Chapter 3

Next Steps in Scala

This chapter continues the previous chapter's introduction to Scala. In this chapter, we'll introduce some more advanced features. When you complete this chapter, you should have enough knowledge to enable you to start writing useful scripts in Scala. As with the previous chapter, we recommend you try out these examples as you go. The best way to get a feel for Scala is to start writing Scala code.

STEP 7. PARAMETERIZE ARRAYS WITH TYPES

In Scala, you can instantiate objects, or class instances, using new. When you instantiate an object in Scala, you can *parameterize* it with values and types. Parameterization means "configuring" an instance when you create it. You parameterize an instance with values by passing objects to a constructor in parentheses. For example, the following Scala code instantiates a new java.math.BigInteger and parameterizes it with the value "12345":

```
val big = new java.math.BigInteger("12345")
```

You parameterize an instance with types by specifying one or more types in square brackets. An example is shown in Listing 3.1. In this example, greetStrings is a value of type Array[String](an "array of string") that is initialized to length 3 by parameterizing it with the value 3 in the first line of code. If you run the code in Listing 3.1 as a script, you'll see yet anotherHello, world! greeting. Note that when you parameterize an instance with both a type and a value, the type comes first in its square brackets, followed by the value in parentheses.

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

greetStrings(0) = "Hello"
greetStrings(1) = ", "
greetStrings(2) = "world!\n"

for (i <- 0 to 2)
  print(greetStrings(i))
```

Listing 3.1 - Parameterizing an array with a type.

Note

Although the code in Listing 3.1 demonstrates important concepts, it does not show the recommended way to create and initialize an array in Scala. You'll see a better way in Listing 3.2 here.

Had you been in a more explicit mood, you could have specified the type of greetStringsexplicitly like this:

```
val greetStrings: Array[String] = new Array[String](3)
```

Given Scala's type inference, this line of code is semantically equivalent to the actual first line of Listing 3.1. But this form demonstrates that while the type parameterization portion (the type names in square brackets) forms part of the type of the instance, the value parameterization part (the values in parentheses) does not. The type of greetStrings isArray[String], not Array[String](3).

The next three lines of code in Listing 3.1 initialize each element of the greetStrings array:

```
greetStrings(0) = "Hello"
greetStrings(1) = ", "
greetStrings(2) = "world!\n"
```

As mentioned previously, arrays in Scala are accessed by placing the index inside parentheses, not square brackets as in Java. Thus the zeroth element of the array isgreetStrings(0), not greetStrings[0].

These three lines of code illustrate an important concept to understand about Scala concerning the meaning of val. When you define a variable with val, the variable can't be reassigned, but the object to which it refers could potentially still be changed. So in this case, you couldn't reassign greetStrings to a different array; greetStrings will always point to the same Array[String] instance with which it was initialized. But you *can* change the elements of that Array[String] over time, so the array itself is mutable.

The final two lines in Listing 3.1 contain a for expression that prints out each greetStringsarray element in turn:

```
for (i <- 0 to 2)
  print(greetStrings(i))
```

The first line of code in this for expression illustrates another general rule of Scala: if a method takes only one parameter, you can call it without a dot or parentheses. The to in this example is actually a method that takes one Int argument. The code 0 to 2 is transformed into the method call (0).to(2). [1] Note that this syntax only works if you explicitly specify the receiver of the method call. You cannot write "println 10", but you can write "Console println 10".

Scala doesn't technically have operator overloading, because it doesn't actually have operators in the traditional sense. Instead, characters such as +, -, *, and / can be used in method names. Thus, when you typed 1 + 2 into the Scala interpreter in Step 1, you were actually invoking a method named + on the Int object 1, passing in 2 as a parameter. As illustrated in Figure 3.1, you could alternatively have written 1 + 2 using traditional method invocation syntax, (1).+(2).

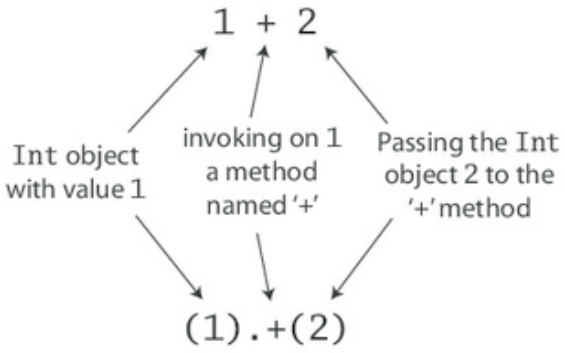

Figure 3.1 - All operations are method calls in Scala.

Another important idea illustrated by this example will give you insight into why arrays are accessed with parentheses in Scala. Scala has fewer special cases than Java. Arrays are simply instances of classes like any other class in Scala. When you apply parentheses surrounding one or more values to a variable, Scala will transform the code into an invocation of a method named apply on that variable. So greetStrings(i) gets transformed into greetStrings.apply(i). Thus accessing an element of an array in Scala is simply a method call like any other. This principle is not restricted to arrays: any application of an object to some arguments in parentheses will be transformed to an apply method call. Of course this will compile only if that type of object actually defines an apply method. So it's not a special case; it's a general rule.

Similarly, when an assignment is made to a variable to which parentheses and one or more arguments have been applied, the compiler will transform that into an invocation of an updatemethod that takes the arguments in parentheses as well as the object to the right of the equals sign. For example:

```
greetStrings(0) = "Hello"
```

will be transformed into:

```
greetStrings.update(0, "Hello")
```

Thus, the following is semantically equivalent to the code in Listing 3.1:

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

greetStrings.update(0, "Hello")
greetStrings.update(1, ", ")
greetStrings.update(2, "world!\n")

for (i <- 0.to(2))
  print(greetStrings.apply(i))
```

Scala achieves a conceptual simplicity by treating everything, from arrays to expressions, as objects with methods. You don't have to remember special cases, such as the differences in Java between primitive and their corresponding wrapper types, or between arrays and regular objects. Moreover, this uniformity does not incur a significant performance cost. The Scala compiler uses Java arrays, primitive types, and native arithmetic where possible in the compiled code.

Although the examples you've seen so far in this step compile and run just fine, Scala provides a more concise way to create and initialize arrays that you would normally use (seeListing 3.2). This code creates a new array of length three, initialized to the passed strings,"zero", "one", and "two". The compiler infers the type of the array to be Array[String], because you passed strings to it.

```
val numNames = Array("zero", "one", "two")
```

Listing 3.2 - Creating and initializing an array.

What you're actually doing in Listing 3.2 is calling a factory method, named apply, which creates and returns the new array. This apply method takes a variable number of arguments[2] and is defined on the Array *companion object*. You'll learn more about companion objects in Section 4.3. If you're a Java programmer, you can think of this as calling a static method named apply on class Array. A more verbose way to call the same applymethod is:

```
val numNames2 = Array.apply("zero", "one", "two")
```

STEP 8. USE LISTS

One of the big ideas of the functional style of programming is that methods should not have side effects. A method's only act should be to compute and return a value. Some benefits gained when you take this approach are that methods become less entangled, and therefore more reliable and reusable. Another benefit (in a statically typed language) is that everything that goes into and out of a method is checked by a type checker, so logic errors are more likely to manifest themselves as type errors. Applying this functional philosophy to the world of objects means making objects immutable.

As you've seen, a Scala array is a mutable sequence of objects that all share the same type. AnArray[String] contains only strings, for example. Although you can't change the length of an array after it is instantiated, you can change its element values. Thus, arrays are mutable objects.

For an immutable sequence of objects that share the same type you can use Scala's List class. As with arrays, a List[String] contains only strings. Scala's List, scala.List, differs from Java'sjava.util.List type in that Scala Lists are always immutable (whereas Java Lists can be mutable). More generally, Scala's List is designed to enable a functional style of programming. Creating a list is easy, and Listing 3.3 shows how:

```
val oneTwoThree = List(1, 2, 3)
```

Listing 3.3 - Creating and initializing a list.

The code in Listing 3.3 establishes a new val named oneTwoThree, initialized with a newList[Int] with the integer elements 1, 2, and 3.[3] Because Lists are immutable, they behave a bit like Java strings: when you call a method on a list that might seem by its name to imply the list will mutate, it instead creates and returns a new list with the new value. For example,List has a method named `:::' for list concatenation. Here's how you use it:

```
val oneTwo = List(1, 2)
val threeFour = List(3, 4)
```

```
val oneTwoThreeFour = oneTwo ::: threeFour
println(oneTwo + " and " + threeFour + " were not mutated.")
println("Thus, " + oneTwoThreeFour + " is a new list.")
```

If you run this script, you'll see:

```
List(1, 2) and List(3, 4) were not mutated.
Thus, List(1, 2, 3, 4) is a new list.
```

Perhaps the most common operator you'll use with lists is `::', which is pronounced "cons."Cons prepends a new element to the beginning of an existing list and returns the resulting list. For example, if you run this script:

```
val twoThree = List(2, 3)
val oneTwoThree = 1 :: twoThree
println(oneTwoThree)
```

You'll see:

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

Note

In the expression "1 :: twoThree", :: is a method of its right operand, the list,twoThree. You might suspect there's something amiss with the associativityof the :: method, but it is actually a simple rule to remember: If a method is used in operator notation, such as a * b, the method is invoked on the left operand, as in a.*(b)—unless the method name ends in a colon. If the method name ends in a colon, the method is invoked on the *right* operand. Therefore, in 1 :: twoThree, the :: method is invoked on twoThree, passing in1, like this: twoThree.::(1). Operator associativity will be described in more detail in Section 5.9.

Given that a shorthand way to specify an empty list is Nil, one way to initialize new lists is to string together elements with the cons operator, with Nil as the last element.[4] For example, the following script will produce the same output as the previous one, "List(1, 2, 3)":

```
val oneTwoThree = 1 :: 2 :: 3 :: Nil
println(oneTwoThree)
```

Scala's List is packed with useful methods, many of which are shown in Table 3.1. The full power of lists will be revealed in Chapter 16.

WHY NOT APPEND TO LISTS?

Class List does offer an "append" operation—it's written :+ and is explained in Chapter 24—but this operation is rarely used, because the time it takes to append to a list grows linearly with the size of the list, whereas prepending with :: takes constant time. If you want to build a list efficiently by appending elements, you can prepend them and when you're done call reverse. Or you can use a ListBuffer, a mutable list that does offer an append operation, and when you're done call toList. ListBuffer will be described inSection 22.2.

Some List methods and usages

What it is	What it does
List() or Nil	The empty List
List("Cool", "tools", "rule")	Creates a new List[String] with the three values"Cool", "tools", and "rule"
val thrill = "Will" :: "fill" :: "until" :: Nil	Creates a new List[String] with the three values"Will", "fill", and "until"
List("a", "b") ::: List("c", "d")	Concatenates two lists (returns a new List[String]with values "a", "b", "c", and "d")
thrill(2)	Returns the element at index 2 (zero based) of thethrill list (returns "until")
thrill.count(s => s.length == 4)	Counts the number of string elements in thrill that have length 4 (returns 2)
thrill.drop(2)	Returns the thrill list without its first 2 elements (returns List("until"))
thrill.dropRight(2)	Returns the thrill list without its rightmost 2 elements (returns List("Will"))
thrill.exists(s => s == "until")	Determines whether a string element exists in thrillthat has the value "until" (returns true)
thrill.filter(s => s.length == 4)	Returns a list of all elements, in order, of the thrilllist that have length 4 (returns List("Will", "fill"))
thrill.forall(s => s.endsWith("l"))	Indicates whether all elements in the thrill list end with the letter "l" (returns true)
thrill.foreach(s => print(s))	Executes the print statement on each of the strings in the thrill list (prints "Willfilluntil")
thrill.foreach(print)	Same as the previous, but more concise (also prints"Willfilluntil")
thrill.head	Returns the first element in the thrill list (returns"Will")
thrill.init	Returns a list of all but the last element in the thrilllist (returns List("Will", "fill"))
thrill.isEmpty	Indicates whether the thrill list is empty (returnsfalse)
thrill.last	Returns the last element in the thrill list (returns"until")
thrill.length	Returns the number of elements in the thrill list (returns 3)
thrill.map(s => s + "y")	Returns a list resulting from adding a "y" to each string element in the thrill list (returnsList("Willy", "filly", "untily"))
thrill.mkString(", ")	Makes a string with the elements of the list (returns"Will, fill, until")
thrill.filterNot(s => s.length == 4)	Returns a list of all elements, in order, of the thrilllist *except those* that have length 4 (returnsList("until"))
thrill.reverse	Returns a list containing all elements of the thrilllist in reverse order (returnsList("until", "fill", "Will"))
thrill.sort((s, t) => s.charAt(0).toLower < t.charAt(0).toLower)	Returns a list containing all elements of the thrilllist in alphabetical order of the first character lowercased (returns List("fill", "until", "Will"))
thrill.tail	Returns the thrill list minus its first element (returns List("fill", "until"))

STEP 9. USE TUPLES

Another useful container object is the *tuple*. Like lists, tuples are immutable, but unlike lists, tuples can contain different types of elements. Whereas a list might be a List[Int] or aList[String], a tuple could contain both an integer and a string at the same time. Tuples are very useful, for example, if you need to return multiple objects from a method. Whereas in Java you would often create a JavaBean-like class to hold the multiple return values, in Scala you can simply return a tuple. And it is simple: To instantiate a new tuple that holds some objects, just place the objects in parentheses, separated by commas. Once you have a tuple instantiated, you can access its elements individually with a dot, underscore, and the one-based index of the element. An example is shown in Listing 3.4:

```
val pair = (99, "Luftballons")
println(pair._1)
println(pair._2)
```

Listing 3.4 - Creating and using a tuple.

In the first line of Listing 3.4, you create a new tuple that contains the integer 99, as its first element, and the string, "Luftballons", as its second element. Scala infers the type of the tuple to be Tuple2[Int, String], and gives that type to the variable pair as well. In the second line, you access the _1 field, which will produce the first element, 99. The "." in the second line is the same dot you'd use to access a field or invoke a method. In this case you are accessing a field named _1. If you run this script, you'll see:

```
99
Luftballons
```

The actual type of a tuple depends on the number of elements it contains and the types of those elements. Thus, the type of (99, "Luftballons") is Tuple2[Int, String]. The type of('u', 'r', "the", 1, 4, "me") is Tuple6[Char, Char, String, Int, Int, String].[5]

ACCESSING THE ELEMENTS OF A TUPLE

You may be wondering why you can't access the elements of a tuple like the elements of a list, for example, with "pair(0)". The reason is that a list's apply method always returns the same type, but each element of a tuple may be a different type: _1 can have one result type, _2 another, and so on. These _N numbers are one-based, instead of zero-based, because starting with 1 is a tradition set by other languages with statically typed tuples, such as Haskell and ML.

STEP 10. USE SETS AND MAPS

Because Scala aims to help you take advantage of both functional and imperative styles, its collections libraries make a point to differentiate between mutable and immutable collections. For example, arrays are always mutable; lists are always immutable. Scala also provides mutable and immutable alternatives for sets and maps, but uses the same simple names for both versions. For sets and maps, Scala models mutability in the class hierarchy.

For example, the Scala API contains a base *trait* for sets, where a trait is similar to a Java interface. (You'll find out more about traits in Chapter 12.) Scala then provides two subtraits, one for mutable sets and another for immutable sets.

As you can see in Figure 3.2, these three traits all share the same simple name, Set. Their fully qualified names differ, however, because each resides in a different package. Concrete set classes in the Scala API, such as the HashSet classes shown in Figure 3.2, extend either the mutable or immutable Set trait. (Although in Java you "implement" interfaces, in Scala you "extend" or "mix in" traits.) Thus, if you want to use a HashSet, you can choose between mutable and immutable varieties depending upon your needs. The default way to create a set is shown in Listing 3.5:

```
var jetSet = Set("Boeing", "Airbus")
jetSet += "Lear"
println(jetSet.contains("Cessna"))
```

Listing 3.5 - Creating, initializing, and using an immutable set.

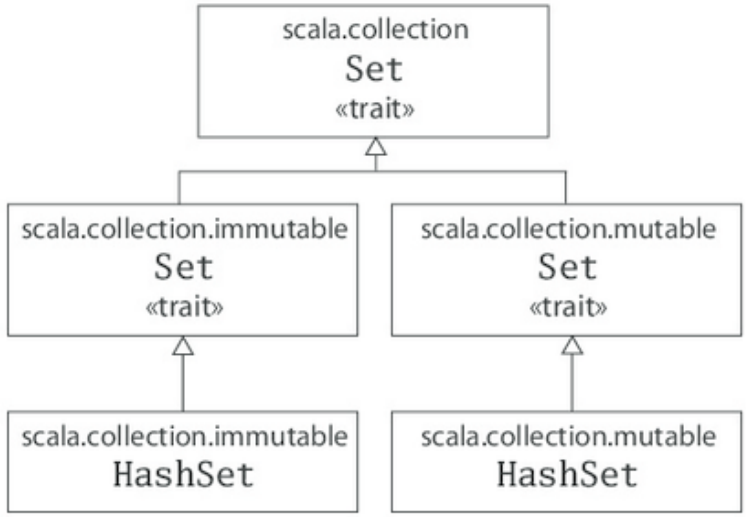

Figure 3.2 - Class hierarchy for Scala sets.

In the first line of code in Listing 3.5, you define a new var named jetSet and initialize it with an immutable set containing the two strings, "Boeing" and "Airbus". As this example shows, you can create sets in Scala similarly to how you create lists and arrays: by invoking a factory method named apply on a Set companion object. In Listing 3.5, you invoke apply on the companion object for scala.collection.immutable.Set, which returns an instance of a default, immutable Set. The Scala compiler infers jetSet's type to be the immutable Set[String].

To add a new element to a set, you call + on the set, passing in the new element. On both mutable and immutable sets, the + method will create and return a new set with the element added. In Listing 3.5, you're working with an immutable set. Although mutable sets offer an actual += method, immutable sets do not.

In this case, the second line of code, "jetSet += "Lear"", is essentially a shorthand for:

```
jetSet = jetSet + "Lear"
```

Thus, in the second line of Listing 3.5, you reassign the jetSet var with a new set containing"Boeing", "Airbus", and "Lear". Finally, the last line of Listing 3.5 prints out whether or not the set contains the string "Cessna". (As you'd expect, it prints false.)

If you want a mutable set, you'll need to use an *import*, as shown in Listing 3.6:

```
import scala.collection.mutable

val movieSet = mutable.Set("Hitch", "Poltergeist")
movieSet += "Shrek"
println(movieSet)
```

Listing 3.6 - Creating, initializing, and using a mutable set.

In the first line of Listing 3.6 you import the mutable Set. As with Java, an import statement allows you to use a simple name, such as Set, instead of the longer, fully qualified name. As a result, when you say Set on the third line, the compiler knows you meanscala.collection.mutable.Set. On that line, you initialize movieSet with a new mutable set that contains the strings "Hitch" and "Poltergeist". The subsequent line adds "Shrek" to the mutable set by calling the += method on the set, passing in the string "Shrek". As mentioned previously,+= is an actual method defined on mutable sets. Had you wanted to, instead of writingmovieSet += "Shrek", you could have written movieSet.+=("Shrek").[6]

Although the default set implementations produced by the mutable and immutable Setfactory methods shown thus far will likely be sufficient for most situations, occasionally you may want an explicit set class. Fortunately, the syntax is similar. Simply import that class you need, and use the factory method on its companion object. For example, if you need an immutable HashSet, you could do this:

```
import scala.collection.immutable.HashSet

val hashSet = HashSet("Tomatoes", "Chilies")
println(hashSet + "Coriander")
```

Another useful collection class in Scala is Map. As with sets, Scala provides mutable andimmutable versions of Map, using a class hierarchy. As you can see in Figure 3.3, the class hierarchy for maps looks a lot like the one for sets. There's a base Map trait in packagescala.collection, and two subtrait Maps: a mutable Map in scala.collection.mutable and an immutable one in scala.collection.immutable.

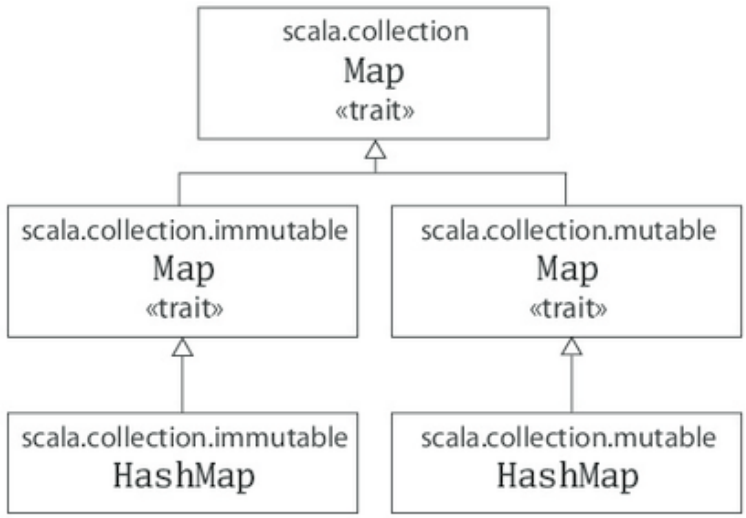

Figure 3.3 - Class hierarchy for Scala maps.

Implementations of Map, such as the HashMaps shown in the class hierarchy in Figure 3.3, extend either the mutable or immutable trait. You can create and initialize maps using factory methods similar to those used for arrays, lists, and sets.

```
import scala.collection.mutable

val treasureMap = mutable.Map[Int, String]()
treasureMap += (1 -> "Go to island.")
treasureMap += (2 -> "Find big X on ground.")
treasureMap += (3 -> "Dig.")
println(treasureMap(2))
```

Listing 3.7 - Creating, initializing, and using a mutable map.

For example, Listing 3.7 shows a mutable map in action. On the first line of Listing 3.7, you import the mutable Map. You then define a val named treasureMap, and initialize it with an empty mutable Map that has integer keys and string values. The map is empty because you pass nothing to the factory method (the parentheses in "Map[Int, String]()" are empty).[7] On the next three lines you add key/value pairs to the map using the -> and += methods. As illustrated previously, the Scala compiler transforms a binary operation expression like 1 -> "Go to island." into (1).->("Go to island."). Thus, when you say 1 -> "Go to island.", you are actually calling a method named -> on an integer with the value 1, passing in a string with the value "Go to island." This -> method, which you can invoke on any object in a Scala program, returns a two-element tuple containing the key and value.[8] You then pass this tuple to the+= method of the map object to which treasureMap refers. Finally, the last line prints the value that corresponds to the key 2 in the treasureMap.

If you run this code, it will print:

```
Find big X on ground.
```

If you prefer an immutable map, no import is necessary, as immutable is the default map. An example is shown in Listing 3.8:

```
val romanNumeral = Map(
  1 -> "I", 2 -> "II", 3 -> "III", 4 -> "IV", 5 -> "V"
)
println(romanNumeral(4))
```

Listing 3.8 - Creating, initializing, and using an immutable map.

Given there are no imports, when you say Map in the first line of Listing 3.8, you'll get the default: a scala.collection.immutable.Map. You pass five key/value tuples to the map's factory method, which returns an immutable Map containing the passed key/value pairs. If you run the code in Listing 3.8 it will print "IV".

STEP 11. LEARN TO RECOGNIZE THE FUNCTIONAL STYLE

As mentioned in Chapter 1, Scala allows you to program in an imperative style, but encourages you to adopt a more functional style. If you are coming to Scala from an imperative background—for example, if you are a Java programmer—one of the main challenges you may face when learning Scala is figuring out how to program in the functional style. We realize this style might be unfamiliar at first, and in this book we try hard to guide you through the transition. It will require some work on your part, and we encourage you to make the effort. If you come from an imperative background, we believe that learning to program in a functional style will not only make you a better Scala programmer, it will expand your horizons and make you a better programmer in general.

The first step is to recognize the difference between the two styles in code. One telltale sign is that if code contains any vars, it is probably in an imperative style. If the code contains no vars at all—*i.e.*, it contains *only* vals—it is probably in a functional style. One way to move towards a functional style, therefore, is to try to program without vars.

If you're coming from an imperative background, such as Java, C++, or C#, you may think ofvar as a regular variable and val as a special kind of variable. On the other hand, if you're coming from a functional background, such as Haskell, OCaml, or Erlang, you might think ofval as a regular variable and var as akin to blasphemy. The Scala perspective, however, is thatval and var are just two different tools in your toolbox, both useful, neither inherently evil. Scala encourages you to lean towards vals, but ultimately reach for the best tool given the job at hand. Even if you agree with this balanced philosophy, however, you may still find it challenging at first to figure out how to get rid of vars in your code.

Consider the following while loop example, adapted from Chapter 2, which uses a var and is therefore in the imperative style:

```
def printArgs(args: Array[String]): Unit = {
  var i = 0
  while (i < args.length) {
    println(args(i))
    i += 1
```

```
  }
}
```

You can transform this bit of code into a more functional style by getting rid of the var, for example, like this:

```
def printArgs(args: Array[String]): Unit = {
  for (arg <- args)
    println(arg)
}
```

or this:

```
def printArgs(args: Array[String]): Unit = {
  args.foreach(println)
}
```

This example illustrates one benefit of programming with fewer vars. The refactored (more functional) code is clearer, more concise, and less error-prone than the original (more imperative) code. The reason Scala encourages a functional style is that it can help you write more understandable, less error-prone code.

But you can go even further. The refactored printArgs method is not *purely* functional because it has side effects—in this case, its side effect is printing to the standard output stream. The telltale sign of a function with side effects is that its result type is Unit. If a function isn't returning any interesting value, which is what a result type of Unit means, the only way that function can make a difference in the world is through some kind of side effect. A more functional approach would be to define a method that formats the passed args for printing, but just returns the formatted string, as shown in Listing 3.9:

```
def formatArgs(args: Array[String]) = args.mkString("\n")
```

Listing 3.9 - A function without side effects or vars.

Now you're really functional: no side effects or vars in sight. The mkString method, which you can call on any iterable collection (including arrays, lists, sets, and maps), returns a string consisting of the result of calling toString on each element, separated by the passed string. Thus if args contains three elements "zero", "one", and "two", formatArgs will return"zero\none\ntwo". Of course, this function doesn't actually print anything out like the printArgsmethods did, but you can easily pass its result to println to accomplish that:

```
println(formatArgs(args))
```

Every useful program is likely to have side effects of some form; otherwise, it wouldn't be able to provide value to the outside world. Preferring methods without side effects encourages you to design programs where side-effecting code is minimized. One benefit of this approach is that it can help make your programs easier to test.

For example, to test any of the three printArgs methods shown earlier in this section, you'd need to redefine println, capture the output passed to it, and make sure it is what you expect. By contrast, you could test the formatArgs function simply by checking its result:

```
val res = formatArgs(Array("zero", "one", "two"))
assert(res == "zero\none\ntwo")
```

Scala's assert method checks the passed Boolean and if it is false, throws AssertionError. If the passed Boolean is true, assert just returns quietly. You'll learn more about assertions and tests in Chapter 14.

That said, bear in mind that neither vars nor side effects are inherently evil. Scala is not a pure functional language that forces you to program everything in the functional style. Scala is a hybrid imperative/functional language. You may find that in some situations an imperative style is a better fit for the problem at hand, and in such cases you should not hesitate to use it. To help you learn how to program without vars, however, we'll show you many specific examples of code with vars and how to transform those vars to vals in Chapter 7.

A BALANCED ATTITUDE FOR SCALA PROGRAMMERS

Prefer vals, immutable objects, and methods without side effects. Reach for them first. Use vars, mutable objects, and methods with side effects when you have a specific need and justification for them.

STEP 12. READ LINES FROM A FILE

Scripts that perform small, everyday tasks often need to process lines in files. In this section, you'll build a script that reads lines from a file and prints them out prepended with the number of characters in each line. The first version is shown in Listing 3.10:

```
import scala.io.Source

if (args.length > 0) {

  for (line <- Source.fromFile(args(0)).getLines())
    println(line.length + " " + line)
}
else
  Console.err.println("Please enter filename")
```

Listing 3.10 - Reading lines from a file.

This script starts with an import of a class named Source from package scala.io. It then checks to see if at least one argument was specified on the command line. If so, the first argument is interpreted as a filename to open and process. The expression Source.fromFile(args(0))attempts to open the specified file and returns a Source object, on which you call getLines. ThegetLines method returns an Iterator[String], which provides one line on each iteration, excluding the end-of-line character. The for expression iterates through these lines and prints for each the length of the line, a space, and the line itself. If there were no arguments supplied on the command line, the final else clause will print a

message to the standard error stream. If you place this code in a file named countchars1.scala, and run it on itself with:

```
$ scala countchars1.scala countchars1.scala
```

You should see:

```
22 import scala.io.Source
0
22 if (args.length > 0) {
0
51   for (line <- Source.fromFile(args(0)).getLines())
37     println(line.length + " " + line)
1 }
4 else
46   Console.err.println("Please enter filename")
```

Although the script in its current form prints out the needed information, you may wish to line up the numbers, right adjusted, and add a pipe character, so that the output looks instead like:

```
22 | import scala.io.Source
 0 |
22 | if (args.length > 0) {
 0 |
51 |   for (line <- Source.fromFile(args(0)).getLines())
37 |     println(line.length + " " + line)
 1 | }
 4 | else
46 |   Console.err.println("Please enter filename")
```

To accomplish this, you can iterate through the lines twice. The first time through you'll determine the maximum width required by any line's character count. The second time through you'll print the output, using the maximum width calculated previously. Because you'll be iterating through the lines twice, you may as well assign them to a variable:

```
val lines = Source.fromFile(args(0)).getLines().toList
```

The final toList is required because the getLines method returns an iterator. Once you've iterated through an iterator, it is spent. By transforming it into a list via the toList call, you gain the ability to iterate as many times as you wish, at the cost of storing all lines from the file in memory at once. The lines variable, therefore, references a list of strings that contains the contents of the file specified on the command line. Next, because you'll be calculating the width of each line's character count twice, once per iteration, you might factor that expression out into a small function, which calculates the character width of the passed string's length:

```
def widthOfLength(s: String) = s.length.toString.length
```

With this function, you could calculate the maximum width like this:

```
var maxWidth = 0
for (line <- lines)
  maxWidth = maxWidth.max(widthOfLength(line))
```

Here you iterate through each line with a for expression, calculate the character width of that line's length, and, if it is larger than the current maximum, assign it to maxWidth, a var that was initialized to 0. (The max method, which you can invoke on any Int, returns the greater of the value on which it was invoked and the value passed to it.) Alternatively, if you prefer to find the maximum without vars, you could first find the longest line like this:

```
val longestLine = lines.reduceLeft(
  (a, b) => if (a.length > b.length) a else b
)
```

The reduceLeft method applies the passed function to the first two elements in lines, then applies it to the result of the first application and the next element in lines, and so on, all the way through the list. On each such application, the result will be the longest line encountered so far because the passed function, (a, b) => if (a.length > b.length) a else b, returns the longest of the two passed strings. "reduceLeft" will return the result of the last application of the function, which in this case will be the longest string element contained in lines.

Given this result, you can calculate the maximum width by passing the longest line towidthOfLength:

```
val maxWidth = widthOfLength(longestLine)
```

All that remains is to print out the lines with proper formatting. You can do that like this:

```
for (line <- lines) {
  val numSpaces = maxWidth - widthOfLength(line)
  val padding = " " * numSpaces
  println(padding + line.length + " | " + line)
}
```

In this for expression, you once again iterate through the lines. For each line, you first calculate the number of spaces required before the line length and assign it to numSpaces. Then you create a string containing numSpaces spaces with the expression " " * numSpaces. Finally, you print out the information with the desired formatting. The entire script looks as shown inListing 3.11:

```
import scala.io.Source

def widthOfLength(s: String) = s.length.toString.length

if (args.length > 0) {

  val lines = Source.fromFile(args(0)).getLines().toList

  val longestLine = lines.reduceLeft(
    (a, b) => if (a.length > b.length) a else b
  )
  val maxWidth = widthOfLength(longestLine)

  for (line <- lines) {
    val numSpaces = maxWidth - widthOfLength(line)
    val padding = " " * numSpaces
    println(padding + line.length + " | " + line)
  }
}
```

```
else
  Console.err.println("Please enter filename")
```

Listing 3.11 - Printing formatted character counts for the lines of a file.

CONCLUSION

With the knowledge you've gained in this chapter, you should be able to start using Scala for small tasks, especially scripts. In later chapters, we will dive further into these topics and introduce other topics that weren't even hinted at here.

Footnotes for Chapter 3:

[1] This to method actually returns not an array but a different kind of sequence, containing the values 0, 1, and 2, which the for expression iterates over. Sequences and other collections will be described in Chapter 17.

[2] Variable-length argument lists, or *repeated parameters*, are described in Section 8.8.

[3] You don't need to say new List because "List.apply()" is defined as a factory method on thescala.List *companion object*. You'll read more on companion objects in Section 4.3.

[4] The reason you need Nil at the end is that :: is defined on class List. If you try to just say1 :: 2 :: 3, it won't compile because 3 is an Int, which doesn't have a :: method.

[5] Although conceptually you could create tuples of any length, currently the Scala library only defines them up to Tuple22.

[6] Because the set in Listing 3.6 is mutable, there is no need to reassign movieSet, which is why it can be a val. By contrast, using += with the immutable set in Listing 3.5 required reassigning jetSet, which is why it must be a var.

[7] The explicit type parameterization, "[Int, String]", is required in Listing 3.7 because without any values passed to the factory method, the compiler is unable to infer the map's type parameters. By contrast, the compiler can infer the type parameters from the values passed to the map factory shown in Listing 3.8, thus no explicit type parameters are needed.

[8] The Scala mechanism that allows you to invoke -> on any object, *implicit conversion*, will be covered in Chapter 21.