> var average:Int = (2+3+4+5)/4
average: Int = 3
```

Syntactically the only difference between these two is that one says `val` and the other says `var`. That is followed by a name with a colon and a type after it. The rules for names in Scala are that they need to start with a letter or an underscore followed by zero or more letters, numbers, and underscores.[10] So `abc`, `abc123_def`, and `_Team2150` are all valid Scala names while `2points` is not. You also cannot use keywords as variable names. The only keywords that have been introduced so far are `val` and `var`, but there will be others, and you cannot use those as names for things as they are reserved by the language.

Scala is also case sensitive. So the names `AGE`, `age`, `Age`, and `agE` are all different. In general, it is considered very poor style to use names that differ only in capitalization as it can quickly lead to confusion. Most names will not involve underscores either, and numbers only appear where they make sense. Scala borrows a standard naming convention from Java called camel case. The names of values begin with a lower case letter and the first letter of subsequent words are capitalized. For example, `theClassAverage` is a name that follows this convention. Type names use the same convention except that the first letter is capitalized. This is called camel case because the capital letters look like humps.

The types in both of these examples are followed by an equal sign and an expression. Unlike many other programming languages, this is not optional in Scala. In Scala, when you declare a `val` or `var`, you must give it an initial value.[11]

While the initial value is not optional, the type generally is. Scala is able to figure out

[10] Scala also allows names that are either made entirely of operator symbols or have a standard name followed by an underscore and then operator symbols. Symbolic names should only be used in special situations, and using them improperly makes code difficult to read. For this reason, we will ignore these types of names for now.

[11] There are very good reasons for requiring initialization of variables. Even in languages that do not require it, a programmer can make his/her life a lot easier by initializing all variables at creation. The declaration and initialization should ideally happen at the point where you have a real value to put into the variable. This prevents many errors and as a result, can save you a lot of time in your programming.

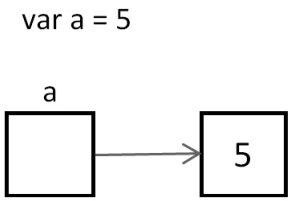

FIGURE 2.3: This figure shows how you should think about value and variable declarations in Scala. The variable itself stores a reference to an object. The difference between `val` and `var` is whether or not you can change what is referred to, not whether that object can be modified.

the types of things for us in many situations. If we leave off the colon and the type, Scala will simply use whatever type it infers is appropriate for the expression in the initial value. Most of the time, the type that it gives us will be exactly what we want. Using this we could instead have the following shorter forms of these declarations.

```
scala> val age = 2015-1996
age: Int = 19

scala> var average = (2+3+4+5)/4
average: Int = 3
```

The reason for using a `val` or `var` declaration is that they give a name to the value that we can refer back to later. For example, we could now type in `age+10` and Scala would give us `29`. The names serve two purposes. They prevent us from typing in expressions over and over again. They also help give meaning to what we are doing. You should try to pick names that help you or other readers figure out what is going on with a piece of code.

So far we have discussed all the similarities between `val` and `var` and you might be wondering in what way they are different. The declarations themselves are basically identical. The difference is not in the syntax, but in the meaning, or semantics. A `val` declaration gives a name to a reference to a value. That reference cannot be changed. It will refer to the thing it was originally set to forever. In the REPL, you can declare another `val` with the same name, but it does not do anything to change the original. A `var` declaration, on the other hand, allows the reference to change. In both cases we are not naming the value, we are naming a box that stores a reference the value. The significance of this will be seen in section 7.7. Figure 2.3 shows a visual representation of how you should picture what a `val` or `var` declaration does in Scala.

The act of changing the reference stored in one of these boxes we call variables is referred to as an ASSIGNMENT. Assignment in Scala is done with the same equal sign that was used to set the initial value. In an assignment though there is no `val` or `var` keyword. If you accidentally include either `var` or `val` you will be making a new variable, not changing the old one.

```
scala> average = 8
average: Int = 8
```

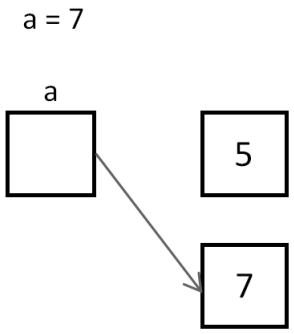

FIGURE 2.4: This figure shows what you might imagine happening with each of the lines assigning new values to the variable `average`.

```
scala> average = 2*average
average: Int = 16
```

The first assignment causes the box named average to change from referring to the object 3 to the object 8. The second one uses the previously referred to value and multiplies it by two, then stores a reference to that new value back into the variable. The effects of these lines are illustrated in figure 2.4.

As a general rule, you should prefer `val` declarations over `var` declarations. Try to make everything a `val`, and only convert it to a `var` if you find that you truly need to do so. The reason for this is that it simplifies the logic of your program and makes it less likely that you will mess things up. Things that change are harder to reason about than things that stay the same.

2.7.1 Patterns in Declarations

There is a bit more to the initialization of `val` and `var` declarations than was mentioned above. Technically, the initialization is able to do something called PATTERN MATCHING that we will get to in detail in chapter 5. For now, the only aspect we will care about is that we can put tuples on the left hand side of the equals sign where we would normally put just a variable name. First, let us see what happens if we do a `val` declaration with a tuple on the right hand side.

```
scala> val t = (100,5.7)
t: (Int, Double) = (100,5.7)
```

Note that `t` refers to the tuple and has a type `(Int,Double)`. This is exactly what we would expect. The power that pattern matching provides is that if you put multiple names inside of parentheses on the left of the equals, much like a tuple, all the names will be bound. That type of behavior is shown here.

```
scala> val (price,weight) = t
price: Int = 100
weight: Double = 5.7
```

The same can be done with a `var` and then all the names will have the ability to change what they refer to. This is the second way of getting values out of tuples. It is more readable because we can pick meaningful names for the variables. After doing the example above, you could use `price` and `weight` instead of `t._1` and `t._2`.

2.7.2 Using Variables

Let us use the ability to name values to do a little problem solving. We are given a total time in seconds, and we want to know what that is in hours, minutes, and seconds. We then want to print that out in a reasonable format of "hh:mm:ss". The first step in solving this problem is to figure out how to go from just seconds to hours, minutes, and seconds. Once we have that, we can worry about formatting it to get the right string value.

How do we get from seconds to hours, minutes, and seconds? First, how do you get from seconds to minutes? That is fairly easy, you simply divide by 60. Thanks to the fact that integer division truncates, you will get the proper number of whole minutes. Here are two lines that define a number of total seconds as well as a number of total minutes.

```
scala> val totalSeconds = 123456
totalSeconds: Int = 123456

scala> val totalMinutes = totalSeconds/60
totalMinutes: Int = 2057
```

That number of minutes is not exactly the amount of time we want though. There are seconds left over. How do we figure out how many seconds we should display? We could do `totalSeconds-(60*totalMinutes)`, but a simpler expression is used here.

```
scala> val displaySeconds = totalSeconds%60
displaySeconds: Int = 36
```

The modulo gives us the remainder after we have gotten all the full groups of 60. That is exactly what we want. Now how do we get the number of hours and the number of minutes to display? The math is the same because there are 60 minutes in each hour.

```
scala> val displayMinutes = totalMinutes%60
displayMinutes: Int = 17

scala> val displayHours = totalMinutes/60
displayHours: Int = 34
```

What we see from this is that 123456 seconds is 34 hours, 17 minutes, and 36 seconds. We could repeat this same process for a different number of seconds if we used a different value for `totalSeconds`.

Now that we have these values, we want to figure out how to get them into a string with the format "hh:mm:ss". A first attempt at that might look like the following.

```
scala> val finalString = displayHours+":"+displayMinutes+":"+displaySeconds
finalString: String = 34:17:36
```

For this particular number of seconds, this works just fine. However, if you play around with this at all, you will find that it has a significant shortcoming. If the number of minutes or seconds is less than 10, only one digit is displayed when we want two. So we need to come up with a way to get a leading zero on numbers that only have one digit. To do this, we will break the problem into two steps.

The first step will be to get the number of minutes and seconds as `String`s.

```
scala> val min=displayMinutes.toString
min: String = 17

scala> val sec=displaySeconds.toString
sec: String = 36
```

This might seem odd, but the string version has something that the number itself does not, an easy way to tell how many digits/characters are in it. When there is only one digit, we want to add an extra zero. When there is not, we leave it as is. We can get this effect by using the `*` method on the `String` and a little math. The short names were selected to keep our expression shorter for formatting, but that is not required.

```
scala> val finalString=displayHours+":"+("0"*(2-min.length))+min+":"+(
| "0"*(2-sec.length))+sec
finalString: String = 34:17:36
```

The result for these values is the same, but we could force some different value into `min` and `sec` to see that this does what we want.

```
scala> val min="5"
min: String = 5

scala> val sec="2"
sec: String = 2

scala> val finalString=displayHours+":"+("0"*(2-min.length))+min+":"+(
| "0"*(2-sec.length))+sec
finalString: String = 34:05:02
```

2.8 Details of `Char` and `String`

There is a lot more to `Char` and `String` than we covered in section 2.4. Some of it you really should know before we go further. We saw how we can make character literals or string literals that contain keys that appear on the keyboard and that go nicely into a text file. What about things that we cannot type as nicely or that have other meanings? For example, how do you put double quotes in a `String`? Typing the double quote closes off the string literal instead of putting one in. You are not allowed to have a normal string break across lines, so how do you get a newline in a String?

2.8.1 Escape Characters

We can do all of these things and more with ESCAPE CHARACTERS. These are denoted by a backslash in front of one or more characters. For example, if you want to put a double quote in a string, simply put a backslash in front of the double quote. You can insert a newline with a `\n`. If you want to insert a backslash simply put in two backslashes. Table 2.3 shows some commonly used escape characters.

In addition to escape characters, the backslash can be used to put any type of special

Literal	Meaning	Unicode Hex Encoding
\b	backspace	\u0008
\f	form feed	\u000C
\n	line feed	\u000A
\r	carriage return	\u000D
\t	tab	\u0009
\"	double quote	\u0022
\'	single quote	\u0027
\\	backslash	\u005C

TABLE 2.3: Table of special character escape sequences in Scala.

character into a string. If you know the UNICODE value for a special character, you can put \u followed by four hexadecimal digits in a string to specify that character.[12]

2.8.2 Raw Strings

There are some times when using the escape characters becomes a pain. For example, there are times when you need to build strings that have a number of backslashes. Each one you want requires you to put in two. This can get unwieldy. In addition, if you have a long, multi-line string, it can be difficult to format the string the way you want. For these types of situations, Scala includes a special form of string that begins and ends with three double quotes. Anything you put between the set of three double quotes is taken to be part of the string without alteration. These types of strings are called RAW STRINGS. The following shows an example of using this to enter a long string in the REPL.

```
scala> """This is a long string.
| It spans multiple lines.
| If I put in \n and \\ or \" they are taken literally."""
res42: String =
This is a long string.
It spans multiple lines.
If I put in \n and \\ or \" they are taken literally.
```

2.8.3 String Interpolation

In section 2.7.2, there were a number of expressions that put together strings using plus signs for concatenation. This approach can be challenging to write and read in code.[13] For that reason, there is an alternate approach to building strings that include values called STRING INTERPOLATION. The syntax for doing string interpolation is to put a "s" or a "f" in front of the string,[14] then put expressions in the string that begin with a dollar sign if they are to be evaluated and their values inserted.

The earlier example originally put together the string for the time using the expression

```
displayHours+":"+displayMinutes+":"+displaySeconds
```

[12]The topic of Unicode characters is beyond the scope of this book, but a simple web search will lead you to descriptions and tables of the different options.

[13]Using + to build long strings is also inefficient.

[14]The string interpolation mechanism in Scala is extensible, and programmers can add other options. The "s" and "f" forms are the main ones supported by the standard libraries.

Using string interpolation, this could be written as

```
s"$displayHours:$displayMinutes:$displaySeconds"
```

More complex expressions can be inserted into the string by enclosing the expression in curly braces after the dollar sign.

```
scala> val age = 2015-1996
age: Int = 19
scala> s"$age+10 = ${age+10}"
res43: String = 19+10 = 29
```

Here `$age` is nested inside an `s` processed string. The `s` interpolator knows to insert the value of the variable `age` at this location(s) in the string. There is no set rule for when you should use string interpolation instead of concatenation. You should pick whichever option you find easiest to read and understand.

The "f" interpolation requires that you place a format specifier after the expression. Coverage of these format specifiers is beyond the scope of this book. The interested reader is encouraged to look up details on his/her own. It should be noted that the format strings used by Scala are heavily based on those used in the printf function for the C programming language, and they appear in many libraries across different languages.

2.8.4 String Methods

There are many methods that you can call on the String type. Tab completion shows you some of them.

```
scala> "hi".
+                    concat            isInstanceOf       startsWith
asInstanceOf         contains          lastIndexOf        subSequence
charAt               contentEquals     length             substring
chars                endsWith          matches            toCharArray
codePointAt          equalsIgnoreCase  offsetByCodePoints toLowerCase
codePointBefore      getBytes          regionMatches      toString
codePointCount       getChars          replace            toUpperCase
codePoints           indexOf           replaceAll         trim
compareTo            intern            replaceFirst
compareToIgnoreCase  isEmpty           split
```

These are the methods that come from the Java String type, and they provide a lot of the basic functionality that one needs when working with strings. Through a language feature in Scala called implicit conversions, there are others that are also available. The listing below shows those. You can see that it includes multiplication, as introduced earlier. It also includes methods like `toInt` and `toDouble`, which will convert strings with the proper values to those types.

```
*             foldLeft           mkString           stripLineEnd
++            foldRight          nonEmpty           stripMargin
++:           forall             padTo              stripPrefix
+:            foreach            par                stripSuffix
/:            format             partition          sum
:+            formatLocal        patch              tail
:\            groupBy            permutations       tails
>             grouped            prefixLength       take
>=            hasDefiniteSize    product            takeRight
```

```
addString      head               r                   takeWhile
aggregate      headOption         reduce              to
apply          indexOf            reduceLeft          toArray
asInstanceOf   indexOfSlice       reduceLeftOption    toBoolean
canEqual       indexWhere         reduceOption        toBuffer
capitalize     indices            reduceRight         toByte
collect        init               reduceRightOption   toDouble
collectFirst   inits              replaceAllLiterally toFloat
combinations   intersect          repr                toIndexedSeq
compare        isDefinedAt        reverse             toInt
compareTo      isEmpty            reverseIterator     toIterable
contains       isInstanceOf       reverseMap          toIterator
containsSlice  isTraversableAgain sameElements        toList
copyToArray    iterator           scan                toLong
copyToBuffer   last               scanLeft            toMap
corresponds    lastIndexOf        scanRight           toSeq
count          lastIndexOfSlice   segmentLength       toSet
diff           lastIndexWhere     seq                 toShort
distinct       lastOption         size                toStream
drop           length             slice               toString
dropRight      lengthCompare      sliding             toTraversable
dropWhile      lines              sortBy              toVector
endsWith       linesIterator      sortWith            union
exists         linesWithSeparators sorted             updated
filter         map                span                view
filterNot      max                split               withFilter
find           maxBy              splitAt             zip
flatMap        min                startsWith          zipAll
fold           minBy              stringPrefix        zipWithIndex
```

Going through all these methods is well beyond the scope of this chapter, but it is beneficial to see examples that use some of them. To do this, let us consider a situation where we have a person's name written as "*first last*". We wish to build a new string that has the same name in the format of "*last, first*". In order to do this, we must first find where the space is, then get the parts of the original string before and after the space. Once we have done that, we can simply put the pieces back together in the reverse order with a comma between them.

To find the location of the space, we will use the `indexOf` method. This method gives us a numeric index for the first occurrence of a particular character or substring in a string.

```
scala> val name = "Mark Lewis"
name: String = Mark Lewis

scala> val spaceIndex = name.indexOf(" ")
spaceIndex: Int = 4
```

The index of the space is 4, not 5, because the indexes in strings, and everything else except tuples, start counting at 0. So the `'M'` is at index 0, the `'a'` is at index 1, etc.

Now that we know where the space is, we need to get the parts of the string before and after it. That can be accomplished using the substring method.

```
scala> val first = name.substring(0,spaceIndex)
first: String = Mark

scala> val last = name.substring(spaceIndex+1)
```

```
last: String = Lewis
```

The first usage passes two arguments to `substring`. The first is the index of the first character to grab, in this case, it is 0. The second is the index after the last character to grab. In this case, it is the index of the space. The fact that the second bound is exclusive is significant, and it is a standard approach for methods of this nature in many languages. The second form takes a single argument, and it returns the substring from that index to the end of the string, making it ideal for getting the last name.

The two strings can now be put back together using concatenation or string interpolation. The following shows how to do it with string interpolation.

```
scala> val name2 = s"$last, $first"
name2: String = Lewis, Mark
```

One could also pull out the names using the `splitAt` method.

```
scala> val (first,last) = name.splitAt(spaceIndex)
first: String = Mark
last: String = " Lewis"

scala> val name2 = s"${last.trim}, $first"
name2: String = Lewis, Mark
```

The `splitAt` method returns a tuple, and we use a pattern here to pull out the two elements. Note that the space itself was included in the second element of the tuple. To get rid of that, we use the `trim` method. This method gives us back a new string with all leading and trailing whitespace removed.

If you only want a single character from a string, you can get it by INDEXING into the string with parentheses. Simply specify the index of the character you want in parentheses after the name of the string. So we could get the last initial from our original name string like this.

```
scala> name(spaceIndex+1)
res44: Char = L
```

2.8.5 Immutability of Strings

When looking through the list of methods on the String type, you might have noticed methods called `toLowerCase` and `toUpperCase`. These methods illustrate a significant feature of strings, the fact that they are IMMUTABLE. This means that once a string object has been created, it can not be changed. The `toLowerCase` method might sound like it changes the string, but it does not. Instead, it makes a new string where all the letters are lower case, and gives that back to us. This is illustrated by the following.

```
scala> val lowerName = name.toLowerCase
lowerName: String = mark lewis

scala> name
res45: String = Mark Lewis
```

The `lowerName` variable refers to a string that is all lower case, but when we check on the value of the original `name` variable, it has not been changed. None of the methods of String change the value. Any that look like they might simply give back modified values. This is

what makes the String type immutable. The `trim` method used above demonstrates this same behavior. Most of the types we will deal with are immutable.[15]

2.9 Sequential Execution

Sequential execution is used when we write a program and want the instructions to execute in the same order that they appear in the program, without repeating or skipping any instructions from the sequence. So far, all of the program instructions we have written have been executed one after another in the same order that we have typed them in. The instructions have been executed sequentially.

Working in the REPL is great for certain tasks, but what if you have a sequence of things you want to do, and you want to do it multiple times. Having to type in the same set of instructions repeatedly is not a very good option. The time conversion above is a perfect example of that. If we want to do this for a different number of seconds, we have to repeat all the commands we just performed. Indeed, you cannot really say that you have programmed until you put in a fixed set of instructions for solving a problem that you can easily run multiple times. That is what a program really is. So now it is time to write our first program of any significance.

We have used the REPL to enter commands one at a time. This is a great way to test things out in Scala and see what a few commands do. A second way of giving commands to Scala is to write little programs as SCRIPTs. The term script is used to describe small programs that perform specific tasks. There are languages, called scripting languages, that have been created specifically to make the task of writing such small programs easier. Scala is not technically a scripting language, but it can be used in that way. The syntax was created to mirror a lot of the things that are commonly put into scripting languages, and if you run the `scala` command and give it the name of a file that contains Scala code, that file will be run as a script. The statements in it are executed in order.[16] The script for our time conversion looks like this.

Listing 2.1: TimeConvert.scala

```
val totalSeconds = 123456
val displaySeconds = totalSeconds%60
val totalMinutes = totalSeconds/60
val displayMinutes = totalMinutes%60
val displayHours = totalMinutes/60
val sec = displaySeconds.toString
val min = displayMinutes.toString
val finalString = displayHours+":"+("0"*(2-min.length))+min+
    ":"+("0"*(2-sec.length))+sec
println(finalString)
```

If you put this into a file called `TimeScript.scala` and then run `scala TimeScript.scala`, you will get the output `34:17:36`. The `println` statement is required for the script because unlike the REPL, the script does not print out values of all statements. You can run through

[15]The first mutable type we will encounter will be the Array type in chapter 6. That chapter will go further in demonstrating the significance of this distinction.

[16]We will see later that the statements are not always executed in order because there are statements that alter the flow of control through the program. Since we have not gotten to those yet though, execution is completely sequential at this point.

this code in the REPL using the `:load` command. If you do "`:load TimeScript.scala`" you will see it print out all of the intermediate values as well as the result of the `println`.

This script allows us to run the commands repeatedly without retyping. By editing the value of `totalSeconds`, we can test other total times fairly quickly. However, a better solution would be to allow a user to tell us how many seconds to use every time we run the script. We can easily get this behavior by replacing the top line of the script we had with these three lines.

Listing 2.2: TimeConvert2.scala

```
import io.StdIn._
print("Enter the number of seconds. ")
val totalSeconds = readInt()
```

The second line prints a prompt to let the user know that we are waiting for something to be input. After that we have altered the initialization of `totalSeconds` so that instead of giving it a fixed number, it is initialized to the value returned by `readInt`. This calls a function that reads in a single integer from the user. The first line is there because the full name of `readInt` is `io.StdIn.readInt`. The `import` statement allows us to use a shortened name whenever we want to read a value. The underscore in the import causes it to bring in other functions such as `readLine` and `readDouble` which allow us to read in strings and double values respectively. If you make this change and run the script, you will be able to enter any number of seconds, assuming it is a valid `Int`, and see it converted to hours, minutes, and seconds.

The following code shows the usage of `readLine` and `readDouble`.

```
import io.StdIn._
val name = readLine()
val number = readDouble()
```

Note that all of these functions read a full line from the user and expect it to match the desired type. If you want the user to enter multiple numbers on one line, you cannot use `readInt` or `readDouble`. For that you would have to read a `String` with `readLine`, then break it apart and get the numeric values.

2.9.1 Comments

When writing programs in files, not in the REPL, it is often useful to include plain English descriptions of parts of the code. This is done by writing comments. If you are writing code for a course, you likely need to have your name in the code. Your name is likely not valid Scala, so it should go in a comment. Different instructors and companies will have different commenting standards that you should follow. In a professional setting, comments are used primarily for two reasons. The first is to indicate what is going on in the code, particularly in parts of the code that might be difficult for readers to understand. The second is for documentation purposes using tools that generate documentation from code.

There are two basic comment types in Scala, single line comments and multiline comments. Single line comments are made by putting `//` in the code. Everything after that in the line will be a comment and will be ignored when the program is compiled and run. Multiline comments begin with `/*` and end with `*/`. You can put anything you want between those, and they can be spaced out across many lines. Code shown in this book will have limited commenting as descriptions of the code appear in the text of the book, and there is little point in duplicating that content.

2.10 A Tip for Learning to Program

In many ways, learning to program, whether in Scala or any other programming language, is very much like learning a new natural language. The best way to learn a natural language is through immersion. You need to practice it and be surrounded by it. The key is to not simply memorize the rules and vocabulary, but to put them into use and learn them through regular usage. You should strongly consider approaching programming in the same way.

So what does it mean to immerse yourself in a programming language? Clearly you are not going to have conversations in it or enjoy television or radio broadcasts in it. The way to immerse yourself in a programming language is to take a few minutes every day to write in it. You should consider trying to spend 15-30 minutes each day writing code. The REPL in Scala is an excellent tool for you to enter in statements to see what they do. Try to play around with the language. Instead of approaching it as memorizing keywords and rules, try to put things into use. The things that you use frequently will stick and become natural. Those things that you do not use regularly you will have to look up, but that is normal. Programmers, even professional programmers with many years of experience in a language, still keep references handy.

Over time, the number of lines of code that you write in these short time intervals each day will grow as the basics become second nature and you begin to practice more advanced concepts. By the end of this book you might find yourself writing a hundred lines or so of functional code on certain days during that time span. By that time you will hopefully also pick up a "pet project", something that you are truly interested in programming and that you will think about the structure and logic of much more frequently.

Especially early on, you might find it hard to think of anything that you can write. To help you with this, many of the chapters in this book contain a "Self-Directed Study" section, like the one below. Use these as a jumping off point for the material in each chapter. After that will come a set of exercises and often a set of larger problems called projects. Remember that one of the significant goals of learning to program is improving your problem solving skills. While the Self-Directed Study section will help you to familiarize yourself with the details presented in a chapter, the exercises and projects are actual problems that you are supposed to solve in a formal way using Scala. You should use these to help provide you with the immersion you need to learn the language.

2.11 End of Chapter Material

2.11.1 Problem Solving Approach

Many students who are new to programming struggle with putting the English descriptions for solving a problem that they have in their head into whatever programming language they happen to be learning. The reality is that for any given line of code, there are a fairly small number of "productive" things that you could write. In the REPL you can test out any statement that you want, but in a script, an expression like `4+5` does not do much when used alone as a statement. Sections like this one will appear at the end of a number of chapters as we introduce new concepts that might stand alone as statements, or which alter statements we have talked about previously in a significant way. The goal of

these sections is to help focus your thinking so you can narrow down the list of possibilities any time that you are trying to decide what to put into the next line of code.

Given what we have just learned, there are only three types of statements that you would put in a script that stand alone:

1. A call to `print` or `println` to display information to the user. The function name should be followed with parentheses that contain the *expression* you want to print.
2. A variable declaration using `val` or `var`. A `val` declaration would look like `val` *name* `=` *expression*. The *name* could be followed with a colon and a type, though most of the time those will be left off.
3. An assignment into a previously declared `var` of the form *name* `=` *expression*. The *expression* must have a type that agrees with the type the variable was created with.

If you want to read information using a function like `readLine()`, `readInt()`, or `readDouble`, that should appear as part of an *expression* in one of the above statements. Remember to include the `import io.StdIn._` statement at the top of your file if you are going to be reading user input.

2.11.2 Summary of Concepts

- When you install Scala on your computer you get a number of different executable commands.
  - The `scala` command can run scripts or applications. If no argument is given it opens up the REPL for you to type in individual statements.
  - The `scalac` command is used to compile Scala source code to bytecode.
- Programming languages have relatively simple rules that they always follow with no ambiguity.
  - Tokens are the smallest piece with meaning. They are like words in English.
  - Expressions are combinations of tokens that have a value and a type.
  - Statements are complete instructions to the language. In Scala, any expression is a valid statement.
  - The simplest expressions are literals.
    - `Int` literals are just numbers with no decimal points like 42 or 365.
    - Adding an `L` to the end of an integer number makes a `Long` literal.
    - Numbers that include decimal points or scientific notation using `e` syntax are of the type `Double`.
    - Adding an `f` to the end of a number makes it a `Float`.
    - `Char` literals are single characters between single quotes.
    - `String` literals can have multiple characters between double quotes. Raw strings start and end with three double quotes and allow newlines.
- An object is a combination of information and functionality that operates on that information.
  - The information is called data members, fields, or properties.
  - The functionality is called methods.

  - All values in Scala are objects.
  - Methods are normally invoked using the "dot" notation. Arguments go in parentheses after the method name.
    * Scala allows the `.` to be left out if there is at least one argument to the method.
    * Parentheses are also optional for argument lists of length zero or one.
    * Operators are really method calls. So `4+5` is really `(4).+(5)`.

- Numbers in computers are not exactly like numbers in math, and you need to know some of the differences so you will understand when they lead to unexpected behavior.
  - All values stored in a computer are stored in binary, base 2, numbers. Each digit is called a bit. Different types use different numbers of bits for storage. The finite number of bits means that there are minimum and maximum values that can be stored in each type.
  - Negative integer values are stored using two's compliment numbers.
  - Binary numbers require a large number of digits, though they are all either 0 or 1, and converting to and from decimal is non-trivial. For this reason, computer applications frequently use base 8, octal, and base 16, hex. You can make a hexadecimal literal by putting a leading `0x` on an integer.
  - Non-integer numeric values are stored in floating point notation. This is like scientific notation in binary. These use the types `Float` and `Double`. Due to finite precision, not all decimal numbers can be represented perfectly in these numbers, and there are small rounding errors for arithmetic.

- Additional mathematical functions, like trigonometric functions and square root are methods of the `math` object.

- You can declare variables using the keywords `val` and `var`. The name of a variable should start with a letter or underscore and can be followed by letters, underscores, or numbers. A `var` declaration can be reassigned to reference a different value.

- String interpolation allows you to easily put values into strings without using `+` to concatenate them.

- There are many methods you can call on strings that allow you to do basic operations on them.

- Strings are immutable. That means that once a string is created, it is never changed. Methods that look like they change a string actually make a new string with the proper alterations.

- Instructions can be written together in scripts. The default behavior of a script is for lines of code to execute sequentially. Script files should have names that end with `.scala`. You run a script by passing the filename as a command-line argument to the `scala` command.

- Learning a programming language is much like learning a natural language. Do not try to memorize everything. Instead, immerse yourself in it and the things you use frequently will become second nature. Immersion in a programming language means taking a few minutes each day to write code.