Chapter 2

First Steps in Scala

It's time to write some Scala code. Before we start on the in-depth Scala tutorial, we put in two chapters that will give you the big picture of Scala, and most importantly, get you writing code. We encourage you to actually try out all the code examples presented in this chapter and the next as you go. The best way to start learning Scala is to program in it.

To run the examples in this chapter, you should have a standard Scala installation. To get one, go to http://www.scala-lang.org/downloads and follow the directions for your platform.You can also use a Scala plug-in for Eclipse, IntelliJ, or NetBeans. For the steps in this chapter, we'll assume you're using the Scala distribution from scala-lang.org.[1]

If you are a veteran programmer new to Scala, the next two chapters should give you enough understanding to enable you to start writing useful programs in Scala. If you are less experienced, some of the material may seem a bit mysterious to you. But don't worry. To get you up to speed quickly, we had to leave out some details. Everything will be explained in a less "fire hose" fashion in later chapters. In addition, we inserted quite a few footnotes in these next two chapters to point you to later sections of the book where you'll find more detailed explanations.

STEP 1. LEARN TO USE THE SCALA INTERPRETER

The easiest way to get started with Scala is by using the Scala interpreter, an interactive "shell" for writing Scala expressions and programs. The interpreter, which is called scala, will evaluate expressions you type and print the resulting value. You use it by typing scala at a command prompt:[2]

```
$ scala
Welcome to Scala version 2.11.7
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

After you type an expression, such as 1 + 2, and hit enter:

```
scala> 1 + 2
```

The interpreter will print:

```
res0: Int = 3
```

This line includes:

- an automatically generated or user-defined name to refer to the computed value (res0, which means result 0),
- a colon (:), followed by the type of the expression (Int),

- an equals sign (=),
- the value resulting from evaluating the expression (3).

The type Int names the class Int in the package scala. Packages in Scala are similar to packages in Java: They partition the global namespace and provide a mechanism for information hiding.[3] Values of class Int correspond to Java's int values. More generally, all of Java's primitive types have corresponding classes in the scala package. For example,scala.Boolean corresponds to Java's boolean. scala.Float corresponds to Java's float. And when you compile your Scala code to Java bytecodes, the Scala compiler will use Java's primitive types where possible to give you the performance benefits of the primitive types.

The resX identifier may be used in later lines. For instance, since res0 was set to 3 previously,res0 * 3 will be 9:

```
scala> res0 * 3
res1: Int = 9
```

To print the necessary, but not sufficient, Hello, world! greeting, type:

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

The println function prints the passed string to the standard output, similar toSystem.out.println in Java.

STEP 2. DEFINE SOME VARIABLES

Scala has two kinds of variables, vals and vars. A val is similar to a final variable in Java. Once initialized, a val can never be reassigned. A var, by contrast, is similar to a non-final variable in Java. A var can be reassigned throughout its lifetime. Here's a val definition:

```
scala> val msg = "Hello, world!"
msg: String = Hello, world!
```

This statement introduces msg as a name for the string "Hello, world!". The type of msg isjava.lang.String, because Scala strings are implemented by Java's String class.

If you're used to declaring variables in Java, you'll notice one striking difference here: neitherjava.lang.String nor String appear anywhere in the val definition. This example illustrates *type inference*, Scala's ability to figure out types you leave off. In this case, because you initializedmsg with a string literal, Scala inferred the type of msg to be String. When the Scala interpreter (or compiler) can infer types, it is often best to let it do so rather than fill the code with unnecessary, explicit type annotations. You can, however, specify a type explicitly if you wish, and sometimes you probably should. An explicit type annotation can both ensure the Scala compiler infers the type you intend, as well as serve as useful documentation for future readers of the code. In contrast to Java, where you specify a variable's type before its name, in Scala you specify a variable's type after its name, separated by a colon. For example:

```
scala> val msg2: java.lang.String = "Hello again, world!"
```

```
msg2: String = Hello again, world!
```

Or, since java.lang types are visible with their simple names[4] in Scala programs, simply:

```
scala> val msg3: String = "Hello yet again, world!"
msg3: String = Hello yet again, world!
```

Going back to the original msg, now that it is defined, you can use it as you'd expect, for example:

```
scala> println(msg)
Hello, world!
```

What you can't do with msg, given that it is a val, not a var, is reassign it.[5] For example, see how the interpreter complains when you attempt the following:

```
scala> msg = "Goodbye cruel world!"
<console>:8: error: reassignment to val
       msg = "Goodbye cruel world!"
           ^
```

If reassignment is what you want, you'll need to use a var, as in:

```
scala> var greeting = "Hello, world!"
greeting: String = Hello, world!
```

Since greeting is a var not a val, you can reassign it later. If you are feeling grouchy later, for example, you could change your greeting to:

```
scala> greeting = "Leave me alone, world!"
greeting: String = Leave me alone, world!
```

To enter something into the interpreter that spans multiple lines, just keep typing after the first line. If the code you typed so far is not complete, the interpreter will respond with a vertical bar on the next line.

```
scala> val multiLine =
     |   "This is the next line."
multiLine: String = This is the next line.
```

If you realize you have typed something wrong, but the interpreter is still waiting for more input, you can escape by pressing enter twice:

```
scala> val oops =
     |
     |
You typed two blank lines.  Starting a new command.

scala>
```

In the rest of the book, we'll leave out the vertical bars to make the code easier to read (and easier to copy and paste from the PDF eBook into the interpreter).

STEP 3. DEFINE SOME FUNCTIONS

Now that you've worked with Scala variables, you'll probably want to write some functions. Here's how you do that in Scala:

```
scala> def max(x: Int, y: Int): Int = {
         if (x > y) x
         else y
       }
max: (x: Int, y: Int)Int
```

Function definitions start with def. The function's name, in this case max, is followed by a comma-separated list of parameters in parentheses. A type annotation must follow every function parameter, preceded by a colon, because the Scala compiler (and interpreter, but from now on we'll just say compiler) does not infer function parameter types. In this example, the function named max takes two parameters, x and y, both of type Int. After the close parenthesis of max's parameter list you'll find another ": Int" type annotation. This one defines the *result type* of the max function itself.[6] Following the function's result type is an equals sign and pair of curly braces that contain the body of the function. In this case, the body contains a single if expression, which selects either x or y, whichever is greater, as the result of the max function. As demonstrated here, Scala's if expression can result in a value, similar to Java's ternary operator. For example, the Scala expression "if (x > y) x else y" behaves similarly to "(x > y) ? x : y" in Java. The equals sign that precedes the body of a function hints that in the functional world view, a function defines an expression that results in a value. The basic structure of a function is illustrated in Figure 2.1.

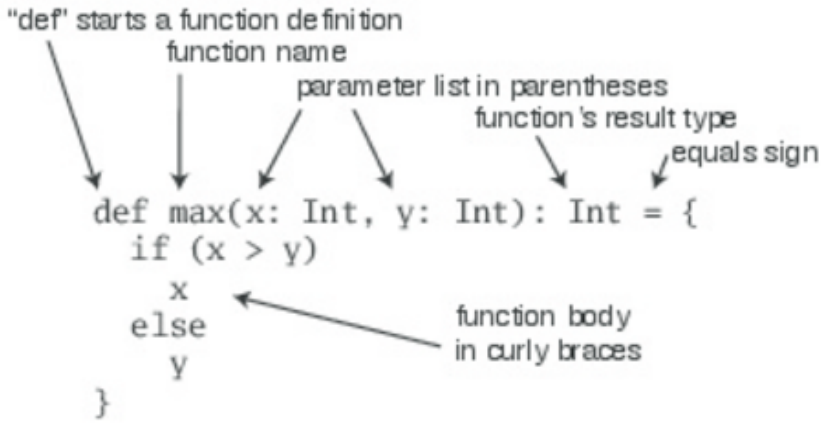

Figure 2.1 - The basic form of a function definition in Scala.

Sometimes the Scala compiler will require you to specify the result type of a function. If the function is *recursive*,[7] for example, you must explicitly specify the function's result type. In the case of max, however, you may leave the result type off and the compiler will infer it.[8]Also, if a function consists of just one statement, you can optionally leave off the curly braces. Thus, you could alternatively write the max function like this:

```
scala> def max(x: Int, y: Int) = if (x > y) x else y
max: (x: Int, y: Int)Int
```

Once you have defined a function, you can call it by name, as in:

```
scala> max(3, 5)
res4: Int = 5
```

Here's the definition of a function that takes no parameters and returns no interesting result:

```
scala> def greet() = println("Hello, world!")
greet: ()Unit
```

When you define the greet() function, the interpreter will respond with greet: ()Unit. "greet" is, of course, the name of the function. The empty parentheses indicate the function takes no parameters. And Unit is greet's result type. A result type of Unit indicates the function returns no interesting value. Scala's Unit type is similar to Java's void type; in fact, every void-returning method in Java is mapped to a Unit-returning method in Scala. Methods with the result type of Unit, therefore, are only executed for their side effects. In the case of greet(), the side effect is a friendly greeting printed to the standard output.

In the next step, you'll place Scala code in a file and run it as a script. If you wish to exit the interpreter, you can do so by entering :quit or :q.

```
scala> :quit
$
```

STEP 4. WRITE SOME SCALA SCRIPTS

Although Scala is designed to help programmers build very large-scale systems, it also scalesdown nicely to scripting. A script is just a sequence of statements in a file that will be executed sequentially. Put this into a file named hello.scala:

```
println("Hello, world, from a script!")
```

then run:[9]

```
$ scala hello.scala
```

And you should get yet another greeting:

```
Hello, world, from a script!
```

Command line arguments to a Scala script are available via a Scala array named args. In Scala, arrays are zero based, and you access an element by specifying an index in parentheses. So the first element in a Scala array named steps is steps(0), not steps[0], as in Java. To try this out, type the following into a new file named helloarg.scala:

```
// Say hello to the first argument
println("Hello, " + args(0) + "!")
```

then run:

```
$ scala helloarg.scala planet
```

In this command, "planet" is passed as a command line argument, which is accessed in the script as args(0). Thus, you should see:

```
Hello, planet!
```

Note that this script included a comment. The Scala compiler will ignore characters between// and the next end of line and any characters between /* and */. This example also showsStrings being concatenated with the + operator. This works as you'd expect. The expression"Hello, " + "world!" will result in the string "Hello, world!".

STEP 5. LOOP WITH WHILE; DECIDE WITH IF

To try out a while, type the following into a file named printargs.scala:

```
var i = 0
while (i < args.length) {
  println(args(i))
  i += 1
}
```

Note

Although the examples in this section help explain while loops, they do not demonstrate the best Scala style. In the next section, you'll see better approaches that avoid iterating through arrays with indexes.

This script starts with a variable definition, var i = 0. Type inference gives i the type scala.Int, because that is the type of its initial value, 0. The while construct on the next line causes the*block* (the code between the curly braces) to be repeatedly executed until the boolean expression i < args.length is false. args.length gives the length of the args array. The block contains two statements, each indented two spaces, the recommended indentation style for Scala. The first statement, println(args(i)), prints out the ith command line argument. The second statement, i += 1, increments i by one. Note that Java's ++i and i++ don't work in Scala. To increment in Scala, you need to say either i = i + 1 or i += 1. Run this script with the following command:

```
$ scala printargs.scala Scala is fun
```

And you should see:

```
Scala
is
fun
```

For even more fun, type the following code into a new file with the name echoargs.scala:

```
var i = 0
while (i < args.length) {
  if (i != 0)
    print(" ")
  print(args(i))
  i += 1
}
```

```
println()
```

In this version, you've replaced the println call with a print call, so that all the arguments will be printed out on the same line. To make this readable, you've inserted a single space before each argument except the first via the if (i != 0) construct. Since i != 0 will be false the first time through the while loop, no space will get printed before the initial argument. Lastly, you've added one more println to the end, to get a line return after printing out all the arguments. Your output will be very pretty indeed. If you run this script with the following command:

```
$ scala echoargs.scala Scala is even more fun
```

You'll get:

```
Scala is even more fun
```

Note that in Scala, as in Java, you must put the boolean expression for a while or an if in parentheses. (In other words, you can't say in Scala things like if i < 10 as you can in a language such as Ruby. You must say if (i < 10) in Scala.) Another similarity to Java is that if a block has only one statement, you can optionally leave off the curly braces, as demonstrated by the if statement in echoargs.scala. And although you haven't seen any of them, Scala does use semicolons to separate statements as in Java, except that in Scala the semicolons are very often optional, giving some welcome relief to your right little finger. If you had been in a more verbose mood, therefore, you could have written the echoargs.scala script as follows:

```
var i = 0;
while (i < args.length) {
  if (i != 0) {
    print(" ");
  }
  print(args(i));
  i += 1;
}
println();
```

STEP 6. ITERATE WITH FOREACH AND FOR

Although you may not have realized it, when you wrote the while loops in the previous step, you were programming in an *imperative* style. In the imperative style, which is the style you normally use with languages like Java, C++, and C, you give one imperative command at a time, iterate with loops, and often mutate state shared between different functions. Scala enables you to program imperatively, but as you get to know Scala better, you'll likely often find yourself programming in a more *functional* style. In fact, one of the main aims of this book is to help you become as comfortable with the functional style as you are with imperative style.

One of the main characteristics of a functional language is that functions are first class constructs, and that's very true in Scala. For example, another (far more concise) way to print each command line argument is:

```
args.foreach(arg => println(arg))
```

In this code, you call the foreach method on args and pass in a function. In this case, you're passing in a *function literal* that takes one parameter named arg. The body of the function isprintln(arg). If you type the above code into a new file named pa.scala and execute with the command:

```
$ scala pa.scala Concise is nice
```

You should see:

```
Concise
is
nice
```

In the previous example, the Scala interpreter infers the type of arg to be String, since String is the element type of the array on which you're calling foreach. If you'd prefer to be more explicit, you can mention the type name. But when you do, you'll need to wrap the argument portion in parentheses (which is the normal form of the syntax anyway):

```
args.foreach((arg: String) => println(arg))
```

Running this script has the same behavior as the previous one.

If you're in the mood for more conciseness instead of more explicitness, you can take advantage of a special shorthand in Scala. If a function literal consists of one statement that takes a single argument, you need not explicitly name and specify the argument.[10] Thus, the following code also works:

```
args.foreach(println)
```

To summarize, the syntax for a function literal is a list of named parameters, in parentheses, a right arrow, and then the body of the function. This syntax is illustrated in Figure 2.2.

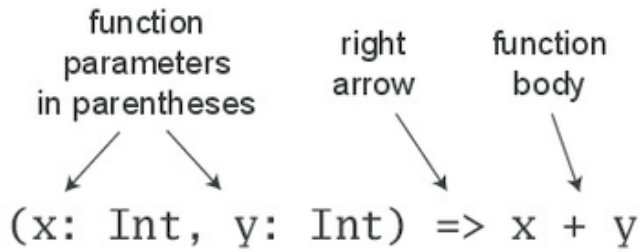

Figure 2.2 - The syntax of a function literal in Scala.

Now, by this point you may be wondering what happened to those trusty for loops you have been accustomed to using in imperative languages, such as Java or C. In an effort to guide you in a functional direction, only a functional relative of the imperative for (called a *forexpression*) is available in Scala. While you won't see their full power and expressiveness until you reach (or peek ahead to) Section 7.3, we'll give you a glimpse here. In a new file named forargs.scala, type the following:

```
for (arg <- args)
```

```
    println(arg)
```

The parentheses after the "for" contain arg <- args.[11] To the right of the <- symbol is the familiar args array. To the left of <- is "arg", the name of a val, not a var. (Because it is always aval, you just write "arg" by itself, not "val arg".) Although arg may seem to be a var, because it will get a new value on each iteration, it really is a val: arg can't be reassigned inside the body of the for expression. Instead, for each element of the args array, a *new* arg val will be created and initialized to the element value, and the body of the for will be executed.

If you run the forargs.scala script with the command:

```
$ scala forargs.scala for arg in args
```

You'll see:

```
for
arg
in
args
```

Scala's for expression can do much more than this, but this example is enough to get you started. We'll show you more about for in Section 7.3 and Chapter 23.

CONCLUSION

In this chapter, you learned some Scala basics and, hopefully, took advantage of the opportunity to write a bit of Scala code. In the next chapter, we'll continue this introductory overview and get into more advanced topics.

Footnotes for Chapter 2:

[1] We tested the examples in this book with Scala version 2.11.7.

[2] If you're using Windows, you'll need to type the scala command into the "Command Prompt" DOS box.

[3] If you're not familiar with Java packages, you can think of them as providing a full name for classes. Because Int is a member of package scala, "Int" is the class's simple name, and "scala.Int" is its full name. The details of packages are explained in Chapter 13.

[4] The simple name of java.lang.String is String.

[5] In the interpreter, however, you can *define* a new val with a name that was already used before. This mechanism is explained in Section 7.7.

[6] In Java, the type of the value returned from a method is its return type. In Scala, that same concept is called *result type*.

[7] A function is recursive if it calls itself.

[8] Nevertheless, it is often a good idea to indicate function result types explicitly, even when the compiler doesn't require it. Such type annotations can make the code easier to read, because the reader need not study the function body to figure out the inferred result type.

[9] You can run scripts without typing "scala" on Unix and Windows using a "pound-bang" syntax, which is shown in Appendix A.

[10] This shorthand, called a *partially applied function*, is described in Section 8.6.

[11] You can say "in" for the <- symbol. You'd read for (arg <- args), therefore, as "for arg in args."