SCALA

SUCCINCTLY

BY CHRIS ROSE

SUCCINCTLY E-BOOK SERIES

Syncfusion®

Scala Succinctly

By
Chris Rose

Foreword by Daniel Jebaraj

Copyright © 2017 by Syncfusion, Inc.
2501 Aerial Center Parkway
Suite 200
Morrisville, NC 27560
USA
All rights reserved.

Important licensing information. Please read.

This book is available for free download from www.syncfusion.com on completion of a registration form.

If you obtained this book from any other source, please register and download a free copy from www.syncfusion.com.

This book is licensed for reading only if obtained from www.syncfusion.com.

This book is licensed strictly for personal or educational use.

Redistribution in any form is prohibited.

The authors and copyright holders provide absolutely no warranty for any information provided.

The authors and copyright holders shall not be liable for any claim, damages, or any other liability arising from, out of, or in connection with the information in this book.

Please do not use this book if the listed terms are unacceptable.

Use shall constitute acceptance of the terms listed.

SYNCFUSION, SUCCINCTLY, DELIVER INNOVATION WITH EASE, ESSENTIAL, and .NET ESSENTIALS are the registered trademarks of Syncfusion, Inc.

Technical Reviewer: James McCaffery
Copy Editor: John Elderkin
Acquisitions Coordinator: Morgan Weston, social media marketing manager, Syncfusion, Inc.
Proofreader: Darren West, content producer, Syncfusion, Inc.

Table of Contents

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Christopher Rose is an Australian software engineer. His background is mainly in data mining and charting software for medical research. He has also developed desktop and mobile apps and a series of programming videos for an educational channel on YouTube. He is a musician and can often be found accompanying silent films at the Majestic Theatre in Pomona, Queensland.

Chapter 1 Introduction

Scala is a general-purpose language designed with both object-oriented and functional mechanisms. Scala can be used as a standalone application language, but it can also be used to develop modules for Java-based programs. The language was developed to express paradigms that are difficult to express in Java. The Scala compiler (a program called scalac.exe) uses the Java Virtual Machine (JVM) in order to compile source code to Java bytecode for execution.

If you are not already familiar with Scala, I recommend you first learn the basics of the Java language. The two languages share strong links—all of the Java libraries are available in Scala, and Scala integrates seamlessly into existing Java applications. Figure 1 depicts the close relationship between Java and Scala.

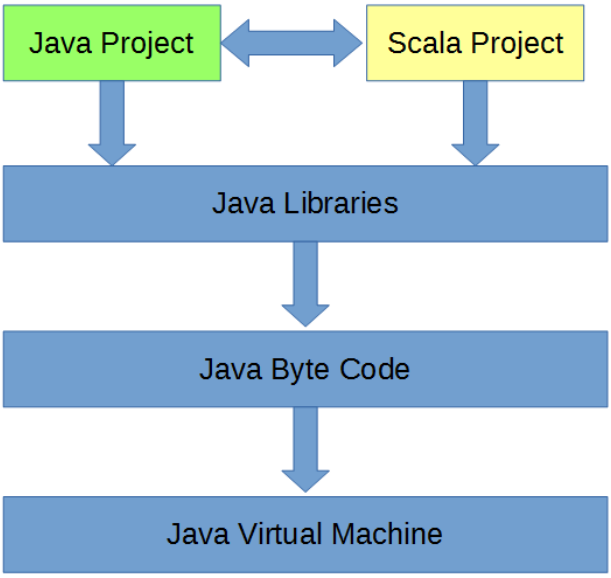

Figure 1: The Relationship between Scala and Java

Installation

Scala can be programmed using the console, but this is not a practical method for programming large-scale projects. The best method for programming useful modules is to install the Eclipse Scala Integrated Development Environment (IDE), which is a set of tools designed to assist development in one or more programming languages.

The Eclipse Scala IDE is available from http://scala-ide.org/. Visit and download the latest version.

Note: In this e-book, I will use the Scala Eclipse IDE exclusively, but there are other options available for developing the Scala application and integrating with existing Java applications. IntelliJ IDEA is another popular Java IDE that can be used to develop Scala modules and projects. You can also use the command line and develop Scala modules without an IDE.

When your download is complete, create or locate the folder in which you would like to store the application—for instance, in C:\Program Files\. Create a new folder called Scala Eclipse, copy the downloaded file to this new folder, and extract the archive's contents. You should end up with a folder filled with the files and folders depicted in Figure 2.

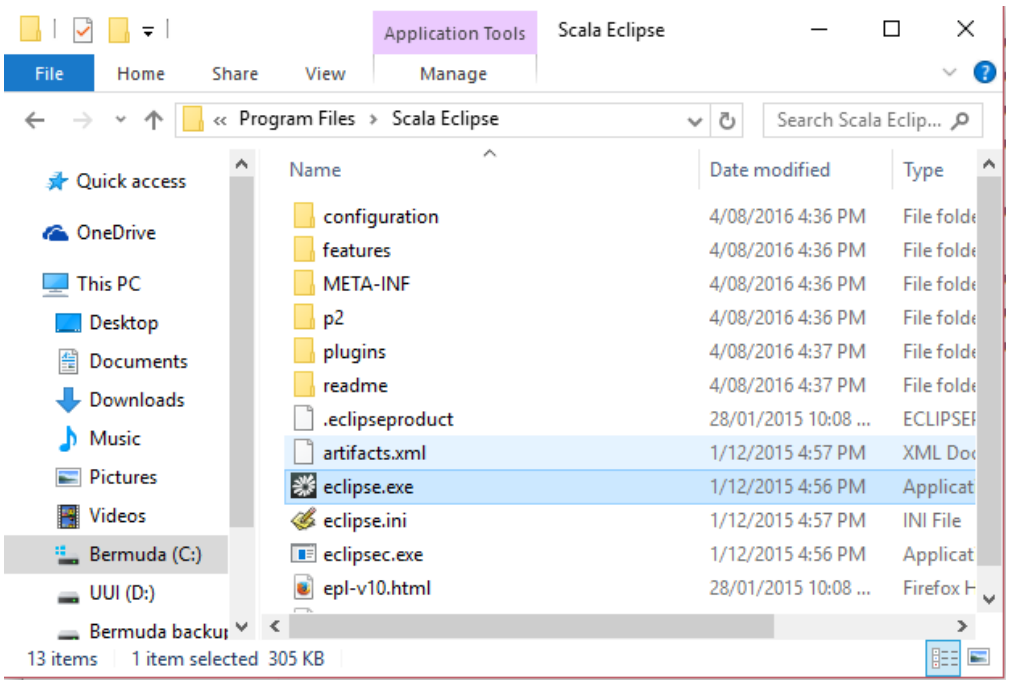

Figure 2: Installing Eclipse Scala IDE

Finally, in order to run the Scala Eclipse IDE conveniently, you might want to create a shortcut. Right-click the file called **eclipse.exe** and select **Create Shortcut**. Eclipse.exe is the main executable file for the IDE. Create a shortcut on your desktop or in some other convenient location. In order to run the IDE, double-click **clipse.exe** (or your newly created shortcut).

Selecting a workspace

When you run Eclipse, the Workspace selection dialog box opens (as per Figure 3). This allows you to specify which workspace you want to work with. A workspace is simply a folder that holds a collection of projects, and you can create a new workspace folder by clicking **Browse...** For instance, you can place your projects into your Documents folder in a subfolder called Scala Workspace. When you have selected a workspace or decided to use the default, click **OK**.

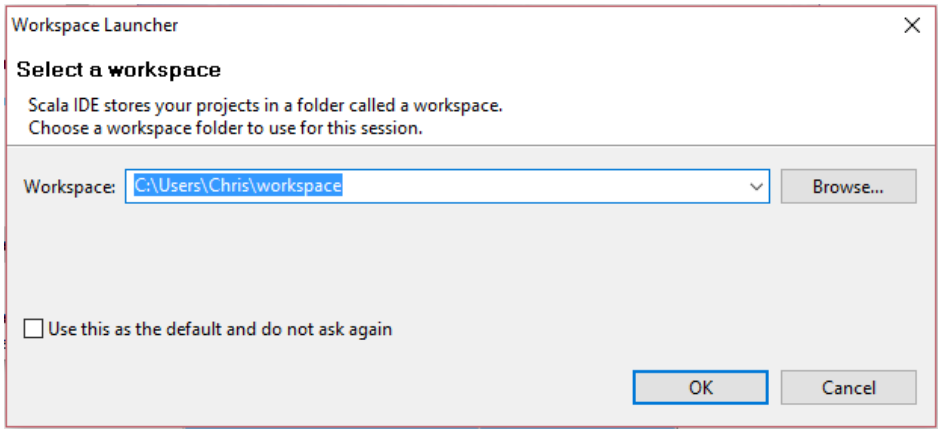

Figure 3: Selecting a Workspace

Hello World

For our first project, we will make a simple Hello World program in order to test that everything has been set up correctly. In order to begin a new project, start the Eclipse Scala IDE. Click **File → New** in the file menu of Scala Eclipse and select **Scala Project** from the submenu (as per Figure 4).

Note: Scala requires the Java Runtime Environment to be set up on the machine (JRE for short). This will probably be installed already. The latest version of the machine can be downloaded from Oracle at http://www.oracle.com/technetwork/java/javase/downloads/jre8-downloads-2133155.html. It is best to maintain an up-to-date JRE on your development machine(s) so that your Scala applications gain all of the benefits and optimizations of the latest JVM. Scala also requires the Java SE development kit, which can be downloaded from http://www.oracle.com/technetwork/pt/java/javase/downloads/jdk8-downloads-2133151.html.

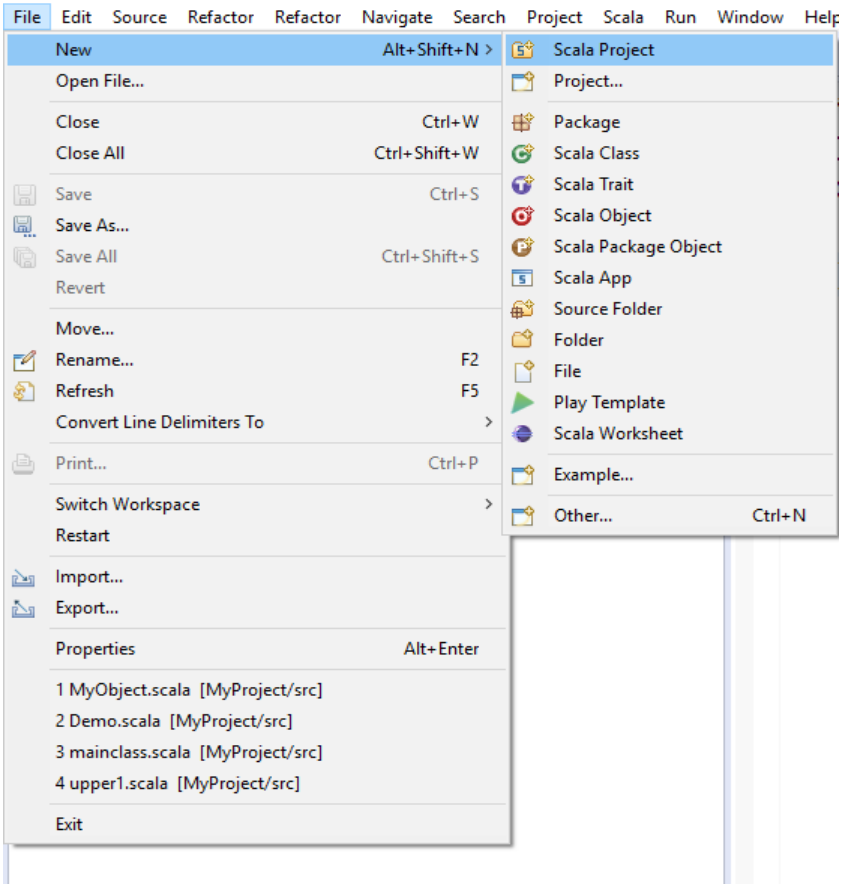

Figure 4: Beginning a New Project

Eclipse will show the New Scala Project window. Type a name for your project in the Project name box and click **Finish**. In Figure 5, I have called my new project HelloWorld.

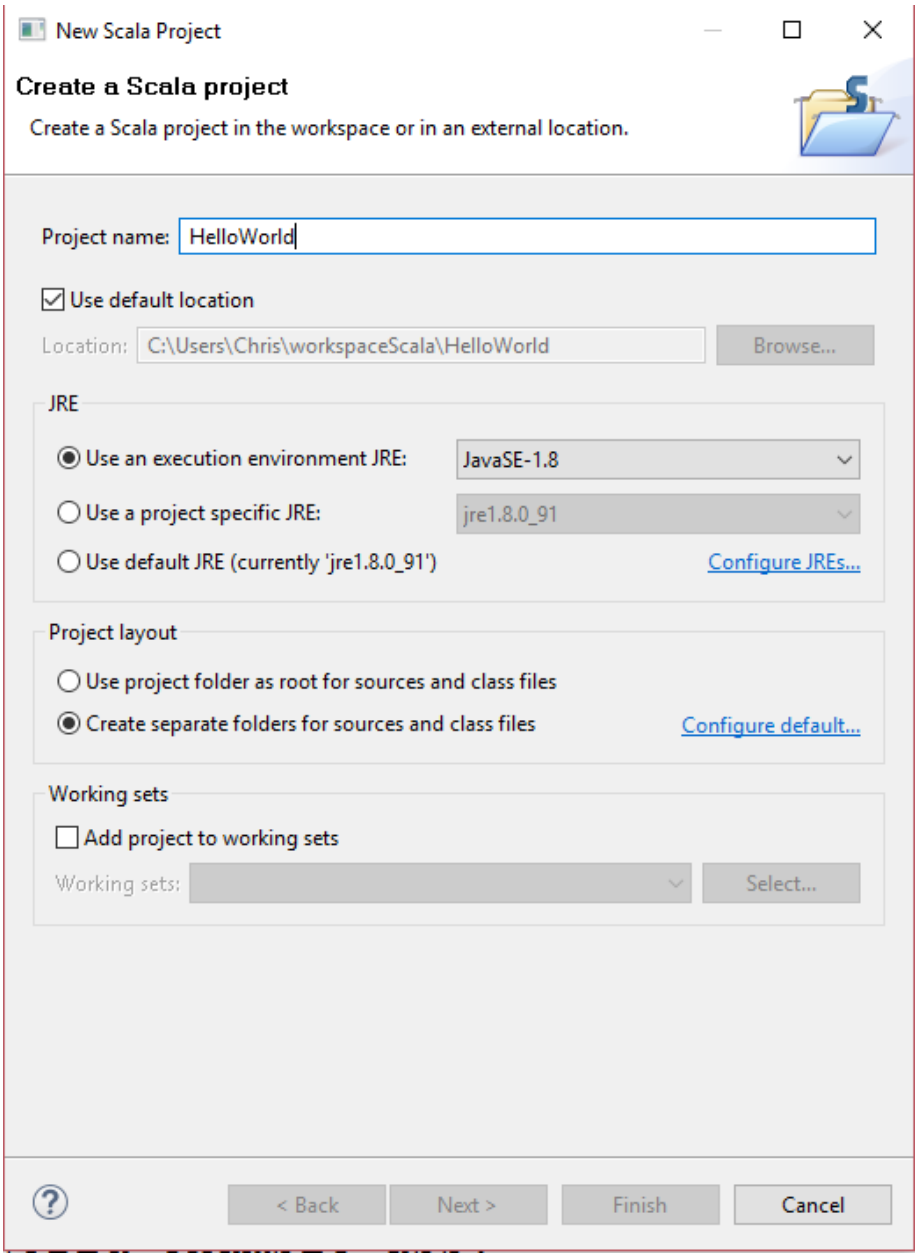

Figure 5: Creating a Project

A project is a collection of classes, objects, files, and resources that will be compiled by the scalac.exe and executed by the JVM. Project names must be unique within the workspace. They must also follow any conventions defined by the operating system. For instance, we should avoid using special symbols in our project names, such as & an $, and stick to letters and digits.

After clicking **Finish**, the Eclipse Scala IDE will create a simple project for you. Next, we need to add the main object, which will hold a program entry point. Right-click the **src** folder in the package explorer window. If that window is not visible on the file menu, click **Window>Show View>Other>Java>Package Explorer**. Next, select **New>Scala Object** (see Figure 6). Eclipse will open the New File Wizard window, as in Figure 7.

Note: The layout of the windows and panels and the debugging options are all the same between the Scale Eclipse IDE and the Java Eclipse IDE. Read Java Succinctly 1 for an introduction to the basic panels, windows, and debugging tools available in Eclipse.

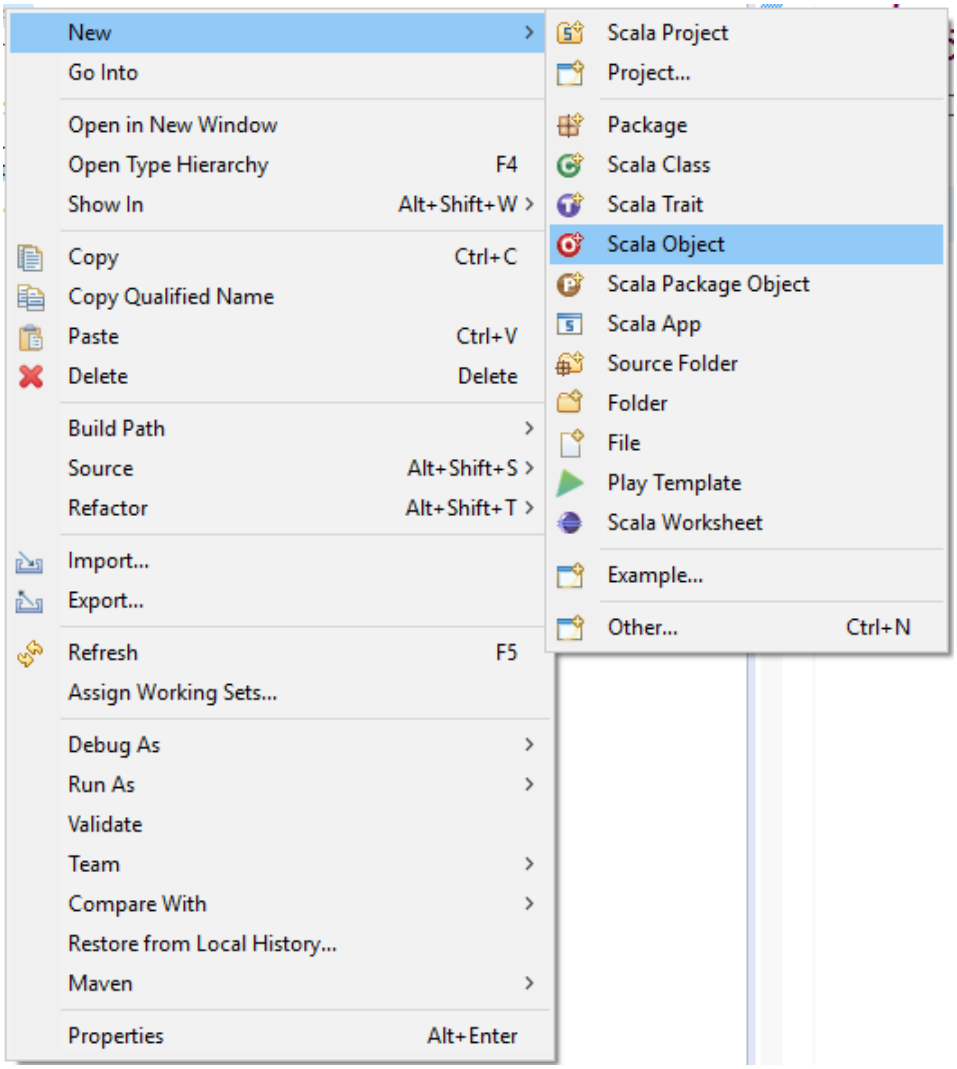

Figure 6: Creating a New Scala Object

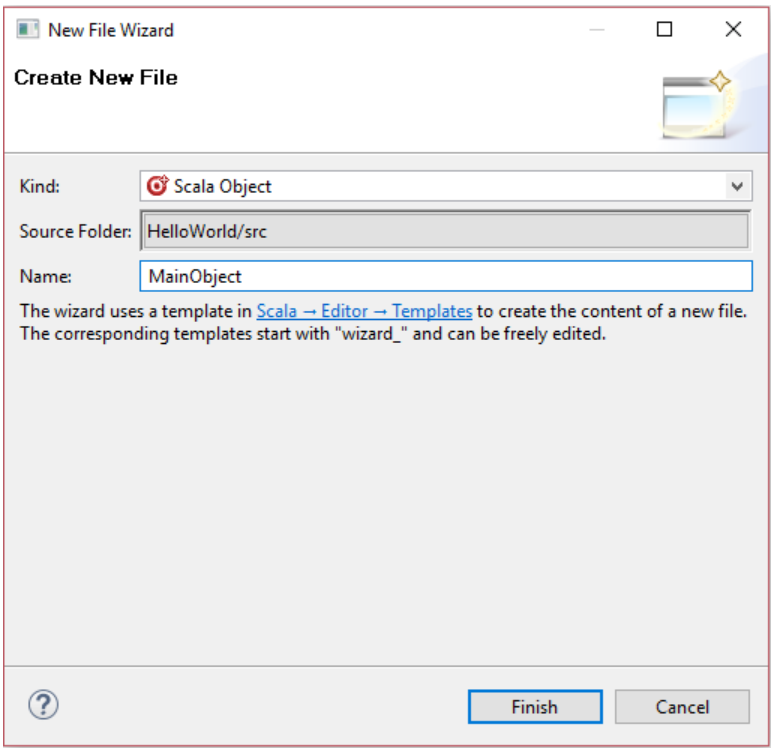

Figure 7: Opening the New File Wizard Window

In the New File Wizard window, give your object a name (I have called mine **`MainObject`**) and click **Finish**. Eclipse will create a new file for you with the code presented in Code Listing 1.

Code Listing 1: MainObject

```
object MainObject {

}
```

Scala is an object-oriented language with features similar to C++, C#, and Java. It operates a collection of objects built from classes. But Scala also allows us to define singleton objects. Singleton objects are the only instance built from a class. The object called **`MainObject`** in Code Listing 1 is actually a singleton. Instead of defining a blueprint with the keyword **`class`**, we define a singleton by using the keyword **`object`**.

In order for the JVM to know where to begin executing our new application, we must create an entry point—a main method. The JVM will call our main method once. It will never create more than one object of type **`MainObject`**, and this is why we can create the class as a singleton object rather than an inheritable or instantiable class. Code Listing 2 shows the completed code for the HelloWorld application.

Note: We do not have to specify a main method. We can create usable modules without a main method. The main method is only used when we intend to create an executable application.

Code Listing 2: HelloWorld Application

```
// Main object definition
object MainObject {
	// Main method definition
	def main(args: Array[String]): Unit = {

		// Print greeting:
		println("Hello world!")

	}	// Close main method
}	// Close MainObject
```

If you know Java programming, Code Listing 2 might look familiar. For one thing, Scala is a curly-brace language. This means it uses **{** and **}** to designate code blocks. Note how each **{** has a matching **}**. Also, be aware that there are several conventions used for tabbing that are intended to make reading code easier.

Note: Scala may be a curly-brace language, but it is quite different from other C-family languages. You might notice that there are no semicolons at the ends of the statements. You can put in the semicolons, but they are almost never necessary in Scala. Mostly, we use semicolons when we need to place multiple statements on a single line; in this case the semicolon is used to delimit statements.

In Code Listing 2, we defined a method called **main**. The method begins with the **def** keyword and is followed by the name of the method (**main** in this case). Here the name **main** is a reserved word, and note that Scala is case sensitive. The inputs to the method are specified in brackets: **(args: Array[String])**. This particular method expects an **Array** of type **String** to be passed as a parameter, while **args** is the variable name or identifier (we will look at in detail for defining variables and passing arguments). We will ignore the **args** array in this e-book, but it is actually the optional command-line arguments; we could read the elements of **args** and respond to any command-line arguments the user passes when running our application from the console, a batch file, or shortcut that supplied arguments.

After we define a function's parameter list, we place a colon followed by the output type of the function—in this case **: Unit**. Note that I use the words "function" and "method" more or less interchangeably. The technical differences between a Scala function and a Scala method are quite subtle but, with regards to the examples in this e-book, those differences are important. See the Scala documentation for the gory details. The **main** method does not return anything to the caller, so we place the keyword **Unit** as the output (which is equivalent to the void return type in other languages). We then use an equals operator (=) and open the code block for the specification of the method's body.

Tip: There are often many optional elements in Scala's syntax. For instance, if a function does not return anything, the ": Unit" is optional. So is the "=" sign. In Code Listing 1 we could have used the line "`def main(args: Array[String]) {`" to define our main method. Scala is very good at inferring information such as data types.

Inside the method body, we use the **`println`** built-in function to print a string of text to the screen. And we close the code blocks for the **`main`** method and the **`MainObject`**.

Running the application

When we have a Scala project with a main entry point, we can run the application by clicking **Scala Application** in the file menu, **under Run>Debug As**—as in Figure 8. You'll be presented with a Select Preferred Launcher dialog window. Check the **Use configuration specific settings** option, then select either the **JVM** or **New Debugger** option and click **OK**. You can change this selection later with **Run > Debug Configurations > Common > Select Other > (option)**.

The first time you use Scala Eclipse, you might get a firewall warning. If so, select the option that allows eclipse.exe to run on your network.

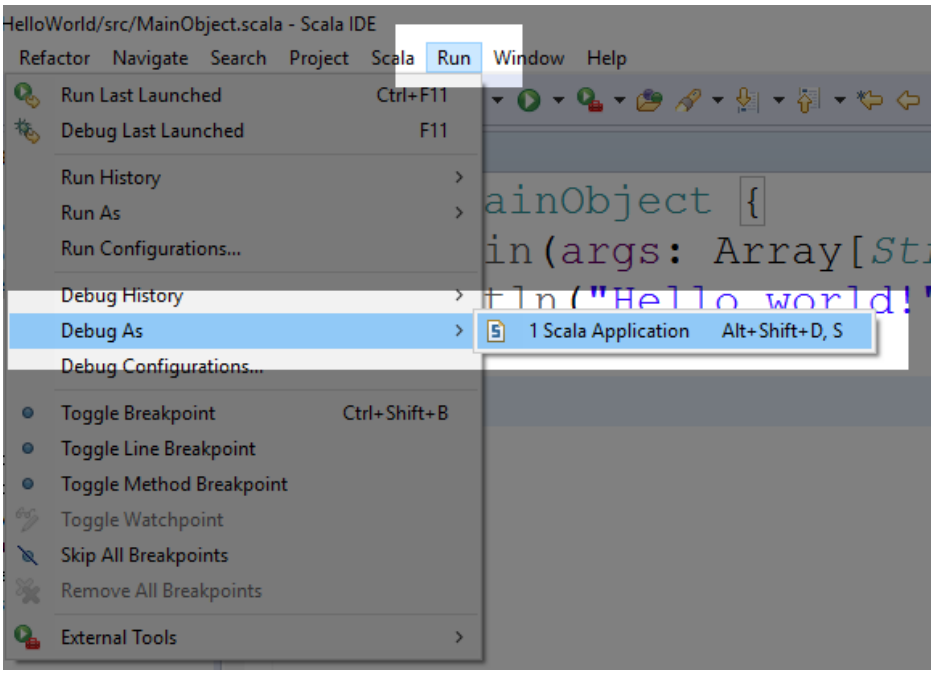

Figure 8: Debugging a Scala Application

When you run the application, it will print the line "Hello world!" to the Console. In Scala Eclipse, the Console is represented by a small window at the lower end of the screen—see Figure 9.

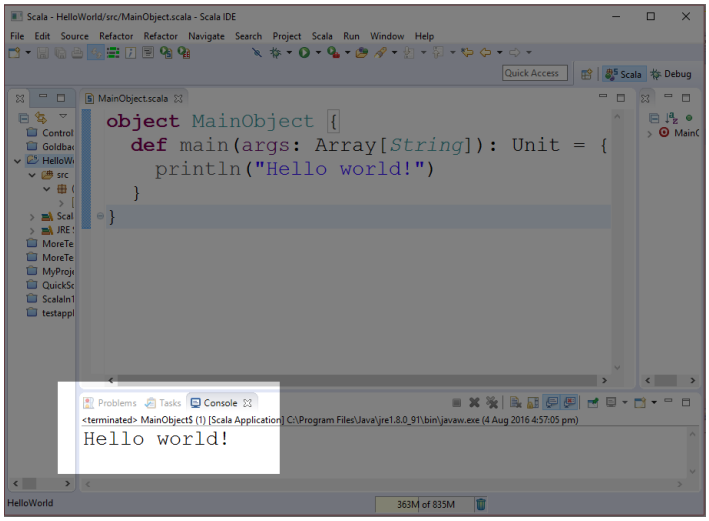

Figure 9: A Console

Note: We will be working entirely with the Console in this e-book, but Scala has the complete Java Library available to it. You can create applications in Java with a Graphical User Interface (GUI), or you can use the Java GUI library of classes to build a GUI in Scala. I will not go into the details of building a GUI here, but I refer you to* Java Succinctly 2*, in which we look at how to build a GUI using Java.

After you have run your Scala application once using the method described here, Scala Eclipse will create Run Configuration for you. When your project has a run configuration, you can run and debug the application by clicking **Debug** or **Run**, as shown in Figure 10.

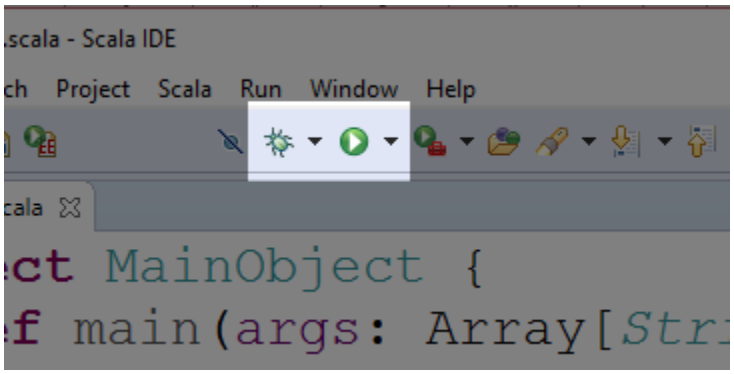

Figure 10: Debug and Run Buttons

Debug and run configurations

The debug and run configurations can be changed by clicking **Run → Debug Configurations** or **Run → Run Configurations**. Clicking these options will open up the Debug Configurations box, as per Figure 11. If you need to supply command-line arguments to your programs (which will be passed as the **`args Array`** parameter to the **`main`** method) or test the project using a different JRE, you can do so with this box. We will use the default configuration throughout this e-book.

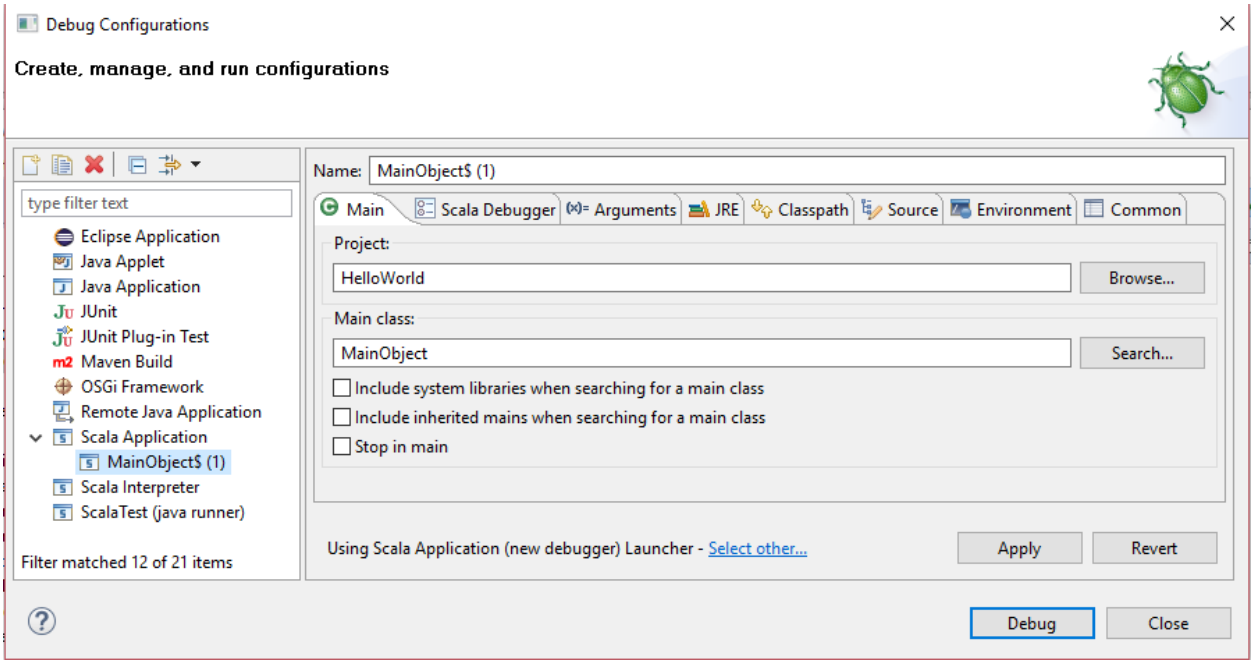

Figure 11: Run and Debug Configurations Box

Problems and errors

Finally, if your project does not run and print “Hello world!” to the console, Eclipse may show an error box. If so, it will ask if you want to Continue with Launch. You should answer **No**. If you answer **Yes**, Eclipse will run the last version of the program known to work, and this is not useful for debugging.

When you select **No**, the error list can be found by clicking the **Problems** tab at the lower end of the screen, as in Figure 12.

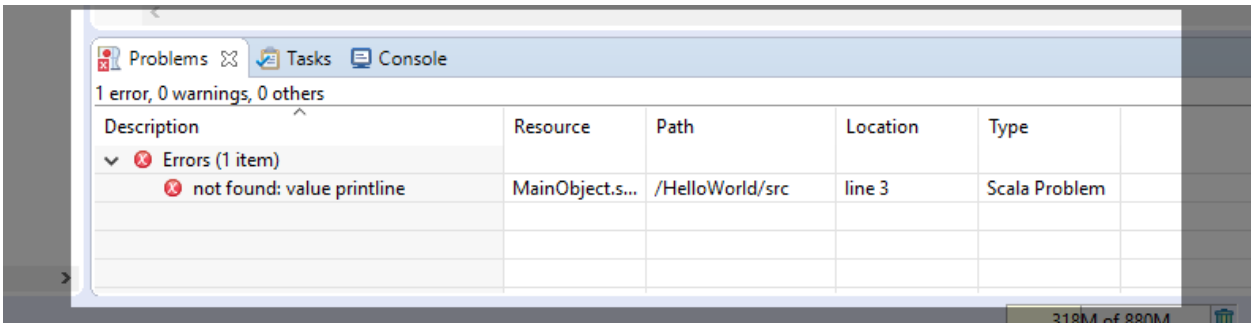

Figure 12: Problems Tab

Note: Many of the debugging techniques in Eclipse Java are also available in Eclipse Scala. I will focus on the main differences between Java and Scala, which means I will not be going into the details of debugging. If you wish to know more about debugging mechanisms and tools, consult* Java Succinctly 1*.

Chapter 2 Variables and Values

A variable is a name used to point to different values. For instance, we might create an integer variable called **personAge** and point it to **35**. We can point a variable to a different value, which means we can later change **personAge** and point it to **36**. In this sense, Scala variables are similar to references in other languages. In Scala, we define variables by using the **var** keyword.

A value is a fixed quantity or object. Values do not change, and they are pointed to by variables. In other languages, values are called constants, and we might define a value called **PI** and set it to **3.14159**. In Scala, we define values by using the **val** keyword.

Code Listing 3: Setting and Changing Vars and Vals

```
object MainObject {
	def main(args: Array[String]): Unit = {
		// Create a variable called myVariable.
		var myVariable = 0

		// Create a value called myValue.
		val myValue = 0

		// We can change the value that myVariable points to:
		myVariable = 10

		// But we can't change a val! The following line is an error!
		myValue = 10
	}
}
```

In Code Listing 3, we create a variable called **myVariable** and set it to **10**. We also create a value called **myValue**, again set to **0**. Next, we change the setting of **myVariable** to **10**. This is fine because variables can be set to many different settings throughout a program. But in the next line we try to set **myValue** to **10**—this line is an error, and I have highlighted it in red. Note that we cannot reassign a value. In order to reassign a **val** would be something like reassigning a meaning to the number 3 or to Pi—the operation makes no sense and is not legal.

The syntax for defining a variable begins with the **var** or **val** keyword, followed by the identifier name, such as **myVariable** or **myValue**. We follow this with the assignment operator "**=**" and supply a value, variable, or literal. This method, used in Code Listing 3, is a shorthand syntax. Scala will infer the data type for the variable from the initial assignment.

We can also explicitly state the data type for the variable by including a colon and the name of the data type (we will look at all the available data types shortly). Code Listing 4 shows some examples of using this longer syntax to define **Int**, **String**, and **Double** variables by specifying the data types.

Code Listing 4: Specifying Data Types

```
object MainObject {
	def main(args: Array[String]): Unit = {
		// Create an Int variable called myInt
		var myInt: Int = 0

		// Create a String variable called personName
		var personName: String = "Thomas"

		// Create a double val set to the Golden Ratio
		val goldenRatio: Double = 1.61803398875
	}
}
```

Identifier names

An identifier is a name we use to stand for something in our program. Identifiers are used to name **`val`** and **`var`**, as well functions, classes, and objects. Scala is flexible when it comes to naming identifiers. There are almost no restrictions at all, unlike languages such as Java, which do not allow arbitrary symbols. By contrast, Scala allows all types of string to be identifiers, including options like "**`&`**", and "**`Days of the week!`**".

Simple identifiers can be made in Scala much the same as with other languages. These identifiers consist of a string of characters that begin with a letter or underscore. The string can contain digits, but it cannot begin with a digit. For example: **`userName`**, **`_height`**, **`record56`**, **`Square_Root`**.

We can also use operator symbols as identifiers. This is often the case when we name member methods that are to act as operators for our objects in object-oriented programming. We will look at naming member methods when we look at classes. For example: **`+`**, **`++`**, **`:::`**. Generally, naming our regular variables with these symbols or strings of these symbols is not a good idea because doing so can make the code difficult to read.

Finally, we can use back quotes to delimit arbitrary strings. These strings can contain spaces, symbols, digits, anything at all. The identifier name is the string without the back quotes. We can use the identifier name by itself, but only in certain circumstances because the compiler sometimes needs the back quotes in order to understand where our variable names begin and end. Code Listing 5 shows some examples of Scala identifiers with simple and complex names.

Code Listing 5: Identifier Examples

```
object MainObject {
    def main(args: Array[String]): Unit = {
		// Identifiers for variables are usually descriptive strings
		// of letters and digits:
		var root2 = Math.sqrt(2)
```

```
        // We can also define identifiers beginning with underscore:
        val _someVar = 23

        // But, we can define an identifier as a series of operators.
        // Note that this doesn't make sense in the current context, and
        // this type of identifier is much more useful when we are
        // defining classes in Object-Oriented programming!
        def +&^%(i: Int, y: Int): Int = 42

        // We can name a variable an arbitrary string of characters, but
        // sometimes we have to use back quotes to delimit the name:
        val `#*^`: Int = 623

        // And we can use back quotes to define arbitrary identifiers:
        val `my identifier has 4's in its name, and $ as well!` = 9

        // If it can, Scala will recognize the identifiers without quotes
        // even their name consists of arbitrary operator symbols:
        println("The value of #*^ is " + #*^)
        // We can include the back quotes if our names are confusing:
        println("The value of #*^ is " + `#*^`)


        // If the names of the variable have spaces, we need to use
        // back quotes because the Scala compiler
        // will split the name into tokens unless it is delimited
        // with back-quotes:
        println("The value of my silly val is: " +
               `my identifier a 4 in its name, and $ as well!`)

        // The following is indecipherable and will generate an error!
        //println("The value of my silly val is: " +
        //     my identifier has 4's in its name, and $ as well!)
    }
}
```

 ***Note: Although we are able to name our identifier's keywords like `def`, this is not a good idea (in fact you would have to use back quotes to do this). We should always try to name identifiers in a descriptive way, and we should never try to redefine keywords by creating identifiers with the same name.***

 ***Tip: It is conventional to use Camel Case to name Scala identifiers. Identifiers begin with a lowercase letter, and every following word within the identifier begins with an uppercase letter, such as `averageIncome` and `computePerimiter`. This is just a convention, and when we name classes, we typically use an uppercase letter to begin each word within the same name, such as `MyClass`. This makes it easy to differentiate between variables and classes.***

Scala is case sensitive, which means the identifiers **`MYID`** and **`myID`** are completely different identifiers, and **`def`** is a keyword, but **`DEF`** is not.

Data Types

Name	Type	Size (Bytes)	Size in Bits	Minimum	Maximum
Byte	Integer	1	8	-128	127
Short	Integer	2	16	-32768	32767
Int	Integer	4	32	-2147483648	2147483647
Long	Integer	8	64	-2^{63}	$2^{63}-1$
Float	Float	4	32	-3.4×10^{38}	3.4×10^{38}
Double	Float	8	64	-2.2×10^{308}	2.2×10^{308}
Boolean	Boolean	1	8	false	true
Char	Character	2	16	Unicode	Unicode

Figure 13: Data Types

The fundamental data types are the same as in Java, except they begin with an uppercase letter. The Scala compiler is often clever enough to deduce the data type from the context, so the data type can often be left out when we are defining variables and values, but, if we want to explicitly state the data types for our variables, we use the names in the first column of Figure 13. Figure 14 depicts an overview of some of the characteristics of the fundamental data types in Scala.

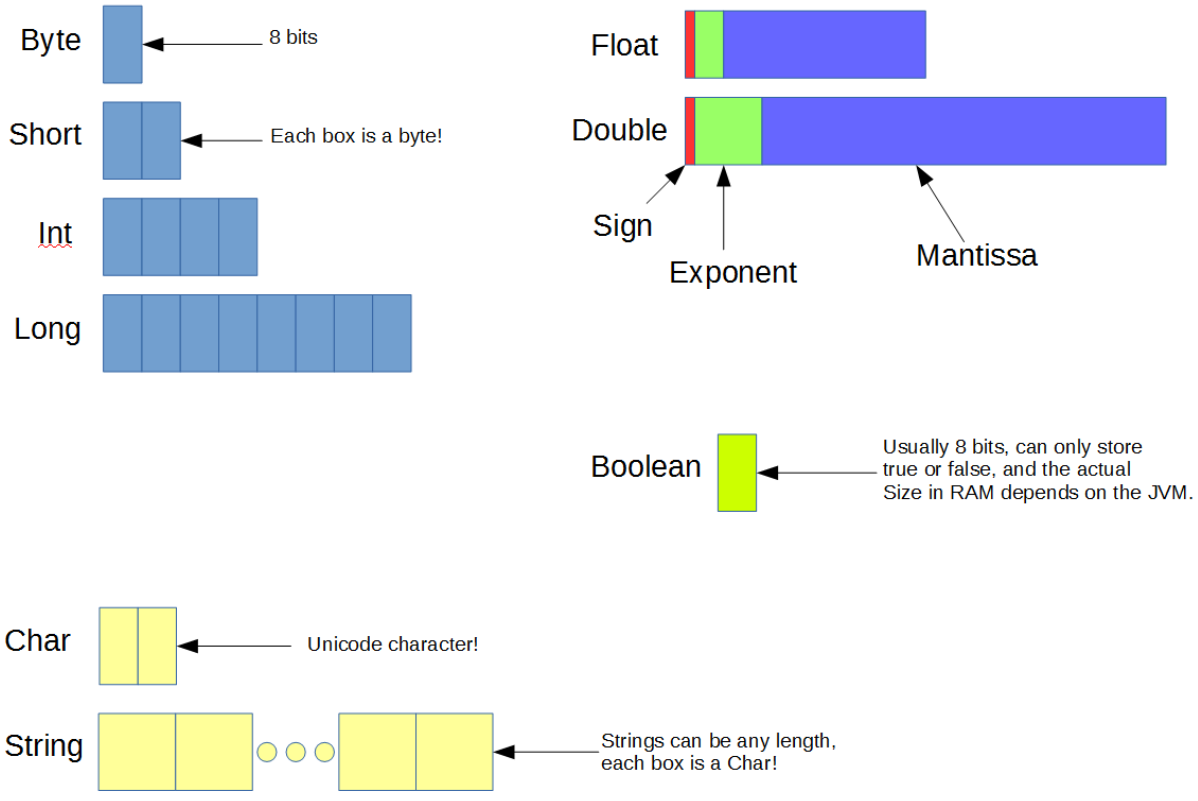

Figure 14: Overview of Fundamental Data Types

Code Listing 6 shows some examples of declaring and defining variables and values (note this listing has no main method and cannot be run).

Code Listing 6: Defining Variables and Values

```
var someInteger: Int = 190 // Declare and define an integer.
val someChar: Char = 'A' // Declare and define a character.
var someBool: Boolean = false // Declare and define a Boolean.
var someFloat = 3.14f // Declare and define a float.
var someDouble = Math.sqrt(2) // Declare and define a double.
```

In the the final two examples, I have not used a data type, but Scala knows that **`someFloat`** is supposed to be a floating-point number because the literal **`3.14f`** is a float (it ends with **`f`**, which is the suffix for a float). Likewise, **`someDouble`** will have the type of **`Double`** because **`Math.sqrt(2)`** is a function that returns **`Double`**.

The integers are whole numbers. For instance, an integer could be set to **178** or **-59**. The different integer types (**Byte**, **Short**, **Int**, and **Long**) are used when we need more or less range for our numbers. Bytes can only store between -128 and 127 inclusive, so if you have a variable you know will fall only between these values, you can save RAM and store the variable as a **Byte**. Otherwise, if your variable needs a lot of range, you might use a **Long**, because it has a range of -2^{63} up to $2^{63}-1$. We usually use **Int** for whole numbers and only use **Byte** or **Short** when we know the range is small and we want to conserve RAM, or when we need to interoperate with a system that uses one of these smaller data types. Likewise, it's rare to use a **Long** unless we know that the particular variables need the range.

Scala uses the same integer arithmetic as other languages. This means that operations result in truncation rather than rounding. For instance, 10/6 will give the result 1, even though the actual value, 1.66666, is nearer to 2. Integer arithmetic always truncates the fractional part of the answer and returns the remaining integer part as the result. If you need to know the remainder after division, this can be returned with % operator. So, 10/6 in terms of integer operations is 10/6, which equals 1 with the remainder returned 10%6, which equals 4; in other words, 10/6 equals 1 with remainder 4.

Floating-point numbers (**Float** and **Double**) are able to express fractional values such as **67.8** and **-99.24**. Floating-point arithmetic often incurs error, and there are many fractions that floating point cannot represent exactly. For instance, 1/3 is impossible for floating point to represent because Scala uses IEEE 754 standard, and this standard only allows exact representations of sums of perfect powers of 2. When we set a **Double** variable to 1/3, the number stored is very close 1/3, but not exact. This is sometimes important—for instance, when checking if two doubles are equal, we sometimes must consider a small amount of error, such that 0.333333333 would be equal to 0.333333332, because the 2 on the end is possibly a rounding error. Code Listing 7 shows an example of using **Math.abs** to test equality of doubles.

Code Listing 7: Testing Equality Between Doubles

```
object MainObject {
    def main(args: Array[String]): Unit = {
      // Define two variables which are mathematically
      // equal:
      var a = (10.0 / 3.0)
      var b = (1.0/3.0) * 2.0 * 5.0

      // This will not work! Testing the exact
      // values of doubles for equality is often
      // a waste of time!
      if(a == b)
            println("The two are equal!")
      else
            println("The two are not equal...")

      // Allowing some small error using Math.abs
      // makes more sense. The following report
      // that a and b are equal:
      if(Math.abs(a - b) < 0.0001)
            println("The two are equal!")
      else
```

```
            println("The two are not equal...")
        }
}
```

In Code Listing 7, we create two variables, **a** and **b**, which should theoretically be set to exactly the same value;—10/3 is mathematically identical to (1/3)*2*5. But in IEEE 754, we will get two different values for these expressions, so we should use **Math.abs** when we compare them, and we should allow for a small degree of error (**0.0001** in the example will report equal value as long as the doubles are similar to within 1/10000). The small value used for the comparison of floating-point types is usually called the epsilon value. We can use exactly the same technique when comparing **Float** values because Floats suffer from the same rounding errors.

Boolean variables are used in logical expressions in order to make decisions and for filtering. They have only two values: **true** or **false**.

The Char data type is used for characters and for Strings. It represents Unicode characters, such as **'A'** or **'@'.**

Literals

A literal is a value that appears in the code, such as **190** or **'A'**. They are used to set variables and values and also to form expressions. All of the literals are values, and like **val**, they cannot be redefined. We can, however, point variables to them.

Integer literals

Integer literals appear as whole numbers, such as **899** or **-77162**. They can have a negative sign to indicate values less than **0**. Integer literals without a suffix are read as base 10 or decimal literals, so that 899 means “eight hundred and ninety-nine.” Integer literals with the **0x** suffix are read as hexadecimal, or base 16 numbers. For instance, **0xff0a** and **0x772e** (hexadecimal is a positional notation, the same as decimal, except that there are 16 digits, 0 through to 9, A, B, C, D, E and F—for more information on hexadecimal, visit Wikipedia: https://en.wikipedia.org/wiki/Hexadecimal). Long integer literals end with L, such as **789827L** or **-898827L**. Long integer literals have a range of -2^{63} to $2^{63}-1$.

Note: In previous versions of Scala, we could use a leading '0' to denote an octal number. For instance, 037 would mean the decimal value 31. Octal literals are now obsolete, and placing a leading 0 at the beginning of an integer literal will cause an error.

Floating-point literals

Type **`Double`** literals contain a decimal point—for example, **`90.7`** or **`-178.5`**. Type **`Float`** literals can contain a decimal point, too, and they end with an '**`f`**'. For instance, **`271f`** or **`-90.872f`**. You can also use scientific notation for the **`Float`** and **`Double`** literals—for example, **`54.9e2`**, which is the same as **`5490.0`** (or 54.9 multiplied by 10 to the power of 2). You can use the '**`f`**' suffix along with scientific notation to create a **`Float`** literal too, e.g., **`16e-1f`** would mean 1.6 or 16 by 10 to the power of negative 1.

Other literals

Character literals are surrounded with single quotes, such as **`'A'`**, **`'%'`**, or **`'6'`**. Note that **`'6'`** is very different from the integer **`6`**. **`'6'`** is a Unicode character with the **`Int`** value of **`54`**. For a complete table of the Unicode characters, visit http://unicode-table.com/en/. There are also some escape sequences available as character literals: **`'\n'`** for new line, **`'\r'`** for carriage return, **`'\t'`** for tab, **`'\"'`** for double quotes. In order to use a Unicode code directly, we place the **`'\u`** suffix followed by the number, so that **`'\u0065'`** is the same as **`'A'`**, because **`'6'`** has a Unicode value of 65.

The Boolean literals are **`true`** and **`false`**. Code Listing 8 shows some Boolean literals. We can use the **`true`** and **`false`** keywords, and we can also use other literals along with logical operators such as '**`>`**' (which means greater than) in order to form logical expressions. In Code Listing 8, 2 is not greater than 5, which means the Boolean called **`twoGreaterThanFive`** will be set to **`false`**.

Code Listing 8: Boolean Literals

```
val myBoolean = true
var myOtherBoolean = false
val twoGreaterThanFive = 2 > 5
```

`String` is not a fundamental data type, but strings are so commonly used that we can introduce them with the other fundamental data types. **`String`** literals are formed by surrounding text with double quotes. We can also use triple-double quotes to denote multiline string literals. Multiline literals can include new line characters. Code Listing 9 shows two examples of string literals—a single line literal and a multiline literal.

Code Listing 9: String Literals and Multiline String Literals

```
var str = "This is a string!"
var multiLineString = """
This is also a string, only this one
can span many lines because it is delimited with
triple quotes! It can also contain single quotes,
like ".
"""
```

Comments

Comments are notes programmers place in the code for themselves and other programmers. Comments are ignored by the Scala compiler. Scala allows the same commenting as Java. We use // to specify a single line comment or to comment on the remaining text on a line, and we use /* and */ to include block comments (see Code Listing 10 for an example using single and multiline comments).

Code Listing 10: Comment Example

```
/* HelloWorld
* Displays the text 'Hello world' to the user
* CommandLine Args: None
* Returns: None
* */

object MainObject {
	def main(args: Array[String]): Unit = {
		// Print 'Hello world!' to the console:
		println("Hello world!") // Single line comment!
	}
}
```

 ***Note: Scala also allows special comments called ScalaDoc comments. These comments begin with /** and end with */. They are used to generate documentation for our code. For more information on the syntax and use of ScalaDoc comments, visit: http://docs.scala-lang.org/style/scaladoc.html.***

Casting

To cast is to change the data type of a variable, value, or literal. Casting in Scala is achieved by calling functions that each of the data types supply. For instance, to cast an **`Int`** to a **`String`**, we would use **`someInt.toString`**. To cast a **`Double`** to a **`Float`**, we would use **`someDouble.toFloat`**. Code Listing 11 shows examples of casting between the various types.

Code Listing 11: Casting

```
object MainObject {
	def main(args: Array[String]) {

		// Casting numerical types to other types and strings:
		var someDouble = 1.3
		println("As a float: " + someDouble.toFloat)
		println("As a char: " + someDouble.toChar)
		println("As an Int: " + someDouble.toInt)
		println("As a String: " + someDouble.toString)
```

```
            // Casting strings to numerical types:
            val myInt = "192".toInt
            val myFloat = "192.2".toFloat

        }
}
```

 ***Note: The `toInt` method is a digit parsing method, which means if we use any symbols not available to the integers, we will cause an error. In order to cast the string "`72.5`" to an integer, we first need to convert it to a `Double` or `Float`, then cast it to an `Int`.***

 ***Note: Casting a `Float` or `Double` to `Int` uses truncation. This is true when we change a `Float` type to a `Short`, `Int`, or `Long`. Numbers are not rounded—they are truncated to the nearest whole value towards Zero.***

Chapter 3 Expressions and Functions

Expressions

An expression is a series of variables, values, operators, and literals we use to compute. For instance, a mathematical expression such as **`100+1`**, or **`99*(89+3)`**. There are several different types of expressions in Scala—arithmetic expressions, Boolean or logical expressions, and string expressions. Code Listing 12 shows some examples of using different types of expression.

Code Listing 12: Basic Expressions

```
object MainObject {
	def main(args: Array[String]): Unit = {
		78 * 9 // Integer expression
		41.3 + 99.7 // Double expression
		78.5f * (2.1f - 7.9) // Float expression
		89 < 78 || ((29 & 1) == 0) // Boolean expression
		("Hello" + " " + "world") * 3 // String expression
	}
}
```

In Code Listing 12, the expressions are not used for anything—they evaluate to some value, but we are not using the value, so the Scala IDE will give us warnings such as: “A pure expression does nothing...” Notice that we can use many of the common arithmetic operators, such as +, -, *, and /, for working with numerical values **`Int`**, **`Double`**, and **`Float`**.

Table 1 lists many of the available operators in Scala and provides some details as to how they are used. Many will look familiar if you are familiar with Java or other C-based languages, but Scala allows us to define arbitrary meanings to operator symbols, so that when we look into classes—and particularly lists and other collections—we will see that there are many more operators defined in Scala.

Table 1: Operators in Scala

Op	Name	Type	Example	Description
+	Addition	Arithmetic	**`someVar+3`**	Adds numerical values and concatenates strings
+	Unary positive	Arithmetic	**`+someVariable`**	Unary positive is useless, use 3 instead of +3
-	Subtraction	Arithmetic	**`someVar-10`**	Subtracts the second operand from the first

Op	Name	Type	Example	Description
-	Unary Negative	Arithmetic	`-myVariable`	Negates numerical values: for int this means 2's complement, for floats it flips the sign bit
*	Multiplication	Arithmetic	`28.3*45.67`	Multiplies two numerical values
/	Division	Arithmetic	`28.3/45.67`	Divides the first operand by the second, returns the quotient
%	Modulus	Arithmetic	`45%3`	Divides the first operand by the second, returns the remainder after division
==	Equal to	Relational	`myVar==100`	Determines if two operands are equal
!=	Not Equal to	Relational	`myVar != 100`	Determines if two operands are not equal
>	Greater than	Relational	`someVar > 90`	Determines if the first operand is greater than the second
<	Less than	Relational	`someVar < 90`	Determines if the first operand is less than the second
>=	Greater or Equal to	Relational	`100.5 >= 90.3`	Determines if the first operand is greater than or equal to the second
<=	Less or Equal to	Relational	`100.05 <= 90.3`	Determines if the first operand is less than or equal to the second
&	Bitwise AND	Bitwise	`someVar & someMask`	Performs the bitwise AND operation between corresponding bits of two operands

Op	Name	Type	Example	Description
|	Bitwise OR	Bitwise	`someVar | someOtherVar`	Performs the bitwise OR operation between corresponding bits of two operands
^	Bitwise XOR	Bitwise	`someVar ^ -1`	Performs the bitwise exclusive OR between corresponding bits of two operands
~	Bitwise complement	Bitwise	`~someVariable`	Flips all the bits of the operand so 0's become 1's and vice versa
>>	Arithmetic Shift right	Bitwise	`someInt>>2`	Shifts all the bits of the input right by the amount specified in the second operand (i.e. divides the operand by 2 to the power of the second operand)
<<	Bitwise shift left	Bitwise	`SomeInt<<2`	Shifts all the bits of the left by the amount specified in the second operand (i.e. multiplies the operand by 2 to the power of the second operand)
>>>	Bitwise Shift Right	Bitwise	`someInt>>>2`	Same as shift right, except 0's come in on the left instead of 1's. Use >> for quick division of signed integers, and >>> for shifting nonsign values or bit fields
&&	Logical AND	Logical	`(someExpression)&&(someOtherExpression)`	Performs a logical AND between two Boolean expressions, used to form logical expressions
||	Logical OR	Logical	`(someExpression)||(someOtherExpression)`	Performs a logical OR between two Boolean expressions, used to form logical expressions
!	Logical NOT	Logical	`!someBoolExpression`	Complements a Boolean value, used to form logical expressions

Op	Name	Type	Example	Description
=	Assignment Equals	Assignment	**someVar = 100**	Used to assign a value to a variable
+=	Addition Assignment	Assignment	**someVar += 10**	Used to add, then assign a value, someVar+=10 means add 10 to someVar
-=	Subtraction Assignment	Assignment	**someVar -= 10**	Used to subtract, then assign a value, someVar -= 10 means subtract 10 from someVar
*=	Multiplication Assignment	Assignment	**someVar *= 10**	Used to multiply, then assign a value, someVar *= 10 means multiply someVar by 10
/=	Division Assignment	Assignment	**someVar /= 10**	Used to divide, then assign a value, someVar /= 10 means divide someVar by 10
%=	Modulus Assignment	Assignment	**someVar %= 10**	Used to get the remainder after division of someVar and 10
>>=	Shift Right Assignment	Assignment	**someVar >>= 2**	Used to shift then assign a value, someVar >>= 2 means shift someVar right two bits
<<=	Shift Left Assignment	Assignment	**someVar <<= 2**	Used to shift then assign a value, someVar <<= 2 means shift someVar left by two bits
>>>=	Shift Right Zero Fill Assignment	Assignment	**someVar >>>2**	Same as >>, only fills the high-order bits with zeros
&=	AND Assignment	Assignment	**myVariable &= 7**	Used to perform bitwise AND then assign a value, someVar &= 7 means AND the bits of someVar with 7

Op	Name	Type	Example	Description
|=	OR Assignment	Assignment	`myVariable |= 7`	Used to perform bitwise OR then assign a value, someVar |= 7 means OR the bits of someVar with 7
^=	XOR Assignment	Assignment	`myVariable ^= 7`	Used to perform exclusive OR then assign a value, myVariable |= 7 means XOR the bits of myVariable with 7

When we form expressions, we can do so using parentheses to override the precedence of the operators. Scala is aware of the normal precedence of arithmetic operators, and employing parentheses is often necessary, especially when we are not sure of the exact precedence or when we want to write clear code. All expressions within parentheses are evaluated first. Brackets can be used in any type of expression, which means we can use them when concatenating strings and when joining Boolean expressions, logical statements, and arithmetic expressions.

The arithmetic that computers perform is always finite. That means addition will only give the correct answer so long as there is no overflow. For example, a byte storing 127+1 does not equal 128 because 128 is outside the range of a byte—it actually wraps around to -128. And, as we've seen, floating-point values are not able to store many exact fractions. They often rely on rounding.

Boolean operators are things like <, >, ||, and ==. They allow us to form logical statements. The example of a Boolean expression in Code Listing 12 means "89 is less than 78 OR 29 ends in a 0." Neither of these statements is true, which means this line will evaluate to false.

The final example in Code Listing 12 is a string expression. We can add strings together using the + operator. We can also use the multiplication operation *, which will add the same string multiple times. The line "`("Hello" + " " + "world") * 3`" will evaluate to "Hello worldHello worldHello world," because we multiply the string "Hello"+ " " + "world" by three.

Creating and calling functions

Functions begin with the **`def`** keyword, which is short for define. A function is similar to any other data type, except that it is evaluated when it is used rather than when it is defined. Functions return values and take parameters, and we can use them to enable code reuse when we have some expression of sequence of statements that we need to execute many times.

After the **def** keyword, we supply an identifier for the function followed by the parameter list in brackets—for example, **def someFunction(parameters)**. The parameter list consists of a comma-separated list of variables that are to be passed to the function in order for it to compute. For instance, **def someFunction(myInt: Int)** and **def someTwoParameterFunction(a: Float, b: Double)**. If there are no parameters, we use () as the parameter list or leave the parameters off, such as with **def someFunction()** and **def someFunction**. Code Listing 13 shows a simple Hello World function that takes no parameters and returns nothing.

Code Listing 13: Function with No Params or Return

```
object MainObject {
	def main(args: Array[String]): Unit = {

		// Define a function with no parameters:
		def helloFunction {
			println("Hello world!")
		}

		// Calling a function with no parameters:
		helloFunction

	}
}
```

After the parameter list (if there is one), we specify the return type. The return type begins with a colon and, if we are using object-oriented programming, is followed by a data type such as **Int** or **Double** or some class. This is the value that the function computes and returns to the caller.

In Scala, we can nest functions. In Code Listing 13, the function called **helloFunction** is actually nested inside the body of the **main** method. This means that it is local to the main method and cannot be called outside of main.

Finally, we can supply an assignment operator, **=**, and specify the body of the function. If the body of the function is only a single statement, we do not need to enclose the body of the function with **{** and **}**. Code Listing 14 shows a complete example of defining and calling a function that takes parameters and returns a value.

Code Listing 14: Creating and Calling a Function

```
object MainObject {
	// Compute the square root of x using
	// the Babylonian method:
	def sqrt(x: Double): Double = {
		if(x < 0)
			-1 // Return -1 as an error value if x < 0
		var q: Double = x / 2
		for(i <- 1 to 20) {
			q = (q + x / q) / 2
```

```
            }
            // return q // We can return using the 'return' keyword
            q // Placing q by itself is the same as return q.
      }

      def main(args: Array[String]): Unit = {
            println("Square root of 61 is " + sqrt(61))
      }
}
```

Code Listing 14 shows the code used to compute the square root of a **`Double`** using the Babylonian method. There is a built-in square root function that will compute the root much faster, but Code Listing 14 illustrates of how to create and call a simple function. The code also uses a **`for`** loop, which is a control structure, and we will look at control structures in a moment.

Note: When we return from a function, Scala will assume that the final evaluated statement is meant to be returned. Code Listing 14 shows that the final statement to be evaluated is "`q`," therefore q will be returned from `sqrt`. If you prefer, you can supply a `return` statement to explicitly return `q`, but there is often no requirement for the `return` keyword.

Note: The return type of `Unit` means the same as void in other languages. It means that the function does not return a value.

Calling a function in Scala is similar to doing so in other languages. We supply the name of the function followed by the parameters enclosed in brackets, such as **`sqrt(56)`**.

Named arguments

We can use named arguments as well. Named arguments allow us to supply the arguments in a different order than the function specifies, or we can specify the arguments by name if that makes the code clearer. Code Listing 15 shows an example of using named arguments. Notice that in the final example, **`printInfo(age = 51, name = "Claire")`**, the arguments do not appear in the same order as they appear in the definition of the function **`printInfo`**. If you name one argument when calling, you must name them all.

Code Listing 15: Named Arguments

```
object MainObject {
      def main(args: Array[String]): Unit = {

            // Specify a function which takes several parameters:
            def printInfo(name: String, age: Int) {
                  println("Patient Name: " + name + " Age: " + age)
            }

            // Call function without named arguments:
```

```
            printInfo("Chris", 35)

            // Some examples using named arguments:
            printInfo(name = "Dennis", age = 190)
            printInfo(age = 51, name = "Claire")
      }
}
```

Default parameters

We can specify default parameters in our function definitions, which means that if the caller does not supply a value for the parameter, it will be set to the default value.

Code Listing 16: Default Values

```
object MainObject {
      def main(args: Array[String]): Unit = {

            def printInfo(name: String = "No name", age: Int = 0) {
                  println("Patient Name: " + name + " Age: " + age)
            }

            // Both parameters will take default values:
            printInfo()

            // Age will take default value:
            printInfo("Simpson")

            // Name will take default value:
            printInfo(age = 65)

      }
}
```

Code Listing 16 shows some examples of using default values for function parameters. First, in the function's parameters list, we supply the default values for any or all of the parameters by specifying some literal after an equals. Then, when we call the function, we can supply any or all of the default values of some specific value or we can leave them to default. Notice that we can use named arguments in conjunction with default values, as in the function call **`printInfo(age=65)`**, which will call the function with the **`name`** argument defaulting to "**`No name`**".

Functions as data

Functions are just data, too. This point is not often clear when programming high-level languages, but it is literally true—everything the computer does is just a bunch of 1's and 0's. We can easily create a variable, point it to a function, then call that function by using our variable (see Code Listing 17). This is something like a function pointer, but the syntax in Scala is intuitive and easy to read.

Code Listing 17: Functions as Data

```
object MainObject {
	// Simple function that doubles the input:
	def doubleInput(i: Int): Int = {
		i + i
	}

	def main(args: Array[String]) {
		// Point val c to doubleInput, the _ means any
		val c = doubleInput _

		// Call the function doubleInput:
		println("Double 6 is " + c(6))
	}
}
```

In Code Listing 17, note the assignation of the function **doubleInput** to the **val** called **c**. We use the name of the function followed by the underscore. The underscore means all inputs; it is a wild card symbol. Normally, when we assign a value, we would need to specify something like **doubleInput(8)**, but if we use the underscore to mean any input, we'll get the variable **c** pointing to the function itself, and we can call **c** using the standard syntax for calling **doubleInput**.

Variable parameters

With the final parameter to a function, we can indicate that there might be a variable number of arguments. This is useful when we want to sum a number of values and we do not necessarily know how many there will be. When we use a variable number of arguments, the argument must be the final value in the parameter list.

Code Listing 18: Variable Arguments

```
object MainObject {
	def main(args: Array[String]) {

		// Function with variable argument list:
		def minimum(args: Int*): Int = {

			// Base case, return 0 if there's no
```

```
                // arguments:
                if(args.length == 0)
                        return 0

                // Otherwise, find the minimum:
                var min = args(0)
                for(i <- args) {
                        if(i < min)
                                min = i
                }

                // Return the smallest number from the list:
                min
        }

        // Call minimum
        println("Minimum: " + minimum(13, -30, 2, -17, 37))
    }
}
```

In Code Listing 18, the function called **minimum** can take any number of **Int** parameters. The syntax used for this mechanism defines the final parameter of the function's parameter list with a *****, such as **(args: Int*)**. Notice we call the function in the usual way, but the number of **Int** parameters can be anything at all. Code Listing 18 uses if statements. For loops, we'll look at control structures in the next section.

The **args** parameter becomes an **Array**. We will look at arrays later. Note also that there is a built-in function for lists called **min** that we could have called to get the minimum with less code.

Evaluation of functions

Another way to think about functions in Scala is to consider that a function is an expression that is evaluated when it is used rather than when it is initially set. Imagine we have a variable called **n** set to some number. If we define a second variable, **x**, and we point it to **n**, the value of **x** will be determined when the assignment occurs.

But, imagine if we have another variable, **y**. In that case, we can use **def** to assign the value of **n** to the variable **y**. The difference is that the value of **y** will be evaluated when it is used rather than when **y** is defined. So, if the value of **n** changes, the value of **y** will also change. See Code Listing 19 for a basic example of this mechanism.

Code Listing 19: Def Evaluation

```
object MainObject {
    def main(args: Array[String]): Unit = {
        // Define a variable n:
        var n = 90
        // Assign the value of n to a new variable, x
```

```
            var x = n

            // Assign the expression 'n' to a function, y
            def y = n

            // Print out the values of our variables:
            println("Before changing 'n':")
            println("The value of e is " + n)
            println("The value of x is " + x)
            println("The value of y is " + y)

            // Change the value of the 'n' variable:
            n += 1

            // Print out the values of our variables again:
            println("After changing 'n':")
            println("The value of e is " + n)
            println("The value of x is " + x)
            println("The value of y is " + y)
        }
}
```

In Code Listing 19, the output shows that after the variable **e** has changed, the value that **x** points to is still the original value of **e**, even though **x** is a **var** and **e** has changed. This means that the value of a variable is evaluated when the variable is defined—**x** points to the **val** of **e** when it is defined, and changing **e** has no effect on **x** after **x**'s definition. However, the value of **y** changes when we change the value of **e** because we used **def** to define **y** as a function. This means that the value of a function is evaluated when we use the function. The output from running the program from Code Listing 19 is presented in Code Listing 20.

Code Listing 20: Def Evaluation Output

```
Before changing 'e':

The value of e is 90

The value of x is 90

The value of y is 90

After changing 'e':

The value of e is 91

The value of x is 90

The value of y is 91
```

Code Listing 20 shows only a very basic use of a function, and in this example our variable (or function) called **y** is little more than pointer to the value of **e**. But we can also define our function, **y**, to include a complex expression or even a code block with many lines of code.

Chapter 4 Control Structures

Control structures are used to allow certain sections of code to loop or to be executed based on a condition. Programs are normally executed one line at a time from top to bottom. Control structures allow us to change this execution order.

Looking at control structures requires us to first consider arrays, lists, and other data types where control structures are used. I will present control structures first, then collections, but I will be using collections in the code for both chapters.

"If" statements

"If" statements in Scala are similar to those of other C-based languages. We can use "if, else-if, and else" blocks in order to route our code execution based on conditions. One interesting point about Scala "if" statements is that they evaluate to something; in other words, they return a value. We will see this mechanism in a moment, but for now, Code Listing 21 shows a basic example of an If/Else If/Else block.

Code Listing 21: Basic If Blocks

```
object MainObject {
	def main(args: Array[String]): Unit = {

		def x = 100
		def y = 200

		if(x < y)
			println(x + " is smaller than " + y)

		else if(x > y) { // Can use { } for multiple lines of code!
			println(x + " is greater than " + y)
		}

		else // Can finish if/else if with a final else:
			println(x + " is equal to " + y)
	}
}
```

"If" conditions work the same way as Java. We begin with an "if," followed by any number of "else-if" conditions, and end with an optional "else" block. Only a single "if" or "else-if" code block will execute, and if none of the previous "if" or "else-if" blocks execute, the "else" block (if supplied) will execute. We supply a condition after the keyword "if," then supply a code block for the program to execute when the condition is true. After the "if," we can supply any number of "else-if" blocks, each with its own condition and code block. Finally, we supply an "else" block at the end.

Code Listing 22 shows an interesting difference between Scala and Java—"if" statements, and indeed the entire If/Else block, actually evaluate to a **val**. This is particularly important when we are filtering lists using **foreach** (which we will look at shortly). In Code Listing 22, 100 is less than 200, so the **val** called **resultFromIf** will be set to -1.

Code Listing 22: If Blocks Evaluate to Values

```
object MainObject {
	def main(args: Array[String]): Unit = {
		def x = 100
		def y = 200

		// Creating a val from an if block return:
		val resultFromIf = {
			if(x < y) -1
			else if(x > y) 1
			else 0
		}
		println("The result is: " + resultFromIf)
	}
}
```

For loops

For loops are used to execute a section of code for some specified number of times. Code Listing 23 counts to 10 using a for loop.

Code Listing 23: Counting with a For Loop

```
object MainObject {
	def main(args: Array[String]) {
		for(i <- 1 to 10) {
			println(i)
		}
	}
}
```

The basic syntax for a for loop is **for(variable <- range)**, where range specifies the range over which the for loop iterates. The variable, **i** in Code Listing 23, is set to 1 first, then the body of the for loop executes. Next, the variable is set to 2, then 3, etc. Each time the variable is incremented, the loop body runs and a new number is printed to the console using **println**.

The range of the loop is specified as **(i <- 1 to 10)**, which means the variable **i** will become consecutive whole numbers (**Int**) from **1** all the way up to and including **10**.

We can also iterate through collections using for loops. We will look at this when we cover collection types.

It is important to note that Scala uses different scoping rules than C++ and other languages. If we have a variable called **myVariable** and we implement a for loop with a counter with the same name, the counter variable will not be the same as the variable defined outside the for loop. Code Listing 24 shows an example of this behavior. The outer **myVariable** in Code Listing 24 will not change, and the variable that iterates through the loop has the same name, but otherwise it is completely distinct.

Code Listing 24: Variable Scope in For Loops

```
object MainObject {
def main(args: Array[String]): Unit = {
		var myVariable: Int = 0

		// For loop with counter:
		for(myVariable <- 1 to 10) {
			// Output the for loop's myVariable:
			println("Value of counter: " + myVariable)
		}

		// Output the value of the original myVariable.
		println("Value of local myVariable: " + myVariable)

	}
}
```

until vs. to

In a for loop, if we use the **until** keyword when specifying our range, Scala will count up to, but not include, the second number. If we use the **to** keyword, the second number will be included. For example, Code Listing 25 shows exactly the same example as Code Listing 24, except that we have used **until** in place of **to**. In Code Listing 24, the loop counts to and includes 10, but when we use **until**, the program counts up to but does not include the 10. The use of **until** rather than **to** is common for iterating through **Array** and other collection elements because the indices of these collection types are **0**-based and a collection with five items would have items numbered **0 until 5** or (0, 1, 2, 3 and 4).

Code Listing 25: Counting to (but Not Including) 10

```
object MainObject {
	def main(args: Array[String]) {
		// Counts: 1, 2, 3, 4, 5, 6, 7, 8, 9
		for(i <- 1 until 10) {
			println(i)
		}
	}
}
```

Multiple range for loops

We can supply multiple ranges for a **for** loop by separating each range in the for loop with a semicolon. In this case, the right-most variable counts up first, then the next to the left increments, and the right-most counts up again. In this way, the program will iterate through all permutations of the ranges. Code Listing 26 shows a simple example of using two ranges.

Code Listing 26: Using Multiple Ranges

```
object MainObject {
	def main(args: Array[String]) {
		// Two ranges for loop:
		//	variable called num will count 1 to 10,
		//	while den is set to 1. Then den will
		//	increment, and num will count 1 to 10
		//	again, etc.
		for(den <- 1 to 10; num <- 1 to 10){
			println(num + " divided by " + den + " is " +
				(num.toDouble / den.toDouble));

		}
	}
}
```

Code Listing 26 sets the variable **den** to 1, then it also sets the variable **num** to 1. It next executes the loop body 10 times, incrementing **num** each time and printing the results of **num**/**den**. When **num** has reached 10, the loop starts again, but **den** is incremented. The program will print 100 lines of output to the screen, and it will divide all combinations of the numbers from 1 to 10, starting from 1/1 and working all the way to 10/10.

Tip: Multiple range for loops are particularly important for iterating through multidimensional arrays. (We will look at multidimensional arrays later.) But the method for iterating through a multidimensional array is to set each range of a multiple range for loop to count through a dimension of the array. So, if we have an array with three dimensions with the sizes of the dimensions being 7, 8, and 9, we could use for (x <- 0 until 7; y <- 0 until 8; z <- 0 until 9).

For loop filters

We can add a condition to the ranges we count in a for loop. This will cause the body of the loop to be executed only when the condition is true. Code Listing 27 shows an example of a for loop with a condition—that the program prints out the even numbers between 1 and 100. Using the output from an “if” statement leads to this example.

Code Listing 27: For Loop Conditions

```
object MainObject {
	def main(args: Array[String]) {
```

```
		// Print the even numbers using a filter:
		for(i <- 1 to 100 if i % 2 == 0) {
			println(i)
		}

		// Print even numbers without using a filter:
		for(i <- 1 to 50) {
			println(i * 2)
		}
	}
}
```

Code Listing 27 contains two examples of how to print the even numbers from 1 to 100. Notice the "**if**" statement in the middle of the first for loop. If **i % 2 == 0**, then **i** is even, and therefore this loop will filter out all the odd numbers. In the second example, we count from 1 to 50 and double the result. This will give the same output and will probably be faster to execute. In this particular example, the second method is preferable because we are iterating through the loop half as many times but we can also use for loop filters when iterating through lists of elements. In the case of iterating through list elements, we would not be able to use the second method to print only the even numbers from the list. We will look at lists again later, but Code Listing 28 shows an example of iterating through a list and filtering out the even numbers.

Code Listing 28: For Loop Filtering List Elements

```
object MainObject {
	def main(args: Array[String]) {

		// Define a list:
		val myNumbers: List[Int] = List(
			2, 6, 1, 7, 4
		)

		// Filter the even numbers:
		for(i <- myNumbers if(i % 2 == 0)) {
				println(i + " is even!")
		}
	}
}
```

While loops

While loops are used to execute a block of code repeatedly until a certain Boolean condition is **`false`**. The syntax for a while loop in Scala is **`while(condition)`**, where **`condition`** is anything that evaluates to **`true`** or **`false`**, i.e. **`Boolean`**. Each time the program encounters the while loop it will evaluate the condition. If the condition evaluates to **`true`**, the program will execute the loop body and repeat the **`while`** loop's condition check. If the condition evaluates to **`false`**, the program will skip the loop body and continue execution after the **`while`** loop. Code Listing 29 is a simple number-guessing game that uses a while loop to repeatedly ask the user for a number until the hidden number is guessed.

Note: We use the `import` to import the `scala.io.StdIn.readInt` function, which is supplied as a standard part of Scala's libraries. In the code, we do not actually use the `readLine` function, which means the import could have read `import scala.io.StdIn.readInt`, but I left the readLine as an example of importing multiple functions from the same class.

Tip: The `Math.random` function generates a pseudorandom `Double` in the range from 0.0 to 1.0. It never generates the number 1.0 itself, but rather all numbers from 0.0 up to 1.0. In order to generate a random `Int` in the range from 0 to X (not including X), we can use `(Math.random * x).toInt`. In order to generate a random number from 1 to X (including X), we can use `(Math.random * x).toInt + 1`.

Code Listing 29: Guess-the-Number Game

```
import scala.io.StdIn.{readLine, readInt}

object MainObject {
      def main(args: Array[String]): Unit = {

            var userAnswer = 0
            var hiddenNumber = (Math.random * 1000).toInt + 1

            println("""I'm thinking of a random
number between 1 and 1000, inclusive.""")

            // Repeat the game while the user has not
            // guessed the number:
            while(userAnswer != hiddenNumber) {

                  print("Enter a number: ")
                  userAnswer = readInt            // Read an int from the user

                  // Give the user a hint:
                  if(hiddenNumber < userAnswer)
                        println("Lower")
                  else if(hiddenNumber > userAnswer)
                        println("Higher")
            }
```

```
            // The user won, print a message and quit.
            println("Yes, you got it, the hidden number was " +
                  hiddenNumber + "!")
      }
}
```

The program in Code Listing 29 generates a random number from 1 to 1000, and the user must guess the number. Notice the user of the **`while`** loop—we are saying that while the user's number is not identical to the **`hiddenNumber`**, the program should loop. When the user guesses the **`hiddenNumber`**, the variable **`userAnswer`** will equal **`hiddenNumber`** and the condition of the while loop (`userAnswer != hiddenNumber`) will be **`false`**. The program will stop executing the loop and begin execution after the body of the loop.

Do while loops

Do while loops are similar to while loops, except that the condition is checked at the end, after the loop's body. This means that do while loops are guaranteed to execute at least once. Code Listing 30 shows the same game as the while loop example, except that here the hidden number can be negative.

Code Listing 30: More Difficult Version of Guess the Number

```
import scala.io.StdIn.readInt

object MainObject {
      def main(args: Array[String]) {

            var userAnswer = 0

            // Select a random number from -1000 to 1000
            var hiddenNumber = (Math.random * 2001).toInt -1000

            println("""I'm thinking of a random
number between -1000 and 1000, inclusive.""")

            // Do while loops guaranteed to execute at least once!
            do {
            print("Enter a number: ")
            userAnswer = readInt      // Read the user's answer

            // Give the user a hint:
            if(hiddenNumber < userAnswer)
                  println("Lower")
            else if(hiddenNumber > userAnswer)
                  println("Higher")
            } while(userAnswer != hiddenNumber)

            println("Umm... no. Anyway, I'm tired of playing. See ya!")
      }
```

```
}
```

The basic syntax for a do while loop is **`do { body } while(condition)`**, where body is the body of the loop and where the condition is some value that evaluates to a **`Boolean`**. As with the while loop, the do while will continue to execute until the condition becomes **`false`**. Then it will drop below the do while and continue execution after the loop.

The reason a do while loop is better suited to this game is because the program initially sets the **`userAnswer`** variable to 0. If we use a **`while`** loop for the game's body and the program happens to randomly select the number 0, the program will assume this 0 is the user's guess and the user will win the game immediately. With a do while, we are guaranteed that the first value we check against our hidden number is actually the user's input, not just the default value.

Example programs

For the final part of this chapter, we will examine some slightly longer and more complex programs. While we learn the Scala language, we should note that we can already use the basics of the language to create important and interesting programs. The following programs are intended for use in the study of number theory, a field that deals primarily with the characteristics and patterns of whole numbers. These programs are definitely not designed optimally, and there are well-known algorithms that work much faster than those presented here, but these programs are useful for studying prime numbers and patterns with small integers.

Testing if a number is prime

The example in Code Listing 31 contains a simple, brute-force method for testing if a number is prime. The program runs through the integers 1 to 100 using a for loop and prints to the console **`true`** or **`false`** depending on whether or not the number is prime.

Code Listing 31: Testing Primality

```
object MainObject {
      def isPrime(j: Int): Boolean = {

            // Base cases:
            if(j < 2) return false
            else if(j == 2 || j == 3) return true
            else if(j % 2 == 0) return false

            // Find the highest number we need to check
            var sqrt = Math.sqrt(j)

            // First composite to test
            var factor = 3

            while(factor <= sqrt) {
```

```
                        // If j is divisible by the factor
                        if(j % factor == 0)
                                // Return false
                                return false

                        // Move factor up to the next odd number
                        factor += 2
                }

                // If j is not divisible by any factor up to
                // the square root of j, then j is prime!
                return true
                }

                def main(args: Array[String]): Unit = {
                        var i = 0
                        for(i <- 1 to 100) {
                                println(i + ": " + isPrime(i))
                }
        }
}
```

The Goldbach conjecture

The second sample program is designed for use in the study of a famous statement made by Christian Goldbach. Goldbach proposed that *every even number greater than 2 could be written as the sum of two primes*. Although it appears to be a perfectly simple statement, it has never been proven or disproven, and a proof either way would be an extraordinary event in mathematics and computer science.

A Goldbach partition is two primes that sum to a given integer. For instance, for the number 18 (which is even), a Goldbach partition might be 7 and 11—because 7 and 11 are both primes and they sum to 18. If you are able to find an even number greater than 2 that does not have any Goldbach partitions, you have managed to solve the problem and proven Goldbach was incorrect. Likewise, if you are able to discern some infallible reason that Goldbach's statement is true for all even numbers greater than 2, you have managed to prove Goldbach correct. At this point, even numbers with hundreds of digits have been checked, and every even number has been found to have one or more Goldbach partitions. Most mathematicians believe the conjecture to be true, but nobody has managed to prove without doubt that Goldbach's conjecture is a fact. The following program outputs all of the Goldbach Partitions for a given number. The user can use the input 0 to exit the program.

Code Listing 32: Goldbach Conjecture Partitions

```
import scala.io.StdIn.{readLine,readInt}

object MainObject {
        def isPrime(j: Int): Boolean = {
                // Base cases:
```

```
            if(j < 2) return false
            else if(j == 2 || j == 3) return true
            else if(j % 2 == 0) return false
            // Find the highest number we need to check
            var sqrt = Math.sqrt(j)

            // First composite to test
            var factor = 3

            while(factor <= sqrt) {
                  // If j is divisible by the factor
                  if(j % factor == 0)
                        // Return false
                        return false
                  // Move factor up to the next odd number
                  factor += 2
                  }
            // If j is not divisible by any factor up to
            // the square root of j, then j is prime!
            return true
      }

      // Function prints the Goldbach partitions of
      // a given Int
      def goldbachPartitions(j: Int): Unit = {
            println("Goldbach Partitions for " + j)
            var currentPartition = 2
            while(currentPartition <= j/2) {
                  if(isPrime(currentPartition)
                              && isPrime(j - currentPartition))
                        println("Partition: " + currentPartition + " and " +
                        (j - currentPartition))
                  currentPartition += 1
            }
      }

      // Main loops until the user inputs 0
      def main(args: Array[String]): Unit = {
            var input = -1
            while(input != 0) {
      input = scala.io.StdIn.readLine("Input Int (use 0 to exit): ").toInt;
                  if(input != 0)
                        goldbachPartitions(input)
                  else
                        println("Bye!")
            }
      }
}
```

Note: The Goldbach conjecture is just one example of a mathematical problem that we can already explore using the basics of Scala. There are many such problems and questions in mathematics, and the interested reader should look up the list of unsolved problems in mathematics on Wikipedia: https://en.wikipedia.org/wiki/List_of_unsolved_problems_in_mathematics.

Chapter 5 Arrays and Lists

Storing objects in collections is a common practice is Scala. Two of the most basic and fundamental collection types are the array and the list. In this section, we will examine how to store data in these simple collections, and we'll look at some of the rich collection features that Scala offers for manipulating these collections.

Arrays

Arrays store a collection of objects with the same data type in contiguous RAM. Arrays are of a fixed size, so that after the array is defined, we cannot add and remove items. If you want to add and remove items from an array-like structure, see the next section on lists.

Code Listing 33: Defining Arrays

```
// Array of 5 Ints

var myIntArray: Array[Int] = new Array[Int](5)

// Array of 10 Doubles

var myDoubleArray = new Array[Double](10)

// Array of 3 Strings

val myStringArray = new Array[String](3)
```

Code Listing 33 shows the definition of three arrays. In order to define an array, we use the **`var`** or **`val`** keyword, followed by an identifier name. We follow this with **`Array[dataType]`**, where **`dataType`** is whatever type we want the array to store. Then we use **`= new Array[dataType](count)`**, where **`count`** is the number of elements in the array. Alternatively, we can skip the redundant declaration part of the definition and use a shortcut notation, as in **`myDoubleArray`**.

Accessing and setting elements

In order to access elements of an array, either for setting or for reading, we use parentheses (normal brackets are used, as opposed to other C-based languages that use square brackets). It is very important to note that array access is 0-based. This means that when we create an array of n elements, the first element has an index of 0 and the final element has an index of n-1. Code Listing 34 shows several examples of setting elements of arrays manually, one at a time.

Code Listing 34: Setting and Accessing Elements

```
object MainObject {
	def main(args: Array[String]): Unit = {
		// Define a double array with 5 elements:
		val doubleArray: Array[Double] = new Array[Double](5)

		// Set the values of the array
		doubleArray(0) = 99.0
		doubleArray(1) = 25.5 / 100.0
		doubleArray(2) = Math.sqrt(10)
		doubleArray((7 >> 2) + 2) = 4.0
		doubleArray(4) = 3.14
		// doubleArray(5) = 100 // Illegal!

		// Access elements and output.
		println("doubleArray(0) = " + doubleArray(0))
		println("doubleArray(1) = " + doubleArray(1))
		println("doubleArray(2) = " + doubleArray(2))
		println("doubleArray(3) = " + doubleArray(3))
		println("doubleArray(4) = " + doubleArray(4))

		// println("doubleArray(5) = " + doubleArray(5))// Illegal!
	}
}
```

Code Listing 34 shows a basic example of a **`Double`** array with five elements. We can set the values of the elements by employing the array identifier followed by the index in brackets. Note that **`doubleArray(0)`** means the first element and **`doubleArray(4)`** means the final element. And there is no element (5)—elements are numbered 0 to n-1.

When we set the value of an array element, we can use any expression we like because each element is a perfectly normal double **`var`**. Note also that we can access elements with expressions. In Code Listing 34, the value of **`doubleArray(3)`** is set using an expression for the index: **`((7>>2)+2)`**. This expression evaluates to **`3`**. The expressions must evaluate to a positive integer when we use them to access array elements—there is no element 3.5 or -6 of an array.

Figure 15 shows an illustration of the double array from Code Listing 34 before and after the values of the elements are set.

Figure 15: Array before and after Settings Elements

Val vs. var arrays

Code Listing 35 illustrates the difference between a **val** and a **var** array. If we define a **val** array, we are not able to point the array identifier to **someOtherArray** because it is a **val**. However, we can change the elements of a **val** array. The elements of a **val** array are **var** and can be changed as needed.

Code Listing 35: Val vs. Var Arrays

```
object MainObject {
	def main(args: Array[String]): Unit = {

		// Define a val and var array:
		val myValArray = new Array[Int](10)
		var myVarArray = new Array[Int](10)

		// Define some other array:
		var someOtherArray = new Array[Int](10)

		// Set the var array to point to someOtherArray
		myVarArray = someOtherArray
		// myValArray = someOtherArray // Illegal!

		// However, we can change the elements of a val array!
		myValArray(0) = 100
	}
}
```

A **var** array can point to a new array, and in Code Listing 35 we create a **var** array, then we point it to **someOtherArray**. As with a **val** array, the elements of a **var** array are themselves **var** and we are free to change them to whatever values we need.

Multidimensional arrays

Multidimensional arrays are useful for storing objects in grids, box-like arrangements, or higher dimensions, and they are often very large in terms of how many elements they have and the amount of RAM they require. It is common to process them using nested loops or for loops with multiple iterators.

Code Listing 36: 2D Array

```
object MainObject {
	def main(args: Array[String]): Unit = {
		// Declare a 2D array of Int
		val array2D = Array.ofDim[Int](5, 5)

		// Set elements in the array:
		array2D(2)(0) = 100
		array2D(3)(4) = 99

		// Read elements:
		println("Element(0)(0): " + array2D(0)(0))
		println("Element(2)(0): " + array2D(2)(0))
		println("Element(3)(4): " + array2D(3)(4))
	}
}
```

Code Listing 36 shows how to create a multidimensional array. We use the syntax **val name =**, where **name** is the identifier for the array. We set this to **Array.ofDim[dataType]**, where **dataType** is the type for the elements of our array. Then we specify the size of the dimensions, which is **(5, 5)** in this example. This will create a 2D matrix of **Int**, and each element will be initialized to **0**. Figure 16 shows an illustration of the array from Code Listing 36 after the elements are set to 100 and 99, as in the Code Listing.

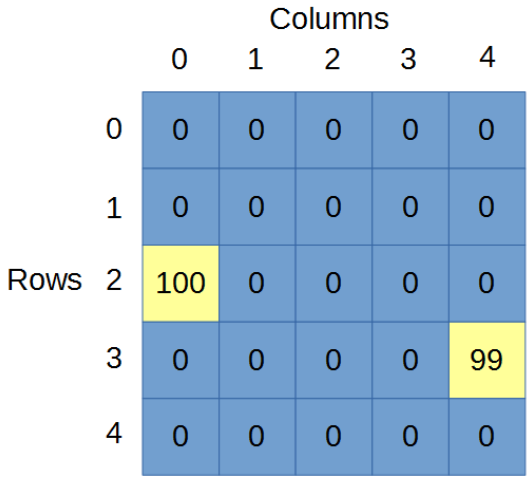

Figure 16: 2D Array

Figure 16 shows the **`array2D`** array drawn out with the first index representing the row and the second representing the column. This decision is arbitrary, and we could easily draw the array in other orientations.

The line in Code Listing 36 that reads **`array2D(2)(0) = 100`** sets the value of the array at row 2, column 0 to 100. And the line that reads **`array2D(3)(4) = 99`** sets the value of the array at row 3, column 4 to 99.

Higher dimensional arrays are also possible. We could define a 3D array with something like **`var my3DArray = Array.ofDim[Int](10, 10, 10)`**. As with 2D arrays, the orientation of the elements in a 3D array is purely conceptual, what matters is that we envisage and illustrate the array in the same way every time we access elements.

Note: Higher dimensional arrays can quickly consume massive amounts of memory. The total number of elements in an array is the product of the dimension sizes. So, if we have an Int array with three dimensions and each dimension has 100 elements, the total number of Int variables in our array is 100*100*100, which is one million. Each Int variable consumes four bytes of memory to store in the system, therefore a 100x100x100 Int array will require approximately four megabytes of memory.

Tip: If you have many arrays, or if you would like to use several of the helpful functions provided in Scala for use with arrays, you can import Array at the top of your program. When you import Array, you can define a new array using the shorthand var someArray = ofDim[Int](10, 10) rather than var someArray = Array.ofDim[Int](10, 10).

ArrayBuffer

An ArrayBuffer is similar to an array, except that we can add and remove items. Code Listing 37 shows a basic example of an ArrayBuffer. The program reads a list of doubles from the user, stores them in an ArrayBuffer, and computes the sum. Note that to use an ArrayBuffer, we import **`scala.collection.mutable.ArrayBuffer`**.

Code Listing 37: ArrayBuffer Basics

```
import scala.io.StdIn._
import scala.collection.mutable.ArrayBuffer

object MainObject {
	def main(args: Array[String]) {

		val userInput = ArrayBuffer[Double]()

		while(true) {
		// Output a prompt:
		print("Input a number (use -1.0 to continue): ")

		// Read some input:
		val x = readDouble

		// If the user inputs something other than -1
		// add it to the array buffer:
		if(x != -1)
			userInput += x // += adds the item to the end end
		//userInput.insert(0, x)// We can also insert items at the start

		// When the user inputs -1:
		else {
			// Init a summation variable
			var sum = 0.0
			// Use a for loop to add the items together
			for(y <- userInput) {
				print("Adding " + y + " ")
				sum += y
			}

			// Output the sum of items:
			println("Sum is " + sum)

			return // Return from main
			}
		}
	}
}
```

We can add multiple items at once to an array by using the **++=** operator. We can also add multiple items at once to any position of the ArrayBuffer by supplying multiple values to the insert method. Code Listing 38 shows several examples of adding and removing single and multiple items from an ArrayBuffer.

Code Listing 38: Adding and Removing Items from ArrayBuffers

```
import scala.io.StdIn._
import scala.collection.mutable.ArrayBuffer

object MainObject {
	def printArrayBuffer(arr: ArrayBuffer[Int]) {

		print("Array Buffer: ")

		// Print out the values in the array buffer
		for(x <- arr)
			print(x + " ")

		// Print a new line:
		println
	}

	def main(args: Array[String]) {

		// Create a new ArrayBuffer
		val nums = new ArrayBuffer[Int]()

		// Add a 1 to end of the array buffer:
		nums += 1
		printArrayBuffer(nums)

		// Add multiple items at once to the end:
		nums ++= Array(2, 5)
		printArrayBuffer(nums)


		// Add a 3 and a 4 after position 2:
		nums.insert(2, 3, 4)
		printArrayBuffer(nums)

		// To remove an item by its index:
		nums.remove(3)
		printArrayBuffer(nums)

		// Remove 2 items beginning at index 1:
		nums.remove(1, 2)
		printArrayBuffer(nums)

	}
}
```

Note that iterating through an **ArrayBuffer** is similar to an **Array**. We can use a simple for loop as per Code Listing 38. We can also use the property called **ArrayBuffer.length** and loop through the items.

Lists

Lists are similar to arrays, except instead of being stored in contiguous memory, the elements are stored as a linked list. Lists are quick to traverse from start to finish, but they are slow to look up items in the middle. Also, we cannot change the items in a list, they are immutable. Code Listing 39 shows some examples of defining and traversing simple lists.

Note: The various collection types each have different implementations. This leads to different performance for different tasks. For instance, we can add items to the beginning or the end of a list in constant time; however, an ArrayBuffer adds elements to the end in constant time, but adding an element to the start takes linear time. For a complete comparison of the performance of certain tasks, see http://docs.scala-lang.org/overviews/collections/performance-characteristics.html.

Tip: When selecting a collection for an algorithm, we typically minimize the amount of time taken to perform the operations on the collection. If you frequently need to add items to the start of the collection, you should use a collection that is implemented as a linked list, such as a list. If you need to reference or index elements at arbitrary positions (such element number 1000 or element number 789), you should use a collection stored in contiguous memory, such as an Array.

Code Listing 39: Lists

```
object MainObject {
      def main(args: Array[String]) {

            // List of 3 integers:
            var integerList: List[Int] = List(100, 101, 102)

            // List of strings:
            var capitalCities: List[String] = List(
                  "Melbourne",
                  "Hobart",
                  "Brisbane",
                  "Sydney")

            // Concatenate items to a list:
            capitalCities = capitalCities.:::(List[String]("Darwin"))

            // Print out the items of a list:
            println(capitalCities)

            // Traverse a list:
            for(i <- integerList)
```

```
                    println("Element: " + i)
        }
}
```

We can also define lists using **::**, which is called **cons** and which is short for construct, and we can use **Nil**, which acts as the tail of the list. When you create lists in this way, you should always use **Nil** at the end to finish the list (see Code Listing 40, and note this is not a complete Code Listing and cannot be run as a program).

Code Listing 40: Lists with Cons and Nil

```
// Empty list:
var anEmptyList = Nil

// List with cons and Nil
var directions = "North" :: "South" :: "East" ::
       "West" :: Nil
```

There are other ways to create simple lists in Scala. For example, we can also use the **List.range** method, which allows us to quickly create a list of items in numerical order. There is also the **List.fill** method, which allows us to create a list of items all set to the same value, as in Code Listing 41.

Code Listing 41: Creating Lists with Range and Fill

```
// Create a list of 100 integers from 1 to 100:

var rangeList = List.range(1, 100)



// List filled with 10 copies of String 'Empty'

var filledList = List.fill(10)("Empty")
```

Lists have three very important methods: **head**, **tail**, and **isEmpty**. Method **head** points to the first element of the list, method **tail** points to all elements after the first. Method **isEmpty** is used to determine whether or not the list is empty. Code Listing 42 shows an example of using **head**, **tail**, and **isEmpty**

Code Listing 42: Head, Tail, and IsEmpty

```
// Empty list:
var anEmptyList = Nil
```

```
// List with cons and Nil
var directions = "North" :: "South" :: "East" :: "West" :: Nil

println("First element of directions: " + directions.head)
println("Final element of directions: " + directions.tail)
println("Directions is empty? " + directions.isEmpty)
println("anEmptyList is empty? " + anEmptyList.isEmpty)
```

Multiple dimensional lists

We can also create lists of lists. These are lists in which each element is itself a list. Conceptually, this is the same as creating a multiple dimensional list. Code Listing 43 shows an example of creating a simple 2D list of integers.

Code Listing 43: 2D Lists

```
// Create a list of lists:
var twoDList =
	List(
		List(1, 2, 3),
		List(4, 5, 6),
		List(7, 8, 9)
	)

// Loop1:
for(l1 <- twoDList) {
	println("Element: " + l1)
}

// Loop2:
// Traverse using nested for loops
for(l1 <- twoDList) {
	for(l2 <- l1) {
		println(l2)
	}
}
```

Code Listing 43 creates a list called **`twoDList`** that consists of three elements, each of which is a **`List`** itself. In order to traverse the list, we can use a simple for loop, but this will only access each of the inner lists. In order to traverse every element of the nested lists, we can nest for loops. The output of Code Listing 43 is shown in Code Listing 44.

Code Listing 44: Output from Code Listing 43

```
Element: List(1, 2, 3)

Element: List(4, 5, 6)
```

```
Element: List(7, 8, 9)

1

2

3

4

5

6

7

8

9
```

Useful methods on lists

There are many useful methods available for lists and the other collections, such as **`List.length`**, which returns the numbers of elements in the list; **`List.last`**, which returns the final element of the list; and **`List`**.**`first`**, which returns the first element in the list. The following section provides a few extra examples that use other operators and methods available to lists. The interested reader should look up the documentation for each of the collection types in order to gain a full appreciation of the diversity of these objects. The documentation for the **`List`** class is available from http://www.scala-lang.org/api/2.7.7/scala/List.html.

Tabulate method

We can also create lists using tabulate. This allows us to create complex patterns of items in our lists using expressions or even “if” statements for each item in the list.

Code Listing 45: Tabulated List

```
object MainObject {
	def main(args: Array[String]) {

	// Create a tabulated list:
	val tabulatedList = List.tabulate(10)(n =>
		if(n % 2 == 0) "" + n + " is even"
		else "" + n + " is odd")
```

```
        // Print the items of the list:
        for(s <- tabulatedList)
                println(s)
        }
}
```

Code Listing 45 shows an example of creating a tabulated list. The Code Listing uses the **`tabulate`** method and the **`=>`** operator in order to create a list of alternating **`n is even`** and **`n is odd`** elements. We will see more of the **`=>`** operator in a moment; for now the important aspect of this Code Listing is the **`tabulate`** method. For each element in the list, the "if" statement will be applied with the result, so that either **`n is even`** or **`n is odd`** will become the elements of our list.

Concatenate operator

In order to join two lists together, we use the concatenate operator, which is represented by three colons, **`:::`**. Code Listing 46 shows an example of using the concatenate operator to join two lists together and produce a third.

Code Listing 46: Concatenate Operator

```
object MainObject {
        def main(args: Array[String]) {

                // Create some lists
                var list1 = List(1, 2, 3)
                var list2 = List(4, 5, 6)

                // Concatenate list1 and list2
                var list3 = list1 ::: list2

                // Print 1, 2, 3, 4, 5, 6
                for(i <- list3)
                        println(i)
        }
}
```

Take, drop, and SplitAt

Code Listing 47: Take, Drop, and SplitAt

```
object MainObject {
        def main(args: Array[String]) {
                // Create a list:
                var integerList = List(
                        1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

```
            // Take: Prints List(1, 2, 3, 4, 5)
            println(integerList take 5)

            // Drop: Prints List(6, 7, 8, 9, 10)
            println(integerList drop 5)

            // SplitAt: Prints (List(1, 2, 3, 4, 5),List(6, 7, 8, 9, 10))
            println(integerList splitAt 5)
        }
}
```

Code Listing 47 shows an example of using **take**, **drop**, and **splitAt** operators on a list. The **take** method creates a list with a specified number of items. In the example, the number supplied is 5, which means the first five items of **integerList** will be returned as a new list.

The operator **drop** is the opposite of **take**—in fact, **drop** will remove x number of items from the list and return a new list. In the example, the number supplied for the drop is 5, so the first five items from the **integerList** will be removed, leaving 6, 7, 8, 9, and 10.

The final example shows how to split a list into smaller lists using the **splitAt** operator. The call to **splitAt** 5 will cause the list to be split into two smaller lists, the first containing elements 0 to index 4 (i.e. five elements) and the second containing elements from index 5 to tail.

Folding

Folding is a technique for working with lists. Imagine we want to take some input, x, and perform an operation on x with each item in a list, then return x. For instance, let's say we want to begin with 0 and add each integer from an **Int** list to compute the sum of the elements of a list. We can do this with a for loop without too much trouble, as per Code Listing 48.

Code Listing 48: Summing List Elements Using For Loop

```
object MainObject {
  def main(args: Array[String]): Unit = {

    // Define a list:
    val myList = List(1, 2, 3, 4, 5)

    // Define a summing variable:
    var sum = 0

    // Sum the elements using a for loop:
    for(listElement <- myList)
      sum += listElement

    // Output the total:
    println("Sum is: " + sum)
  }
}
```

Scala also offers another interesting approach to this problem called folding. Code Listing 49 shows the same example as Code Listing 48, only this time we use **foldLeft**.

Code Listing 49: Summing List Elements Using foldLeft

```
object MainObject {
  def main(args: Array[String]): Unit = {

    // Define a list:
    val myList = List(1, 2, 3, 4, 5)

    // Define the sum, foldLeft using a closure:
    var sum = myList.foldLeft(0)((x,y) => x+y)

    // Output the total:
    println("Sum is: " + sum)
  }
}
```

The **foldLeft** function belongs to the **List** class. It takes two parameters, the first, **(0)**, is an integer. The second is a function to perform (this function is actually a closure—we will look at closures in more detail in a separate chapter). In the Code Listing 49 example, we say that the value of **x** starts at **0**. Each of the list items is then passed to the closure **(x, y) => x+y**. Each list element acts as the **y** variable in the closure, and the value of **x** will sum the elements one after the other. If this is confusing, Chapter 9 will focus on the syntax of closures.

We can start the **x** variable at values other than **0**. For instance, if we begin the **x** variable at **10**, then the sum will be reported as **25** because **10+1+2+3+4+5** is **25.**

Tip: We can also use foldRight, which is the same as foldLeft, except that the iteration through the list occurs in the reverse order. When we use foldLeft, the list is iterated through from the first element to the last. With foldRight, the list is iterated through from the final item to the first.

This has been a very brief look at Scala's folding function. The operation is similar to reduce, and the interested reader can look up other topics, such as **foldRight** and reduce.

Chapter 6 Other Collection Types

Scala has many useful collection types. The most fundamental are the array and the list, but if we want to quickly implement various algorithms, we often use other data types, such as stacks, queues, and maps. In this section, we will look at some of the other useful collection types.

Stacks and Queues

A **Stack** is a LIFO data structure. We add items to the **Stack** using the **push** function, and we remove items using the **pop** function. The order that items are popped is the opposite of the way they are pushed. For instance, if we push the values 1, 2, and 3, the **Stack** will return 3, then 2, then 1 when we pop the items.

A **Queue** is similar to a **Stack** in that it also allows only two operations. For a **Queue**, the two operations are **enqueue** and **dequeue**. We use **enqueue** to add items to the **Queue**, and we use **dequeue** to remove items. A **Queue** returns items in the same order they are **enqueued**. A **Queue** is sometimes called a FIFO data structure, which is short for first-in-first-out. For instance, if we **enqueue** the items 1, 2, then 3, a **Queue** will **dequeue** them in the same order: 1, 2, then 3.

Code Listing 50 shows some basic operations using a **Stack**, and Code Listing 51 shows similar operations using a **Queue**. Note that when we use these data structures (and many others), we need to include an import in order to import the class from the appropriate library.

Code Listing 50: Basic Operations with Stacks

```
import scala.collection.mutable.Stack

object MainObject {
	def main(args: Array[String]): Unit = {

		// Define a stack
		var myStack = new Stack[Int]

		// Push a new item to the stack:
		myStack.push(89)
		println("Number of items: " + myStack.length)
		println("Item at the top of the stack: " + myStack.top)

		// Push a new item to the stack:
		myStack.push(21)
		println("Number of items: " + myStack.length)
		println("Item at the top of the stack: " + myStack.top)

		// Pop off the newest item:
		var itemFromStack = myStack.pop
```

```
            println("Popped item: " + itemFromStack)
            println("Number of items: " + myStack.length)
            println("Item at the top of the stack: " + myStack.top)

            // Push a new item to the stack:
            myStack.push(44)
            println("Number of items: " + myStack.length)
            println("Item at the top of the stack: " + myStack.top)

            // Pop off all remaining items:
            // Note: This is a stack, so the items are popped
            // off in reverse order!
            while(myStack.length != 0)
                  println("Popped item: " + myStack.pop)
      }
}
```

Code Listing 51: Basic Operations with Queues

```
import scala.collection.mutable.Queue

object MainObject {
      def main(args: Array[String]): Unit = {

            // Define a queue:
            var myQueue = new Queue[Int]

            // Add an item to the queue:
            myQueue.enqueue(47)
            println("Number of items: " + myQueue.length)
            println("Item at the front of the queue: " + myQueue.front)

            // Add another item to the queue:
            myQueue.enqueue(83)
            println("Number of items: " + myQueue.length)
            println("Item at the front of the queue: " + myQueue.front)

            // Remove the oldest item from the queue:
            var itemFromStack = myQueue.dequeue
            println("Dequeued item: " + itemFromStack)
            println("Number of items: " + myQueue.length)
            println("Item at the front of the queue: " + myQueue.front)

            // Add an item to the queue:
            myQueue.enqueue(23)
            println("Number of items: " + myQueue.length)
            println("Item at the front of the queue: " + myQueue.front)

            // Loop until the queue is empty:
            // Note this is a queue, so items will be dequeued
            // in the same order they were queued!
```

```
            while(myQueue.length != 0)
                  println("Dequeued item: " + myQueue.dequeue)
      }
}
```

Sets

Sets are a collection type that hold only distinct elements. Sets are a representation of a mathematical entity with the same name. They are designed to allow the same operations as mathematical sets—except that a mathematical set can be defined as containing an infinite number of items, whereas Scala sets contain a finite number of elements. Code Listing 52 shows some basic operations with sets.

Code Listing 52: Operations with Sets

```
object MainObject {
      def main(args: Array[String]): Unit = {

            // Define a Set
            val evenNumbers = Set(2, 4, 6, 8, 10, 12, 14)

            // Print out some properties:
            println("Head: " + evenNumbers.head)
            println("Tail: " + evenNumbers.tail)
            println("IsEmpty: " + evenNumbers.isEmpty)

            // Testing if the set contains 3:
            if(evenNumbers.contains(3))
                  println("Set contains 3!")
            else
                  println("Set does not contain 3...")

            // Test if the set contains 2:
            if(evenNumbers.contains(2))
                  println("Set contains 2!")
            else
                  println("Set does not contain 2...")
      }
}
```

In order to join two sets together, we use the **++** operator (the **++** operator forms the mathematical union of two sets). In Code Listing 52, we join a set containing **(1, 2, 3)** with another containing **(3, 4, 5)**. When we run the program from Code Listing 52, notice that the output shows **set3** containing **(5, 1, 2, 3, 4)**. Notice also that although **set1** and **set2** both contain **3**, the concatenated set contains only one copy of **3**. Code Listing 53 also shows that we can easily add and remove items using the **+** and **–** operators, respectively.

Code Listing 53: Adding and Removing Elements

```
object MainObject {
	def main(args: Array[String]): Unit = {

	// Define some sets:
	var set1 = Set(1, 2, 3)
	var set2 = Set(3, 4, 5)

	// Concatenate with ++ operator:
	var set3 = set1 ++ set2

	// Output the concatenated set:
	println("Set3: " + set3)

	// Adding items to a set:
	println("Set containing an extra 10: " +
		(set3 + 10))

	// Removing items from a set:
	println("Joined set without the 3's: " +
		((set1 ++ set2) - 3))
	}
}
```

Sets are designed to allow fast item lookups. But the order of the elements in a set is meaningless—notice that when we run the program from Code Listing 53, the final ordering of items, (5, 1, 2, 3, 4), is not the same as the order we specified the items in the original sets (indeed, depending on the implementation of the particular Java Runtime you have installed, the order in my machine might be different than in yours). This is because the implementation of sets employs hashing techniques. If the order of elements in your collection is important, you should not use a set, or you should use the Scala **`SortedSet`** collection. However, if you know that every element in your collection will be unique and you want fast item lookups, a **`Set`** is perfect.

Mutable sets

By default, we cannot add and remove items from a set—they are immutable (which means the elements are all fixed). Code Listing 53 showed how to add and remove items, but the example actually created a new set, it did not add and remove items from the immutable set. If you want to add and remove items from a set without creating a new set, use a mutable set (which means the elements are not fixed and we are free to change them) by importing **`scala.collections.mutable.set`**. In order to add items to a mutable set, we can use the **`+`** operator, and to remove items we can use the **`–`** operator. Also notice that when we create a mutable set, we do so using **`var someName = Set[dataType]()`**, where **`dataType`** is the type of data the set contains and **`someName`** is the identifier we want to use for the set.

Code Listing 54: Adding and Removing Items from a Set

```
import scala.io.StdIn.readInt
import scala.collection.mutable.Set

object MainObject {
	def main(args: Array[String]): Unit = {
		// Create a mutable set:
		var setOfInts = Set[Int]()
		var newNumber = 0

		while(newNumber != -1) {
			// Print a prompt:
			print("Input a number (use -1 to quit): ")

			// Read a new number:
			newNumber = readInt

			// If the set contains the new number, remove it:
			if(setOfInts.contains(newNumber))
				setOfInts = setOfInts - newNumber

			// Otherwise, add it (if not -1):
			else if (newNumber != -1)
				setOfInts = setOfInts + newNumber

			// Print out the items in the set so far:
			println("Set contains: " + setOfInts)
		}
	}
}
```

Code Listing 54 shows a program that uses sets to test an interesting phenomenon called “The Birthday Paradox.” The question used to demonstrate the phenomenon is: *How many people, on average, would you need in a room before it is likely that at least two people share a birthday?* The program in Code Listing 54 uses a set of integers from which we repeatedly generate random birthdays until there is a duplicate. At this point, we record the number of birthdays generated so far, add this to a total, and repeat. The experiment is repeated as many times as specified by the iterations variable—I have set this variable to 1,000,000. The more iterations we repeat, the closer we will get to finding the actual average number of people we would need in a room before two or more of them share a birthday.

The Birthday Paradox is not actually a paradox, but it is surprising how few people are needed in a room before two might share a birthday. The program also demonstrates the speed of sets for looking up items. There are a million trials, and the program will likely finish in a second or two on any modern desktop PC. Each trial contains multiple lookups of a set with many elements.

Code Listing 55: Birthday Paradox Tester

```
import scala.collection.mutable.Set

object MainObject {
	def main(args: Array[String]): Unit = {
		// Define a mutable set:
		var birthdays = Set[Int]()

		// Define how many trials to run:
		var iterations = 1000000

		// Set total to 0 birthdays counted so far:
		var totalBirthdays = 0.0

		println("Beginning trials...")

		// Repeat the experiment up to iterations times:
		for(i <- 1 to iterations) {

			// Reset the birthdays:
			var duplicateDetected = false
			birthdays.clear
			while(!duplicateDetected) {

				// Generate a new random birthday:
				val newBirthday = (Math.random() * 365.0).toInt

				// Check if the birthday exists in the set or not:
				if(birthdays.contains(newBirthday)) {
					totalBirthdays += birthdays.size.toDouble
					duplicateDetected = true
				}
				else {
					// Add the birthday to the set:
					birthdays += newBirthday
				}
			}
		}

		// Output the total and average number of days:
		println("Total birthays: " + totalBirthdays)
		println("Average birthdays before duplicate: " + (totalBirthdays /
iterations))
	}
}
```

As with mathematical sets, sets in Scala allow us to form new sets by selecting the intersecting items from two sets or from selecting the items that are not shared between sets. Also note that instead of concatenating sets with the **++** operator, we can use the OR operator **|**. See Code Listing 56 for an example of the **&**, **|**, and **&~** operators.

Code Listing 56: Set-Like Operations on Sets

```
import scala.collection.mutable.Set

object MainObject {
	def main(args: Array[String]): Unit = {

		// Define two sets:
		var set1 = Set(1, 5, 4, 6, 9)
		var set2 = Set(5, 3, 7, 1, 6)

		// Output the shared elements:
		// Note: & operator is the same as: set1.intersects(set2)
		println("Shared elements: " + (set1 & set2))


		// Using | combines all elements:
		println("All Elements: " + (set1 | set2))

		// Using &~ filters to items not shared between sets:
		// Note: &~ is the same as: set1.diff(set2)
		println("Elements in set1, not in set2: " + (set1 &~ set2))
		println("Elements in set2, not in set1: " + (set2 &~ set1))
	}
}
```

We can also filter and count elements in sets that match a particular Boolean expression. Code Listing 57 shows an example of using the filter function.

Code Listing 57: Counting Elements in Filtered Sets

```
object MainObject {
	def main(args: Array[String]): Unit = {

		// Define a set:
		val mySet = Set(1, 5, 4, 6, 9)

		// Filtering:
		println("Number of odd elements in mySet: " +
			mySet.count(x => x % 2 == 1))
		println("Number of even elements in mySet: " +
			mySet.count(x => x % 2 == 0))

		// For these operations, we can also create new sets,
		// instead of just counting elements:
		val evenNumbers = mySet.filter { x => x % 2 == 0 }
		val oddNumbers = mySet.filter { x => x % 2 == 1 }
		println("Set of Even Elements: " + evenNumbers)
		println("Set of Odd Elements: " + oddNumbers)
	}
}
```

Sets are extremely powerful and versatile, and this has necessarily been a brief introduction to them. For more information, consult the Scala documentation for the set class at *http://docs.scala-lang.org/overviews/collections/sets.html*.

Tuples

A **Tuple** is a collection of objects that can be of different types and that we can pass and use as a single entity. This is different from other collections, such as **Array**, that contain objects that all have the same type. **Tuples** are useful for many things, including returning multiple values from a function—instead of actually defining a function with multiple returns, we can pass a **Tuple** and modify its values to act as multiple returned values.

Code Listing 58: Defining Tuples

```
object MainObject {
	def main(args: Array[String]): Unit = {

		// Verbose syntax for tuple of 3 elements:
		val tupleSlow = new Tuple3(2, "Banana", 2.6)

		// Quick syntax for tuple of 3 elements:
		val tupleQuick = (1, "Pineapple", 3.5)

		// Many element tuple:
		val oneOne = (1, 1, "was", 'a', "racehorse",
			2, 2, "was", 1, 2, 1, 1, 1, 1, "race", 2,
			2, 1, 1, 2)
	}
}
```

Code Listing 58 shows the definition of three **Tuples**. The first example shows the verbose syntax in which we use the **new** operator and define a **Tuple** in the same way as we would any other object, i.e. calling the constructor and pass parameters.

The second example shows a simpler syntax for **Tuples**. We can omit the **new Tuple3** and simply specify the parameter list in brackets.

The final example uses the quick syntax, but the **Tuple** has many elements. At the time of writing, the latest version of Scala can contain from 1 to 22 number of elements.

The data type of the **Tuple** is implied by the items passed to the constructor. So the line **new Tuple3(2, "Banana", 2.6)** will create a **Tuple** with data types **Int**, **String**, and **Double**. Likewise, the final example creates a 20-element **Tuple** with data types (**Int**, **Int**, **String**, **Char**, …, **String**, **Int**, **Int**, **Int**, **Int**, **Int**).

Accessing elements of a Tuple

Code Listing 59: Accessing Tuple Elements

```
object MainObject {
	def main(args: Array[String]): Unit = {

		// Define two complex numbers as tuples:
		var complexNumberA = (1.5, 7.8)
		var complexNumberB = (2.6, 5.1)

		// Multiply them together to get complex product:
		var complexProduct = (
			complexNumberA._1 * complexNumberB._1 -
			complexNumberA._2 * complexNumberB._2,
			complexNumberA._1 * complexNumberB._2 +
			complexNumberA._2 * complexNumberB._1
			)

		// Output results:
		println("Complex product of " + complexNumberA + " and " +
complexNumberB +
			" is " + complexProduct)
	}
}
```

Code Listing 59 shows an example of accessing elements of tuples. The elements are numbered from 1 to N, where N is the number of items in the **`Tuple`**. Note that we define two complex numbers as **`Tuple2`** objects, then we multiply them together to produce the complex product. Notice also the use of **`complexNumberA._1`** in order to access the first element of **`complexNumberA`**.

When we print a **`Tuple`** to the console, Scala will surround the elements as a comma-separated list with brackets. So, when we print **`complexNumberA`**, Scala will output **`(1.5, 7.8)`**.

Code Listing 60 shows an example of using **`foreach`** to iterate over the items in a **`Tuple`**. The example will assign the elements of the **`Tuple`** to the variable **`x`** and will print each element out on a separate line.

Code Listing 60: Iterating over Elements of a Tuple

```
object MainObject {
	def main(args: Array[String]): Unit = {

		// Define a tuple:
		val tuple5 = ("One", 2, 3.0f, 4.0, '5')

		// Output elements by iterating over tuple:
		println("Elements of tuple: ")
		tuple5.productIterator.foreach { x => println(x) }
	}
```

```
}
```

 ***Note: It may seem awkward to access elements of a tuple as suchAndSuch._1. If you are wondering why we are not able to use the syntax suchAndSuch(1), it is because the (and) parentheses define a function implicitly, and functions need to have some specific return type—they are not able to return each of the possible types in the tuple.***

Naming elements of a Tuple

We can name the elements of a **Tuple**, then refer to them by name instead of index. Code Listing 61 shows an example of naming the elements of a **Tuple**.

Code Listing 61: Naming Elements of a Tuple

```
object MainObject {
	def main(args: Array[String]): Unit = {

		// Define a tuple:
		val point3D = (-9.5, 5.6, 7.2)

		// Name the elements of the tuple:
		val (x, y, z) = point3D

		// Print out the elements using names:
		println("Element x: " + x)
		println("Element y: " + y)
		println("Element z: " + z)
	}
}
```

Notice that in Code Listing 61 the names **x**, **y**, and **z** refer to the elements of the **Tuple** called **point3D**. This is not a method for naming the elements of **Tuples** in general, but only a method for naming the elements of a specific **Tuple**.

Two elements Tuples shortcut

Code Listing 62 shows a shorthand for creating **Tuple2** objects. We use the syntax “element1 -> element2” as in the definition of **point2D**. Note that we cannot create a **Tuple3** this way. The line **val point3D = -9.5 -> 5.6 -> 7.2** actually creates a **Tuple2** inside another **Tuple2**: ((-9.5, 5.6), 7.2).

Code Listing 62: Shorthand for Tuple2

```
object MainObject {
	def main(args: Array[String]): Unit = {
```

```
            // Short hand for two element tuple:
            val point2D = -9.5 -> 5.6

            // Be careful, the following is not a Tuple3!
            val point3D = -9.5 -> 5.6 -> 7.2

            // Print out the tuples:
            println(point2D)
            println(point3D)
      }
}
```

Maps and Tuples

One of the most common uses of **Tuples** is with **Maps**. A **Map** is a collection of Key/Value pairs, which means **Tuple2** is perfect. **Maps** are sometimes called mappings or associations; they represent a mapping of the keys to the values.

Maps come in two flavors: Immutable and Mutable. The default is Immutable, and in order to use a Mutable map, you should use **import scala.collection.mutable.map**. Code Listing 63 shows some examples of how to use an Immutable **Map**. Note that once an Immutable **Map** is created, the items are fixed.

Code Listing 63: Immutable Maps

```
object MainObject {
      def main(args: Array[String]): Unit = {

            // Immutable map:
            val staff = Map(1 -> "Tom", 2 -> "Tim", 3 -> "Jenny")
            val staff2 = Map(10 -> "Geoff", 7 -> "Sara")

            // Print out some info the staff map:
            println("Keys: " + staff.keys)
            println("Values: " + staff.values)
            println("IsEmpty: " + staff.isEmpty)

            // Concatenate two maps with the ++ operator:
            val staffConcat = staff ++ staff2
            println("All staff: " + staffConcat.values)

            // Access values by key:
            println("Element with key 1: " + staffConcat(1))
            println("Element with key 7: " + staffConcat(7))
            // The following will throw an exception because the key
            // does not exist:
            // println("Element with key 12: " + staffConcat(12))

            // To check if a key exists:
            if(staffConcat.contains(12))
```

```
                println("Element with key 12: " + staffConcat(12))
            else
                println("Element with key 12: Does not exist!")

            // Removing elements by key:
            val timGotFired = staffConcat - 2 // 2 is the key for Tim
            // Now timGotFired will be the same as staffConcat, but Tim has
            // been removed:
            println(timGotFired)
        }
}
```

Note: As with Sets, Scala's Maps are extremely useful and fast. There are many operations available for them, and the interested reader should have a look at http://docs.scala-lang.org/overviews/collections/maps.html for more information.

Mutable Maps

Mutable **Maps** are essentially the same as Immutable **Map**s, except that we can add and remove items without creating a new map each time.

Code Listing 64: Mutable Maps

```
import scala.collection.mutable.Map

object MainObject {
      def main(args: Array[String]): Unit = {

            // Create a new map object:
            val staff: Map[Int, String] = Map()

            // Adding tuples (key/value pairs) to a map with +=
            staff += (5 -> "Teddy")
            staff += (1 -> "Rene")
            staff += new Tuple2(3, "Ronnie")

            // Print out some info on the map:
            println("Keys: " + staff.keys)
            println("Values: " + staff.values)
            println("IsEmpty? " + staff.isEmpty)

            // To remove an item by key:
            staff -= 5
            println(staff) // Teddy got fired!

            staff += (5 -> "Dean") // Dean took Teddy's old key.

            // We are not able to add multiple items with the
            // same key so the following is illegal:
```

```
            // staff += (5, "Teddy")

            // Iterating through a map:
            for(i <- staff.keys) {
                  println("Staff Member ID: " + i + " -> " + staff(i))
            }
            // To set map elements, we use map(x)=xyz
            for(i <- staff.keys) {
                  staff(i) = "Teddy"
            }

            println(staff)
      }
}
```

Code Listing 64 shows the use of a Mutable map. The only real difference is that Mutuable maps can add items and change the values of the keys. Also, note that the operations for **Maps** are the same as those for sets because the keys for a map are a **Set**.

There are many other types of collection available in Scala. Each has a different implementation and is designed for different types of data and algorithms. The interested reader can visit the page http://docs.scala-lang.org/overviews/collections/concrete-mutable-collection-classes.html for more information on the available collection types.

Chapter 7 Classes and Objects

Scala is a language that combines functional and object-oriented paradigms. The object-oriented mechanisms are designed to allow us to create modules of reusable code called classes. A class is a collection of data and methods that operate on that data. We will see that classes are very similar to the objects we have been using all along—for example, the **`MainObject`**. The main difference between a class and an object is that a class is designed to have multiple instances of objects created from it, whereas an object is the only instance of its class.

If you are not familiar with Java's object-oriented programming mechanisms, I strongly suggest you read up on them. Scala is a language designed to address many of the shortcomings of the Java language. Object-oriented programming is all about defining our own data structures to reduce the overall amount of code in our projects and to allow our projects to be maintainable and scalable.

A class is a blueprint for creating objects. Objects are called instances of the class. All of the data types in Scala are objects, including **`Int`** and **`Double`**. When we specify a new **`var`** or **`val`**, we are using objects. The fields of our objects must be initialized, and, unlike with Java, in Scala we cannot create an object with uninitialized fields.

Object-oriented programming allows us a mechanism to combine data and functions that operate on this data. In Java-speak, these are called member variables and member methods. Member variables are variables that belong to the objects, and the member methods are the functions that the objects are able to perform. These can be accessed using the "**`.`**" operator, such as **`someString.length`**. Or, if you have an object with a member variable called **`height`**, you can use **`someClass.height`** to access this variable.

Variable names are used to point to objects. They are references to objects.So an object, such as the number 100, can potentially be referred to by many variables.

Classes

We can add classes to our existing files, but if the classes are complicated and contain a lot of code, it is sometimes better to add a separate code file to our project. We will look at two methods for adding new classes to our projects—in the first, we add a new file for the class. This keeps the code for our class separate from the other classes, but it means that our project has multiple files. Using the second method, we add new classes to existing files. In Scala, we can define multiple classes per file. This has the advantage of minimizing the number of files in our project, but the classes are all mixed together and this can sometimes become difficult to maintain. As a general rule of thumb, if a class is required by other classes, or if a class is complex and requires many methods, the class should be defined in a separate file. Otherwise, if the class is very simple and only used by one other class in our project, we might define the new class inside the same file as the existing class.

Adding a new class

Method 1: Adding a new class file

To add a new class file to your project, click **File > New > Scala Class**, as in Figure 17. You can also add a new class by right-clicking the project in the Package Explorer and selecting **New > Scala Class**.

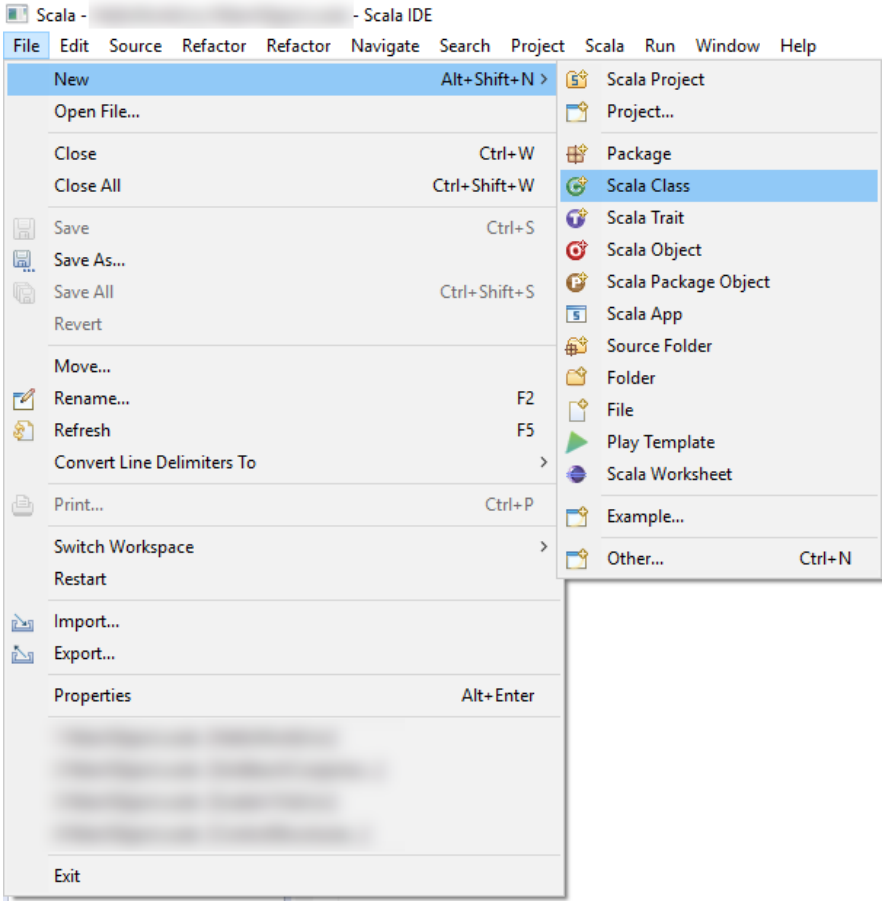

Figure 17: Adding a New Class File

You will be presented with the New File box, as in Figure 18. In this box, you can name your class in the box provided and click **Finish**. It is common to name classes with a leading uppercase letter because this makes it easy to differentiate identifiers that are class names from identifiers that are functions or variables.

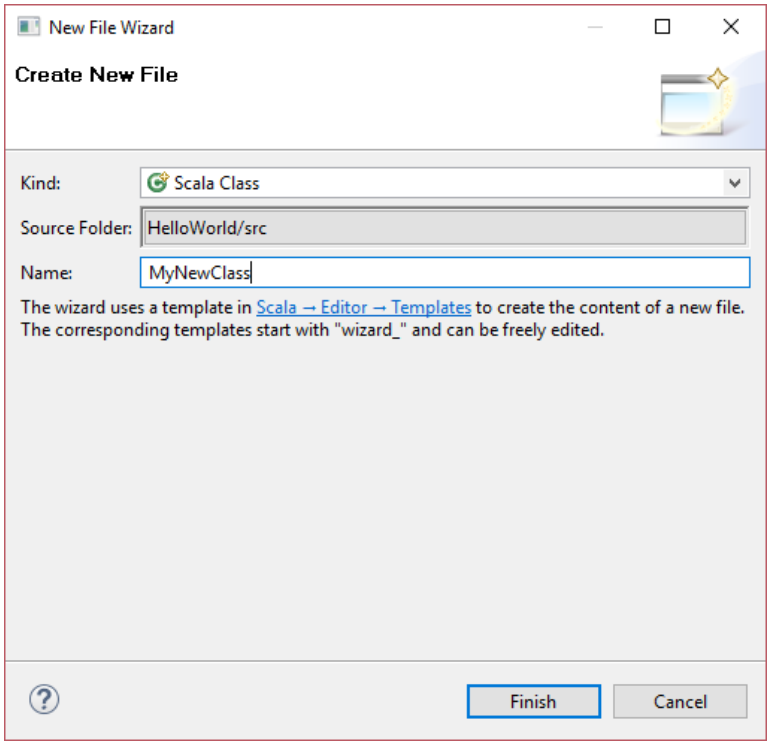

Figure 18: Adding a New Class Step 2

Eclipse will create a new file in your project and write the basic skeleton of a new class with the name provided, as in Code Listing 65.

Code Listing 65: A Blank Class

```
class MyNewClass {

}
```

In the Package Explorer, you will note that we now have a new file added to the src folder (as in Figure 19). We can edit the code for our new class by doubling-clicking its name to open the code in the code view.

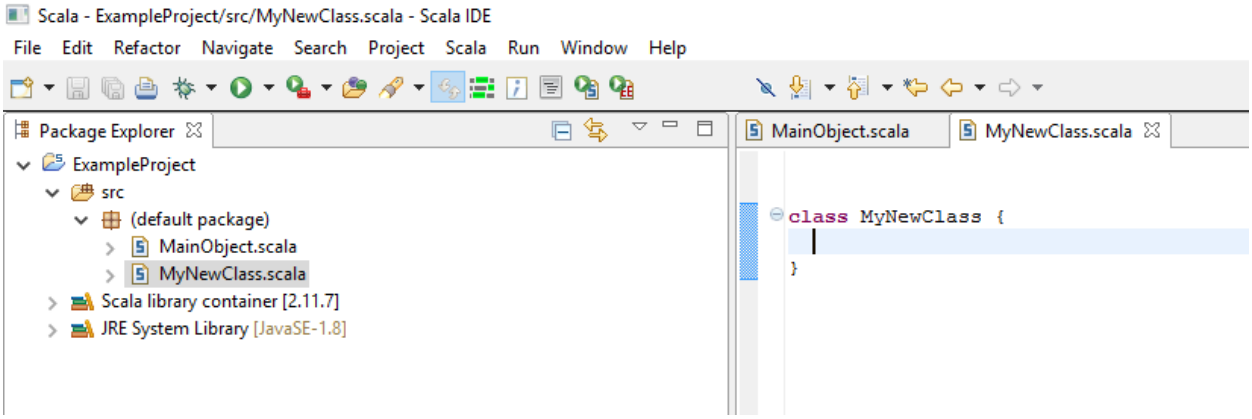

Figure 19: Class File in Package Explorer

Method 2: Adding class to an existing file

We can also code a new class directly into any existing object or class file. Code Listing 66 assumes we did not add the class called **`MyNewClass`** in a separate file and shows us a basic code file for the **`MainObject`** of a new program with the code for the **`MyNewClass`** class defined above the code for the **`MainObject`**.

Code Listing 66: Defining a Class in an Existing File

```
// Definition of a new class
class MyNewClass {
}

// Definition of the MainObject
object MainObject {

  def main(args: Array[String]): Unit = {
    println("All good?")
  }
}
```

Scala is fairly flexible with regards to where we can define a new class. Code Listing 67 shows three examples of new classes defined at different points in our **`MainObject`** file.

Code Listing 67: Adding Classes to an Existing File

```
// Define a new class outside:
class Class1 {
}

// Definition of the MainObject
object MainObject {
  // Define a new class local to MainObject
  class Class2 {
  }
```

```
  def main(args: Array[String]): Unit = {
    // Define a new class local to MainObject.main
    class Class3 {
    }
  }
}
```

Code Listing 67 shows the declaration of three classes, each having a different scope. **`Class1`** is defined outside the body of the **`MainObject`** object, and it has program-wide scope (exactly the same as adding the class to a new file). **`Class2`** is defined inside the body of the **`MainObject`** object. This class is not accessible to outside classes, but it can be used in any methods within the **`MainObject`**. **`Class3`** is defined inside the body of the **`main`** method. This means that the class does not exist outside the **`main`** method.

Class syntax

The syntax for a class begins with the keyword **`class`** and is followed by the name of the class and a code block surrounded by **`{`** and **`}`**. Code Listing 68 shows the basic skeleton of a do-nothing class. This is the basic class that Eclipse will write for us when we add a new class to our project, or the smallest amount of code we are required to write in order to define a new class.

Note: In Java, class files and the classes in them should share the same name. However, this restriction is not part of Scala, and we are free to name our classes anything we like (within reason) and to define multiple classes and objects per file.

Code Listing 68: Skeleton of a Class

```
class ClassName {

  // Body of the class

}
```

A class is simply a blueprint. It defines what types of variables and functions the objects built from it will have. When we create an object from our class blueprint (instantiate the class), the object is called an instance of the class. In order to instantiate a class, we use the **`new`** keyword in a similar way as with Java. Code Listing 69 shows two examples of creating an instance from a class called **`ClassName`** (this Code Listing is not complete and will not compile and run).

Code Listing 69: Creating an Instance from a Class

```
// Two ways to create an instance from a class
// called ClassName:

// Shorthand method:
var classInstance = new ClassName

// Verbose method:

var classInstance: ClassName = new ClassName
```

In Code Listing 69, the first method for creating an instance is to specify either **`var`** or **`val`** (depending on whether you want to change the variable or create a constant object). Next, we use an identifier for the new object, in this case **`classInstance`**, and we set the identifier equal to **`new ClassName`**. This is a shorthand method for creating an instance, and it should look very familiar. This is exactly the same as when we define other basic objects such as **`Int`** and **`Boolean`**.

The second method is slightly more verbose than the first. We can optionally specify the data type for our new object. In the previous example, this is not particularly useful, but Code Listing 70 shows another example of this verbose method, this time using inheritance. Code Listing 70 defines an instance of **`SomeChild`** called **`myInstance`**, but the data type is **`SomeParent`**. We will soon look at inheritance in more detail.

Code Listing 70: Example of Verbose Method with Inheritance

```
// Define a parent class:
class SomeParent {

}

// Define a child class:
class SomeChild extends SomeParent {

}


// Definition of the MainObject
object MainObject {
  def main(args: Array[String]): Unit = {

    // Define a SomeParent object, which is
    // presently an instance of SomeChild
    var myInstance: SomeParent = new SomeChild
  }
}
```

Code Listing 71 shows a basic example of a class complete with a few fields. The listing also shows that we access the fields using the dot syntax.

Code Listing 71: Basic Class with Some Fields

```
// Definition of the Atom class
class Atom {
  // Three fields, or member variables:
  var electronCount: Int = 0
  var name: String = "Unknown"
  var symbol: String = "NoSymbol"
}

object MainObject {
  def main(args: Array[String]): Unit = {

    // Instantiate a new member of Atom class:
    val hydrogen = new Atom

    // Set the fields/member variables:
    hydrogen.electronCount = 1
    hydrogen.name = "Hydrogen"
    hydrogen.symbol = "H"

    // Access the fields/member variables:
    println("Name: " + hydrogen.name + " (" + hydrogen.symbol + ")")
    println("Electrons: " + hydrogen.electronCount)
  }
}
```

Note that in Code Listing 71, the "**.**" means "the field belonging to," so that **hydrogen.symbol** means the **symbol** field belonging to the **hydrogen** object. Each instance of a class has its own fields; if we created a second object from the **Atom** class, **iron**, for example, the fields **hydrogen.symbol** and **iron.symbol** would be two distinct fields that would not necessarily have the same values.

In Scala, we have abstract classes, just like in Java. Unlike Java, Scala has abstract variables. If a class variable is not assigned a value in the class definition, the class must be marked as abstract. This is also true of methods. Methods can be abstract (or have no definition) in Scala, and any class with one or more abstract methods is itself abstract. In Code Listing 71, all variables have been given a default value in the **Atom** class. Also note that the variables are public by default, which means they can be accessed inside the main method without marking them as public (whereas in Java, all members of a class are private by default). We will look at abstract class in more detail later, but this is the reason that I have set each of the members of the **Atom** class to default values **0**, **Unknown**, and **NoSymbol**.

As a second example, Code Listing 72 shows a basic **Box** class. The class consists of two fields, **sideLen1** and **sideLen2**, that we will use to define a box of size **sideLen1*sideLen2**. We will expand this class by adding some member methods.

Code Listing 72: Basic Box Class

```
class Box {
  var sideLen1: Int = 0
  var sideLen2: Int = 0
}
```

Val vs. var in object-oriented programming

It is worth pointing out a particularly detailed nuance of the **val** vs. **var** mechanism. If we have a **val** that refers to some object, and the class has fields marked as **var**, we can change the object's fields even though the object itself is immutable. The **val** means the assignment of the object itself is immutable—it does not refer to the member fields of the object (which may or may not be **val** themselves). Code Listing 73 shows an example of this behavior.

Code Listing 73: Val vs. Var and Objects

```
class Box {
  var sideLen1: Int = 0
  var sideLen2: Int = 0
}


object MainObject {
  def main(args: Array[String]): Unit = {
    // Define a new val:
    val immutableBox = new Box

    // Define a new var:
    var mutableBox = new Box


    // Set the fields of mutableBox
    mutableBox.sideLen1 = 12
    mutableBox.sideLen2 = 13

    // Set the fields of the immutableBox
    immutableBox.sideLen1 = 23
    immutableBox.sideLen2 = 14

    // Set the mutableBox to point to another Box.
    // This is fine because mutableBox is var:
    mutableBox = immutableBox


    // But the following illegal, we cannot reassign the
    // immutableBox, because it is val!
    immutableBox = mutableBox
   }
}
```

Notice that in Code Listing 73 we can change the fields of the object called **immutableBox** even though the object is **val**. But we cannot reassign the object to another **Box** (this reassignment is illustrated by the final line, which I highlighted in red because it is illegal). It is very important to understand what the **val** and **var** refer to when we use the terms in our projects.

Private modifier

When we declare a field in a class, we can mark it as **private**. This prevents any outside objects from interacting with the field. In Scala, members that have no modifier are assumed to be public, so that external objects can interact with the fields. In object-oriented programming, it is recommended that we hide details of the way our classes work because that gives us the flexibility of changing the way the class works without having to worry about other objects accessing the fields directly.

Code Listing 74 shows the code for our **Box** class, but now the **sideLen1** and **sideLen2** fields have been marked as **private** (highlighted in Yellow). Notice that we are not able to set the **sideLen1** field from the **main** method because the **MainObject** object is not part of the **Box** class, and the fields are **private**. Therefore, the final line of the **main** method is illegal, and I have highlighted it in Red.

Code Listing 74: Private Fields

```
class Box {
  private var sideLen1: Int = 0
  private var sideLen2: Int = 0
}



object MainObject {
  def main(args: Array[String]): Unit = {

    // Create an instance from the box class:
    var myBox = new Box

    // It is no longer legal to access the sideLen1
    // or sideLen2 fields outside the Box class.
    // The following line is illegal!
    myBox.sideLen1 = 100
   }
}
```

Member methods

A member method is a function that instances of a class are able to perform. When we define a member method for a class, we access all of the class's private fields. In order to add a member method to a class, we use **def** to define a method inside the body of the class. We have seen this many times already, particularly when the **main** method is a method that we have defined for our **MainObject** objects.

Code Listing 75: Basic Member Methods

```
class Box {
  private var sideLen1: Int = 0
  private var sideLen2: Int = 0

  // Member method called area:
  def area(): Int = {
    return sideLen1 * sideLen2
  }

  // Member method called perimeter:
  def perimeter: Int = { // No params, brackets are optional
    2*(sideLen1 + sideLen2) // Implicit return
  }
}
```

Note: When we use parameters in methods, they are `val` from the point of view of the method, so they cannot be changed. This means that even when a `var` is passed to a method from within the body of the method, the value is immutable. Put another way, Scala does not support C# style out or ref parameters.

Note: The return statement in functions is not needed. Functions return or evaluate to the last value computed. In Scala, it is typical that we try to write functions so that only a single line returns the result. This means that we tend to ensure a Scala function evaluates to a single return statement, and the return keyword is often not used.

Code Listing 75 shows two example methods for our **`Box`** class. The method **`area`** returns **`Int`** and takes no parameters. Likewise, the method **`perimeter`** takes no parameters and returns **`Int`**. When a function takes no parameters, we can leave out the parameter parentheses, as in the **`perimeter`** method.

We can also leave out the code block if a function is only a single statement. This means the **`area`** function of the **`Box`** class could have been written as the following single line of code (the perimeter method could also be a single line): **`def area(): Int = sideLen1 * sideLen2`**.

If a method returns **`Unit`**, i.e. no return value, we can use the brackets in a similar way as with Java by leaving out the return type of **`Unit`** all together. Code Listing 76 shows a new method we can add to our **`Box`** class. This method prints out the **`sideLen1`** and **`sideLen2`** fields, but it does not return anything, and I have left out the return type of **`Unit`**.

Code Listing 76: Unit Is Optional

```
def printMe() {
  println("Box Sides: " + sideLen1 + " " +
      sideLen2)
}
```

Constructors

A constructor is a special member method that we call when we use the **new** operator. In Scala, the constructor for a class is defined by specifying a parameter list in the class's declaration. Code Listing 77 shows an example of our **Box** class, complete with a constructor that takes two integers, **side1** and **side2**. We set the member fields **sideLen1** and **sideLen2** to the parameters passed. Then, in the main method, when we create an instance of our class using the **new** operator, we can pass the lengths as parameters.

Code Listing 77: Constructors

```
// Box class with constructor:
class Box(side1: Int, side2: Int) {

  // Set the member fields to the values
  // passed as paramaters:
  private var sideLen1: Int = side1
  private var sideLen2: Int = side2

}

object MainObject {
   def main(args: Array[String]): Unit = {

     // Call the Box constructor and pass parameters:
     var myBox: Box = new Box(10, 12)

   }
}
```

In order to define multiple constructors, we use the **this** keyword to overload the constructor. This is useful for defining several constructors that take different parameter lists. Code Listing 78 shows an example of our **Box** class with three different constructors.

Code Listing 78: Defining Multiple Constructors

```
// Class with 3 constructors:
class Box(side1: Int, side2: Int) {

  private var sideLen1: Int = 0
  private var sideLen2: Int = 0

  def this() {
    this(-1, -1)  // Call main constructor with -1
  }

  // This constructor takes one parameter,
  // it sets both sideLen fields to the
  // same value:
  def this(side: Int) {
    this    // Call the constructor which takes no arguments
```

```
    // After we have called any fully defined constructor
    // inside the body of a new constructor, we are free to
    // reassign the values of the fields:
    sideLen1 = side
    sideLen2 = side
  }
}


object MainObject {
   def main(args: Array[String]): Unit = {

     // Create a box by calling the main constructor:
     var box1 = new Box(10, 10)

     // Create some boxes by calling the parameter-less constructor
     var box2 = new Box
     var box3 = new Box()

     // Create a box by calling the constructor which takes one
     // parameter:
     var box4 = new Box(100)
   }
}
```

Notice that in Code Listing 78 the first thing inside the additional constructors is a call to some other, fully defined constructor. The main constructor for our class is defined with the class declaration as requiring two **`Int`** parameters. This means that when we define new constructors, they must provide a call to this main constructor in some way. We can either call the main constructor directly, e.g., **`this(-1, -1)`**, or we can call some other constructor that in turn calls the main constructor, e.g., **`this`** in the third constructor. Note that the third constructor calls the parameterless constructor, which in turn calls the main constructor.

Note: Function overloading is a technique in which we create multiple functions with the same name. We can have as many functions with the same name as we need, but the functions must have unique parameter lists.

Note: We can have two or more fields with the same name in different scopes. This is the same as in Java, but in Scala we can also define two or more variables with the same name in nested scopes. The inner variable is said to shadow the outer one because the variable defined in the outer scope is not available until the inner one goes out of scope.

Note: Scala does not have static member variables. We can, however, create singleton objects—these are objects built from classes of which there is only instance. Singletons are simply Scala objects. We can add as many as we like in exactly the same way that we have been adding our `MainObject` object to our programs. Singletons are

similar to classes in every way—except that we do not instantiate them because they already represent the only instance of the singleton.

Note: In addition to allowing singleton objects, in Scala we can also create companion objects. A companion object is an object that has the same name as a class and that is defined in the same file as that class. Companion objects can be used in a similar way to static member methods and fields in Java.

Inheritance

In terms of inheritance, Scala offers mechanisms similar to Java's. We can create a parent class with functions and fields, then inherit from this parent to a more specific child class. Code Listing 79 shows an example of inheritance. In order to inherit from a parent class, we use the **`extends`** keyword.

Code Listing 79: Inheritance

```
// Main parent class:
class GameObject(objName: String, xPos: Int, yPos: Int) {

  val name = objName
  var x = xPos
  var y = yPos

  def print {
     println("Name: " + name + " Pos: " + x + "x" + y)
  }
}

// PointObject class is a child class inheriting from
// GameObject, but it adds a score, which is the amount
// of points the player receives for collecting the object.
class PointsObject(objName: String, xPos: Int, yPos: Int, scoreValue: Int)
  extends GameObject(objName, xPos, yPos) {

  // Define an extra field to record the score
  // this object is worth:
  var score: Int = scoreValue

}

// Another example class, the MoveableObject also inherits from the
// GameObject parent, but it defines several methods for moving
// around.
class MoveableObject(objName: String, xPos: Int, yPos: Int)
  extends GameObject(objName, xPos, yPos) {

  def moveUp = y = y - 1
  def moveDown = y = y + 1
  def moveLeft = x = x - 1
```

```
  def moveRight = x = x + 1
}

// We can also inherit from other child classes:
class Player extends MoveableObject("Player", 100, 100) {

}

object MainObject {

  def main(args: Array[String]): Unit = {

    // Create a GameObject:
    val gameObject = new GameObject("Some generic object", 54, 123)

    // Create a Points object:
    val pointsObject = new PointsObject("Coin", 65, 18, 500)

    // Create a Player object:
    val player: Player = new Player


    // All objects inherit from the GameObject class, so
    // we can call any methods from that class or access any
    // public member fields:
    player.print
    pointsObject.x = 90


    // In addition, the pointsObject has a score field, and
    // the player has several extra methods defined for moving
    // which it inherited from the MoveableObject parent:
    pointsObject.score = 1000

    player.moveUp
    player.moveLeft
    player.moveLeft

    player.print
  }
}
```

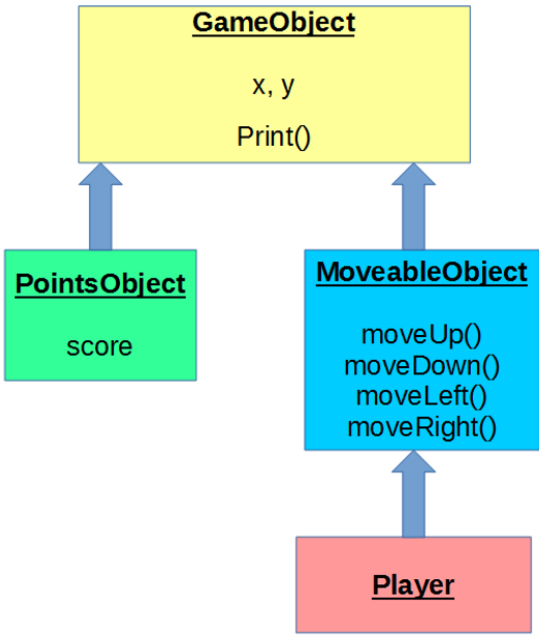

Figure 20: Inheritance Hierarchy from Code Listing 79

Figure 20 is an illustration of the hierarchy defined by Code Listing 79. The main parent is the **`GameObject`** class. Both the **`PointsObject`** and the **`MoveableObject`** classes inherit from this parent. This means they have access to the **`x`** and **`y`** integers from the parent and also to the **`print`** method. The **`Player`** class inherits from the **`MoveableObject`** class, therefore it inherits the **`x`** and **`y`** from the **`MoveableObject`**'s parent along with the additional methods defined for the **`MoveableObject`** class. In this example, the **`Player`** class does not specify any additional fields or methods, but it could.

Notice that in Code Listing 79, when we specify that our new classes extend an existing class, we must call the parent class's constructor **`class Player extends MoveableObject("Player", 100, 100)`**. This means that the **`Player`** class has access to all public members from the parent class and that we should call the parent's constructor with the values **`"Player", 100, 100`** for the parameters.

We can access the parent's methods and fields with the **`super`** keyword in the same way that we do in Java. So, from the **`Player`** class's body, we can access the **`moveUp`** method by calling **`super.moveUp`**.

Abstract classes

I will briefly explain what an abstract class is and how they are defined in Scala, but if you are not already familiar with other object-oriented languages (C++, Java, C#, etc.), I strongly recommend that you become familiar with at least one of them. A lot of technique is involved with object-oriented programming, and this e-book must necessarily concentrate on only Scala and how it differs from some of the other languages.

An abstract class is a class that cannot be instantiated. It can be used as a parent class, and child classes can define meanings for the abstract parts of the parent class. For example, we can create a generic **Shape** class with **computePerimiter** and **computeArea** methods, but the generic parent class itself does not define these methods. We can then inherit from the parent class in a child class such as **Circle** and **Square**, in which we define the body of the parent class's functions.

Code Listing 80: Abstract Class

```
abstract class Shape {

  // Define an abstract field
  type id

  // Define some abstract methods
  def computeArea: Float
  def computePerimeter: Float
}

// Define a Child Class
class Circle(radius: Float) extends Shape {
  var id: Int = 0

  def computeArea: Float = {
    return 3.14159265359f * radius * radius
  }

  def computePerimeter: Float = {
    return 2 * 3.14159265359f * radius
  }
}

object MainObject {

  def main(args: Array[String]): Unit = {

    var circle: Circle = new Circle(6)

    println("Area of Circle: " + circle.computeArea)
  }
}
```

Code Listing 80 shows an abstract parent class called **Shape**. The class contains an undefined field called **id** using the **type** keyword and two abstract methods—**computeArea** and **computePerimiter**. Notice that the class is marked as **abstract**. When we extend from this parent, we must define all of these abstract elements in the child class or else the child class must itself be marked abstract. The **Circle** class inherits from the **Shape** class and provides a definition for each of the parent's abstract elements. This means the **Circle** is not **abstract**, and we can create an instance from it as illustrated in the **main** method of Code Listing 80.

Note: Scala also offers a similar mechanism to Java's interfaces called traits. The interested reader should look up traits in the Scala documentation. Find more information at http://docs.scala-lang.org/tutorials/tour/traits.

Chapter 8 Pattern Matching

Pattern matching is similar to Java's switch/case mechanism. But, as we will see in Scala, pattern matching is more interesting and flexible than switch/case. Code Listing 81 shows a basic example of pattern matching.

Code Listing 81: Simple Matching Example 1

```
object MainObject {

  // This function is an example of pattern matching:
  def matchFruit(index: Int): String = index match {
    case 1 => "Apple"
    case 2 => "Banana"
    case 3 => "Kumquat"
    case _ => "Unknown"
  }

  def main(args: Array[String]): Unit = {

    // 2 and 3 match banana and cumquat
    println("2's Case: " + matchFruit(2))
    println("3's Case: " + matchFruit(3))

    // Anything not mapped in the match/case matches _
    println("100's Case: " + matchFruit(100))
  }
}
```

In Code Listing 81, we use match/case to perform a task much like Java's switch/case mechanism. The function **`matchFruit`** takes an integer parameter called **`index`**, and we match this parameter to various fruits. The first case to correctly match the variable will provide the value to which the variable is mapped. When we call the function with **`matchFruit(2)`**, it will return "**`Banana`**". Likewise, **`matchFruit(3)`** returns the string "**`Kumquat`**".

If we pass a value that does not match any previous case, the underscore case "**`_`**" will execute, and the program will return "**`Unknown`**". The underscore character stands for a wild card, just as it does when we import items using the _. We see the output of this program in Code Listing 82.

Code Listing 82: Output from Code Listing 81

```
2's Case: Banana

3's Case: Cumquat

100's Case: Unknown
```

We can also use match/case without defining a separate function. In Code Listing 81, we defined a separate function called **matchFruit**, but Code Listing 83 shows how to use a match/case to set a variable without calling a distinct function.

Code Listing 83: Simple Matching Example 2

```
object MainObject {
  def main(args: Array[String]): Unit = {

    // Define some variable
    var fruitIndex = 2

    // Perform the matching
    var output = fruitIndex match {
      case 1 => "Apple"
      case 2 => "Banana"
      case 3 => "Cumquat"
      case _ => "Unknown"
    }

    // Output the result:
    println(fruitIndex + "'s Case: " + output)
  }
}
```

Using OR with pattern matching

We can use the OR operator | and combine several conditions into each case. The example in Code Listing 84 takes an input **Int** from **1** to **13** and returns the card classification Ace, King, Small, Medium, etc. Notice the use of | to combine several conditions.

Code Listing 84: Combining Conditions with |

```
object MainObject {
  def main(args: Array[String]): Unit = {

    def classifyPip(x: Int): String = x match {
      case 1 => "Ace"
      case 2|3|4 => "Small"
      case 5|6|7 => "Medium"
      case 8|9|10 => "Large"
      case 11 => "Jack"
      case 12 => "Queen"
      case 13 => "King"
    }

    println("Pip 5 returns: " + classifyPip(5))
    println("Pip 11 returns: " + classifyPip(11))
    println("Pip 1 returns: " + classifyPip(1))
```

```
  }
}
```

Variable scoping

The variables we use in the cases are not the same as any outside variables, even when they have the same names. For instance, Code Listing 85 shows a rather strange output. Study the listing for a moment and try to decide what it will output.

Code Listing 85: Variables in Case vs. Outside

```
object MainObject {
  def main(args: Array[String]): Unit = {

      // Define some variables
      val My_Amazing_Variable = "123"
      val someOtherVar = "456"

      // Perform matching:
      "123" match {
        case someOtherVar =>
          println("someOtherVar")
        case My_Amazing_Variable =>
          println("My_Amazing_Variable")
    }
  }
}
```

Looking at Code Listing 85, we might assume the string "**123**" matches the variable called "**My_Amazing_Variable**" because that variable is set to "**123**". Therefore, we might expect the program in this example to output "**My_Amazing_Variable**". But this is not what happens. The program will output "**someOtherVar**", and it is important that we know why.

Scala will take the string "**123**" to match against its cases. The first case is "**someOtherVar**". There is a local variable called **someOtherVar**, but the **someOtherVar** in the cases is actually shadowing it! The **someOtherVar** in the cases is not related to the local variable with the same name. "**123**" definitely matches some random variable name, which means the program will print "**someOtherVar**" to the screen. It is not testing the value of the local variable **someOtherVar**, but rather it is assigning "**123**" to a new variable with the same name. This output would be exactly the same as if we named the first case **anyRandomVariable**, and the fact that the variable outside the cases shares the same name as the case's variable is irrelevant.

We can test the actual values of local variables in our cases. If we want to use the actual values from the variables defined outside the cases, we must delimit the variable names with back quotes—see Code Listing 86.

Code Listing 86: Delimiting Variable Names with Back Quotes

```
object MainObject {
  def main(args: Array[String]): Unit = {

    // Define some variables:
    val My_Amazing_Variable = "123"
    val someOtherVar = "456"

    "123" match {

      // Use back quotes to test the value of the local
      // variables:
      case `someOtherVar` => println("someOtherVar")
      case `My_Amazing_Variable` => println("My_Amazing_Variable")
    }
  }
}
```

Code Listing 86 will check the values of the local variables called "**someOtherVar**" and "**My_Amazing_Variable**", and it will print "**My_Amazing_Variable**" to the screen because the string "**123**" matches the value of this variable as defined outside the scope of the cases.

Cases and classes

Match/case in Scala is much more powerful than Java's switch/case. We can match objects as well as simple data types. Code Listing 87 shows an example of matching objects. These examples are all about musical key signatures. The exact meaning of the key names and sharps or flats is irrelevant—the listings are simply illustrations of how matching works.

Code Listing 87: Matching Objects of a Case Class

```
object MainObject {

  // Define a class marked with 'case' modifier
  case class KeySignature(name: String, sharpsFlats: Int)

  def main(args: Array[String]): Unit = {
    // Define some KeySignature variables:
    var key1 = new KeySignature("C", 0)
    var key2 = new KeySignature("Bb", -2)
    var key3 = new KeySignature("c", -3)

    // Perform a loop to match our keys:
    for(key <- List(key1, key2, key3)) {

      // Perform the match for each key:
      val fullKeyName = key match {
        case KeySignature("C", 0) => "C Major"
```

```
        case KeySignature("Bb", -2) => "B Flat Major"
        case KeySignature("c", -3) => "C Minor"
        }

      println("Key: " + key + " -> " + fullKeyName)
    }
  }
}
```

In Code Listing 87, we define a class called **KeySignature**. Note that the class is marked with the modifier **case**. This is important if we wish to use the class in a match/case. When we mark a class with the **case** modifier, Scala writes additional methods that enable it to perform pattern matching.

Case classes have an **equals** method, **toString** method, a **hashcode** method, and several other methods written for them. Case classes can be instantiated without the "new" operator because they implement the **apply** method, and all parameters to the constructor of a case class are public and **val**. This is important because it allows matching. Without the **case** modifier, we would need to write our own code to mimic the code in Code Listing 87.

Wild card

Code Listing 87 shows very basic matching. We can also use the wild card symbol for one or all of the parameters for the cases. This is where the term “pattern matching” really becomes applicable. We are not necessarily matching objects against their exact copies, as we do with a Java switch/case, but instead we are matching objects against patterns.

Code Listing 88 shows an example of using the _ wild card in the determination of key signatures.

Code Listing 88: Using _ as a Wild Card in Cases

```
object MainObject {
  case class KeySignature(name: String, sharpsFlats: Int)

  def main(args: Array[String]): Unit = {
    // Define some keys
    var key1 = new KeySignature("C", 0)
    var key2 = new KeySignature("Bb", -2)
    var key3 = new KeySignature("c", -3)

    // Loop through the keys, this loop has an additional
    // couple of keys, "D" and "QWERTY" at the end:
    for(key <- List(key1, key2, key3,
      KeySignature("D", 123),// D does not actually have 123 sharps!
      KeySignature("QWERTY", 5)// 5 sharps is not called QWERTY!
      )) {
      // Perform the matching:
```

```
      val fullKeyName = key match {
        case KeySignature("C", 0) => "C Major"
        case KeySignature("Bb", -2) => "B Flat Major"
        case KeySignature("c", -3) => "C Minor"

        // Using wild cards for parameters:
        case KeySignature(_, 5) => "B Major" // B Major has 5 sharps
        case KeySignature("D", _) => "D Major" // D Major
      }


      println("Key: " + key + " -> " + fullKeyName)
    }
  }
}
```

Code Listing 88 shows that we can match objects even when we do not necessarily match all parameters. The wild card symbol is used in Code Listing 88 to return "**B Major**" when the key has 5 sharps, and the key name is irrelevant because of the _. Likewise, we can match the key "**D Major**" by stating that if the key name is "**D**", then the number of sharps is irrelevant. This is not actually how musical keys work (D Major has two sharps in reality), but this works as an illustration.

Using Any as a data type

We can often use **Any** as a data type to mean multiple types are returned. Notice how the keyword is used in Code Listing 89 to mean “any data type.”

Code Listing 89: Using Any as a Data Type with Matching

```
object MainObject {
  def main(args: Array[String]): Unit = {

    // This function takes a single parameter
    // of any data type:
    def toColorString(q: Any): Any = q match {

      case 1 => "Red"
      case "1" => "Red"
      case "one" => "Red"
      case 2 => "Green"
      case "2" => "Green"
      case "two" => "Green"
      case 3 => "Blue"
      case "3" => "Blue"
      case "three" => "Blue"

      case _ => -1
    }
```

```
    // Test the matching with some calls to toColorString:
    println("Color matched for \"one\": " + toColorString("one"))
    println("Color matched for \"2\": " + toColorString("2"))
    println("Color matched for 3: " + toColorString(3))
    println("Color matched for \"Hello\": " + toColorString("Hello"))
  }
}
```

In Code Listing 89, we specify the data type of the function **toColorString** as **Any**. This means any data type can be passed as the parameter **q**. Then we specify that the function returns **Any** as a data type. This means we can return multiple different data types from this function.

When we match the **q** variable, we provide cases for **Int** and **String**. We also provide a final case that has a pattern of **_**, the wild card. If none of the previous cases matches, we return **-1** as an **Int**. This is a function that takes multiple parameter types, tests them with a series of cases, and returns either a **String** or an **Int**, depending on whether or not the **q** parameter was matched. If you are familiar with Java programming, this function will look extremely odd.

The next example program uses **Any** as a data type again. This time, we return a **String** version of the input if it is an **Int**, and an **Int** version if it is a **String**. Without some context, this is a pointless activity, but it does illustrate how we can easily test and change data types without using the complex syntax that Java requires in order to do the same thing.

Code Listing 90: Flipping Data Types

```
object MainObject {
  def main(args: Array[String]): Unit = {

    // Define the function to flip data types:
    def flipStringAndInt(x: Any): Any = x match {
      case y: Int => y.toString
      case y: String => y.toInt
      case _ => "Unknown data type!"
    }

    // Make some test cases:
    val myInt = flipStringAndInt("190")
    val myString = flipStringAndInt(190)
    val unknown = flipStringAndInt(190.0)

    // Output results:
    println("myInt: " + myInt)
    println("myString: " + myString)
    println("unknown: " + unknown)
  }
}
```

In order to match the type of the argument in a case, we specify another variable—**y** in the example. We say that **y: Int =>**, which means the data type of **y** is **Int**, then we supply the return value. So, when the data type of **y** is an **Int**, the pattern-matching mechanism maps it to a **String**, and vice versa—**String** is mapped to **Int**.

We should note that in Code Listing 90 the function **flipStringAndInt** returns a **String** for any input that is an **Int**, and vice versa. When we pass a **Double** as the input, the function returns the string **Unknown data type!**.

Chapter 9 Closures

A closure is a function that computes with variables defined outside the body of the function. Code Listing 91 shows a simple example of a closure. Closures are sometimes called Lambda functions, and they are similar to Java's anonymous functions. Closures are one of the many mechanisms offered by Scala from the functional programming paradigm (as opposed to the object-oriented programming paradigm).

Code Listing 91: Simple Closure

```
object MainObject {
  def main(args: Array[String]) = {

    // Define a variable:
    var divisor = 9

    // Define a closure which uses the divisor
    // variable:
    var divideClosure = (i: Int) => i / divisor

    // Execute the closure using 90 as the
    // Int i:
    println("90/9=" + divideClosure(90))
  }
}
```

In Code Listing 91, we define a closure called **divideClosure**. First, we specify an identifier for the closure, **divideClosure**, then we use the equals operator to set it to a parameter list **= (i: Int)**. We then use the **=>** operator (sometimes called rocket) and specify the body of the closure. Notice that the closure uses the variable called **divisor**, which is defined outside the body of the closure. In this particular instance, the variable **divisor** is still in scope, but as we will see, this does not need to be the case.

Also note that the use of variable **divisor** in the body of the closure does not shadow the local variable **divisor** as we might expect, especially considering some of the previous examples we have examined. The **divisor** variable in the closure is the local variable **divisor**.

We should note that closures do not need to use external variables. We can define a closure that uses only the parameters defined in its own parameter list. Also, the closure evaluates the values of the variables, so that when we update the values of the variables defined outside the body of the closure, the return value of the closure will be updated, too.

Code Listing 91 provided a completely redundant example of a closure, but that is actually an interesting mechanism. Another interesting aspect of closures is that we can pass them as parameters to a method. Code Listing 92 shows an example of this. It might not seem strange yet, but it will when we look at its implications.

Code Listing 92: Passing a Closure as a Parameter

```
object MainObject {

  def executeClosure(closure: (Int) => Int, parameter: Int) {
    println("The closure said: " + closure(parameter))
  }

  def main(args: Array[String]) = {
    var divisor = 9
    var divideClosure = (i: Int) => i / divisor

    executeClosure(divideClosure, 125)
  }
}
```

In Code Listing 92, we define a function called **executeClosure**. The function takes two parameters—one is a function called closure and the other is a parameter. The **executeClosure** function executes the function and prints the result to the screen. The function is a roundabout way of dividing 125 by nine, and it prints 13 to the screen, which is perhaps not very interesting (this is just basic integer arithmetic, 125/9=13.888, and the 0.8888 is truncated as per the normal rules of integer arithmetic). However, let's have a quick look at another example. This time, let's illustrate something slightly strange about the way closures work.

Code Listing 93: Altering a Closure's Variable

```
object MainObject {

  def executeClosure(closure: (Int) => Int, parameter: Int) {
    println("The closure said: " + closure(parameter))
  }

  def main(args: Array[String]) = {
    var divisor = 9
    var divideClosure = (i: Int) => i / divisor

    divisor = 45
    executeClosure(divideClosure, 125)
  }
}
```

In the main method of Code Listing 93, we define the same closure as before. This time, however, I have added a line and reassigned the **divisor** variable, setting it to **45**. When we call the function **executeClosure** and pass the parameter **125**, the closure will execute **125/45** even though the **divisor** variable is out of scope at the point of execution and it has been changed since the closure was defined. Code Listing 93 correctly computes the result that 125/45 is 2.

A closure, therefore, is a function we can pass around and that is able to refer to variables that are not in scope.

Shorthand syntax

There is a shorthand syntax for simple closures. We can use the _ (the underscore wild card symbol) to mean a single parameter, if there is one. So if the closure takes only a single parameter, we can use the _ instead of a formal parameter list. See Code Listing 94 for an example of this.

Code Listing 94: Shorthand for Closure

```
object MainObject {

def main(args: Array[String]) = {

    // Define divisor variable.
    var divisor = 9

    // Define a closure using _ syntax:
    var divideClosure = (_:Int) / divisor

    // Again, this closure will divide 125 by 9
    // and return 13:
    println("125/9=" + divideClosure(125))
  }
}
```

Notice that in Code Listing 94 we need to specify the data type of the _ symbol with **(_: Int)**. If the data type is specified in the closure already, we can use the underscore by itself.

Code Listing 95 shows a slightly more complicated example of a closure. This particular use of a closure is commonly used for performing operations on lists and arrays.

Code Listing 95: Passing Functionality as a Parameter

```
object MainObject {
  def main(args: Array[String]) = {

    // Define a functions which takes two ints, x and y
    // and a function to perform between them called func:
    def performOperation
      (x: Int, y: Int, func: (Int, Int)=>Int):
        Int = func(x, y)

    // Call the perform operation function with 78 and 26
    // as the Int parameters, and with the closure (a, b)=>
    // a-b as the func parameter:
    println("78-26=" + performOperation(78, 26, (a, b)=>a-b))


    // Call the perform operation function with 6 and 5
    // as the Int parameters, and with _+_ short hand
    // closure as the func parameter:
```

```
    println("6+5=" + performOperation(6, 5, _+_))
  }
}
```

In Code Listing 95, we define a function called **performOperation**. The function takes three parameters—two **Int** and a function. The function parameter is called **func**. It takes two inputs of its own and returns an **Int** (this is all specified by the **(Int, Int)=>Int**). The **performOperation** function performs whichever operation we pass as a final argument between the two **Int** parameters and returns the result.

Tip: Notice the use of the wild card symbol in the second call to the closure in Code Listing 95. When we use multiple wild cards, such as _+_, the first is assigned to the first parameter and the second to the second parameter, etc. The _+_ is shorthand for a+b since a is the first parameter and is substituted for the first occurrence of _. And b is the second parameter—it is therefore substituted with the second instance of _.

The most important aspect of Code Listing 95 is how we call the function. Notice that with the first call to **performOperation**, we pass **78**, **26** as the integer parameters, then we specify the functionality of the func closure using **(a, b)=>a-b**. This means we want the second parameter to be subtracted from the first, so that the first **println** will output **78-26=52**. The second call uses the wild card symbol and the shorthand syntax for the functionality.

Chapter 10 Conclusion

This has been a short introduction to some of the fascinating mechanisms and features of the Scala language. Scala is a flexible and powerful general-purpose language, it is built upon the Java Runtime Environment, and it can be easily incorporated into existing Java applications. The language offers a rich set of mechanisms that address many of the shortcomings of the Java language, and it is an interesting blend of functional and object-oriented programming paradigms.

I hope you have enjoyed this e-book. I have certainly enjoyed writing it. Many other interesting topics remain, such as the **`yield`** keyword and currying. And there are many Scala-related resources available (both for free and in book form). Scala is one of the most fascinating of the modern languages, and it is being quickly adopted by programmers. If you are interested in learning more about Scala, I recommend the following sources:

Scala Documentation: http://docs.scala-lang.org/ and http://docs.scala-lang.org/tutorials/.

Programming in Scala by Martin Odersky, Lex Spoon, and Bill Venners. Available as a free PDF e-book.

Scala by Example by Martin Odersky. Available as a free PDF e-book.

Tutorialspoint Scala Tutorials.

Programming Scala by Dean Wampler and Alex Payne. Published by O'Reilly. Available from Amazon.