Chapter 7

Built-in Control Structures

Scala has only a handful of built-in control structures. The only control structures
are if,while, for, try, match, and function calls. The reason Scala has so few is that it has included function literals since its inception. Instead of accumulating one higher-level control structure after another in the base syntax, Scala accumulates them in libraries. (Chapter 9will show precisely how that is done.) This chapter will show those few control structures that are built in.

One thing you will notice is that almost all of Scala's control structures result in some value. This is the approach taken by functional languages, where programs are viewed as computing a value, thus the components of a program should also compute values. You can also view this approach as the logical conclusion of a trend already present in imperative languages. In imperative languages, function calls can return a value, even though having the called function update an output variable passed as an argument would work just as well. In addition, imperative languages often have a ternary operator (such as the ?: operator of C, C++, and Java), which behaves exactly like if, but results in a value. Scala adopts this ternary operator model, but calls it if. In other words, Scala's if can result in a value. Scala then continues this trend by having for, try, and match also result in values.

Programmers can use these result values to simplify their code, just as they use return values of functions. Without this facility, the programmer must create temporary variables just to hold results that are calculated inside a control structure. Removing these temporary variables makes the code a little simpler, and it also prevents many bugs where you set the variable in one branch but forget to set it in another.

Overall, Scala's basic control structures, minimal as they are, provide all of the essentials from imperative languages. Further, they allow you to shorten your code by consistently having result values. To show you how this works, we'll take a closer look at each of Scala's basic control structures.

7.1 IF EXPRESSIONS

Scala's if works just like in many other languages. It tests a condition and then executes one of two code branches depending on whether the condition holds true. Here is a common example, written in an imperative style:

```
var filename = "default.txt"
if (!args.isEmpty)
  filename = args(0)
```

This code declares a variable, filename, and initializes it to a default value. It then uses an ifexpression to check whether any arguments were supplied to the program. If so, it changes the variable to hold the value specified in the argument list. If no arguments were supplied, it leaves the variable set to the default value.

This code can be written more nicely because, as mentioned in Step 3 in Chapter 2, Scala's ifis an expression that results in a value. Listing 7.1 shows how you can accomplish the same effect as the previous example, without using any vars:

```
val filename =
  if (!args.isEmpty) args(0)
  else "default.txt"
```

Listing 7.1 - Scala's idiom for conditional initialization.

This time, the if has two branches. If args is not empty, the initial element, args(0), is chosen; otherwise, the default value is chosen. The if expression results in the chosen value, and thefilename variable is initialized with that value. This code is slightly shorter, but its real advantage is that it uses a val instead of a var. Using a val is the functional style, and it helps you in much the same way as a final variable in Java. It tells readers of the code that the variable will never change, saving them from scanning all code in the variable's scope to see if it ever changes.

A second advantage to using a val instead of a var is that it better supports *equational reasoning*. The introduced variable is *equal* to the expression that computes it, assuming that expression has no side effects. Thus, any time you are about to write the variable name, you could instead write the expression. Instead of println(filename), for example, you could just write this:

```
println(if (!args.isEmpty) args(0) else "default.txt")
```

The choice is yours. You can write it either way. Using vals helps you safely make this kind of refactoring as your code evolves over time.

Look for opportunities to use vals. They can make your code both easier to read and easier to refactor.

7.2 WHILE LOOPS

Scala's while loop behaves as in other languages. It has a condition and a body, and the body is executed over and over as long as the condition holds true. Listing 7.2 shows an example:

```
def gcdLoop(x: Long, y: Long): Long = {
  var a = x
  var b = y
  while (a != 0) {
    val temp = a
    a = b % a
    b = temp
  }
  b
}
```

Listing 7.2 - Calculating greatest common divisor with a while loop.

Scala also has a do-while loop. This works like the while loop except that it tests the condition after the loop body instead of before. Listing 7.3 shows a Scala script that uses a do-while to echo lines read from the standard input, until an empty line is entered:

```
    var line = ""
    do {
      line = readLine()
      println("Read: " + line)
    } while (line != "")
```

Listing 7.3 - Reading from the standard input with do-while.

The while and do-while constructs are called "loops," not expressions, because they don't result in an interesting value. The type of the result is Unit. It turns out that a value (and in fact, only one value) exists whose type is Unit. It is called the *unit value* and is written (). The existence of () is how Scala's Unit differs from Java's void. Try this in the interpreter:

```
  scala> def greet() = { println("hi") }
  greet: ()Unit

  scala> () == greet()
  hi
  res0: Boolean = true
```

Because no equals sign precedes its body, greet is defined to be a procedure with a result type of Unit. Therefore, greet returns the unit value, (). This is confirmed in the next line: comparing the greet's result for equality with the unit value, (), yields true.

One other construct that results in the unit value, which is relevant here, is reassignment tovars. For example, were you to attempt to read lines in Scala using the following while loop idiom from Java (and C and C++), you'll run into trouble:

```
  var line = ""
  while ((line = readLine()) != "") // This doesn't work!
    println("Read: " + line)
```

When you compile this code, Scala will give you a warning that comparing values of type Unitand String using != will always yield true. Whereas in Java, assignment results in the value assigned (in this case a line from the standard input), in Scala assignment always results in the unit value, (). Thus, the value of the assignment "line = readLine()" will always be () and never be "". As a result, this while loop's condition will never be false, and the loop will, therefore, never terminate.

Because the while loop results in no value, it is often left out of pure functional languages. Such languages have expressions, not loops. Scala includes the while loop nonetheless because sometimes an imperative solution can be more readable, especially to programmers with a predominantly imperative background. For example, if you want to code an algorithm that repeats a process until some condition changes, a while loop can express it directly while the functional alternative, which likely uses recursion, may be less obvious to some readers of the code.

For example, Listing 7.4 shows an alternate way to determine a greatest common divisor of two numbers.[1] Given the same two values for x and y, the gcd function shown in Listing 7.4will return the same result as the gcdLoop function, shown in Listing 7.2. The difference between these two approaches is that gcdLoop is written in an imperative style, using vars and and a while loop, whereas gcd is written in a more functional style that involves recursion (gcdcalls itself) and requires no vars.

```
def gcd(x: Long, y: Long): Long =
  if (y == 0) x else gcd(y, x % y)
```

Listing 7.4 - Calculating greatest common divisor with recursion.

In general, we recommend you challenge while loops in your code in the same way you challenge vars. In fact, while loops and vars often go hand in hand. Because while loops don't result in a value, to make any kind of difference to your program, a while loop will usually either need to update vars or perform I/O. You can see this in action in the gcdLoop example shown previously. As that while loop does its business, it updates vars a and b. Thus, we suggest you be a bit suspicious of while loops in your code. If there isn't a good justification for a particular while or do-while loop, try to find a way to do the same thing without it.

7.3 FOR EXPRESSIONS

Scala's for expression is a Swiss army knife of iteration. It lets you combine a few simple ingredients in different ways to express a wide variety of iterations. Simple uses enable common tasks such as iterating through a sequence of integers. More advanced expressions can iterate over multiple collections of different kinds, filter out elements based on arbitrary conditions, and produce new collections.

Iteration through collections

The simplest thing you can do with for is to iterate through all the elements of a collection. For example, Listing 7.5 shows some code that prints out all files in the current directory. The I/O is performed using the Java API. First, we create a java.io.File on the current directory,".", and call its listFiles method. This method returns an array of File objects, one per directory and file contained in the current directory. We store the resulting array in thefilesHere variable.

```
val filesHere = (new java.io.File(".")).listFiles

for (file <- filesHere)
  println(file)
```

Listing 7.5 - Listing files in a directory with a for expression.

With the "file <- filesHere" syntax, which is called a *generator*, we iterate through the elements of filesHere. In each iteration, a new val named file is initialized with an element value. The compiler infers the type of file to be File, because filesHere is an Array[File]. For each iteration, the body of the for expression, println(file), will be executed. Because File'stoString method yields the name of the file or directory, the names of all the files and directories in the current directory will be printed.

The for expression syntax works for any kind of collection, not just arrays.[2] One convenient special case is the Range type, which you briefly saw in Table 5.4 here. You can create Ranges using syntax like "1 to 5" and can iterate through them with a for. Here is a simple example:

```
scala> for (i <- 1 to 4)
         println("Iteration " + i)
Iteration 1
Iteration 2
Iteration 3
Iteration 4
```

If you don't want to include the upper bound of the range in the values that are iterated over, use until instead of to:

```
scala> for (i <- 1 until 4)
         println("Iteration " + i)
Iteration 1
Iteration 2
Iteration 3
```

Iterating through integers like this is common in Scala, but not nearly as much as in other languages. In other languages, you might use this facility to iterate through an array, like this:

```
// Not common in Scala...
for (i <- 0 to filesHere.length - 1)
  println(filesHere(i))
```

This for expression introduces a variable i, sets it in turn to each integer between 0 andfilesHere.length - 1, and executes the body of the for expression for each setting of i. For each setting of i, the i'th element of filesHere is extracted and processed.

The reason this kind of iteration is less common in Scala is that you can just iterate over the collection directly. When you do, your code becomes shorter and you sidestep many of the off-by-one errors that can arise when iterating through arrays. Should you start at 0 or 1? Should you add -1, +1, or nothing to the final index? Such questions are easily answered, but also easily answered wrong. It is safer to avoid such questions entirely.

Filtering

Sometimes you don't want to iterate through a collection in its entirety; you want to filter it down to some subset. You can do this with a for expression by adding a *filter*, an if clause inside the for's parentheses. For example, the code shown in Listing 7.6 lists only those files in the current directory whose names end with ".scala":

```
  val filesHere = (new java.io.File(".")).listFiles

  for (file <- filesHere if file.getName.endsWith(".scala"))
    println(file)
```

Listing 7.6 - Finding .scala files using a for with a filter.

You could alternatively accomplish the same goal with this code:

```
for (file <- filesHere)
  if (file.getName.endsWith(".scala"))
    println(file)
```

This code yields the same output as the previous code, and likely looks more familiar to programmers with an imperative background. The imperative form, however, is only an option because this particular for expression is executed for its printing side-effects and results in the unit value (). As demonstrated later in this section, the for expression is called an "expression" because it can result in an interesting value, a collection whose type is determined by the for expression's <- clauses.

You can include more filters if you want. Just keep adding if clauses. For example, to be extra defensive, the code in Listing 7.7 prints only files and not directories. It does so by adding a filter that checks the file's isFile method.

```
for (
  file <- filesHere
  if file.isFile
  if file.getName.endsWith(".scala")
) println(file)
```

Listing 7.7 - Using multiple filters in a for expression.

Nested iteration

If you add multiple <- clauses, you will get nested "loops." For example, the for expression shown in Listing 7.8 has two nested loops. The outer loop iterates through filesHere, and the inner loop iterates through fileLines(file) for any file that ends with .scala.

```
def fileLines(file: java.io.File) =
  scala.io.Source.fromFile(file).getLines().toList

def grep(pattern: String) =
  for (
    file <- filesHere
    if file.getName.endsWith(".scala");
    line <- fileLines(file)
    if line.trim.matches(pattern)
  ) println(file + ": " + line.trim)

grep(".*gcd.*")
```

Listing 7.8 - Using multiple generators in a for expression.

If you prefer, you can use curly braces instead of parentheses to surround the generators and filters. One advantage to using curly braces is that you can leave off some of the semicolons that are needed when you use parentheses because, as explained in Section 4.2, the Scala compiler will not infer semicolons while inside parentheses.

Mid-stream variable bindings

Note that the previous code repeats the expression line.trim. This is a non-trivial computation, so you might want to only compute it once. You can do this by binding the result to a new variable using an equals sign (=). The bound variable is introduced and used just like a val, only with the val keyword left out. Listing 7.9 shows an example.

```
def grep(pattern: String) =
      for {
        file <- filesHere
        if file.getName.endsWith(".scala")
        line <- fileLines(file)
        trimmed = line.trim
        if trimmed.matches(pattern)
      } println(file + ": " + trimmed)

    grep(".*gcd.*")
```

Listing 7.9 - Mid-stream assignment in a for expression.

In Listing 7.9, a variable named trimmed is introduced halfway through the for expression. That variable is initialized to the result of line.trim. The rest of the for expression then uses the new variable in two places, once in an if and once in println.

Producing a new collection

While all of the examples so far have operated on the iterated values and then forgotten them, you can also generate a value to remember for each iteration. To do so, you prefix the body of the for expression by the keyword yield. For example, here is a function that identifies the.scala files and stores them in an array:

```
def scalaFiles =
  for {
    file <- filesHere
    if file.getName.endsWith(".scala")
  } yield file
```

Each time the body of the for expression executes, it produces one value, in this case simplyfile. When the for expression completes, the result will include all of the yielded values contained in a single collection. The type of the resulting collection is based on the kind of collections processed in the iteration clauses. In this case the result is an Array[File], becausefilesHere is an array and the type of the yielded expression is File.

Be careful, by the way, where you place the yield keyword. The syntax of a for-yield expression is like this:

```
for clauses yield body
```

The yield goes before the entire body. Even if the body is a block surrounded by curly braces, put the yield before the first curly brace, not before the last expression of the block. Avoid the temptation to write things like this:

```
for (file <- filesHere if file.getName.endsWith(".scala")) {
  yield file  // Syntax error!
}
```

For example, the for expression shown in Listing 7.10 first transforms the Array[File] namedfilesHere, which contains all files in the current directory, to one that contains only .scala files. For each of these it generates an Iterator[String], the result of the fileLines method, whose definition is shown in Listing

7.8. An Iterator offers methods next and hasNext that allow you to iterate over a collection of elements. This initial iterator is transformed into anotherIterator[String] containing only trimmed lines that include the substring "for". Finally, for each of these, an integer length is yielded. The result of this for expression is an Array[Int]containing those lengths.

```
    val forLineLengths =
      for {
        file <- filesHere
        if file.getName.endsWith(".scala")
        line <- fileLines(file)
        trimmed = line.trim
        if trimmed.matches(".*for.*")
      } yield trimmed.length
```

Listing 7.10 - Transforming an Array[File] to Array[Int] with a for.

At this point, you have seen all the major features of Scala's for expression, but we went through them rather quickly. A more thorough coverage of for expressions is given in Chapter 23.

7.4 EXCEPTION HANDLING WITH TRY EXPRESSIONS

Scala's exceptions behave just like in many other languages. Instead of returning a value in the normal way, a method can terminate by throwing an exception. The method's caller can either catch and handle that exception, or it can itself simply terminate, in which case the exception propagates to the caller's caller. The exception propagates in this way, unwinding the call stack, until a method handles it or there are no more methods left.

Throwing exceptions

Throwing an exception in Scala looks the same as in Java. You create an exception object and then throw it with the throw keyword:

```
  throw new IllegalArgumentException
```

Although it may seem somewhat paradoxical, in Scala, throw is an expression that has a result type. Here's an example where result type matters:

```
  val half =
    if (n % 2 == 0)
      n / 2
    else
      throw new RuntimeException("n must be even")
```

What happens here is that if n is even, half will be initialized to half of n. If n is not even, an exception will be thrown before half can be initialized to anything at all. Because of this, it is safe to treat a thrown exception as any kind of value whatsoever. Any context that tries to use the return from a throw will never get to do so, and thus no harm will come.

Technically, an exception throw has type Nothing. You can use a throw as an expression even though it will never actually evaluate to anything. This little bit of technical gymnastics might sound weird, but is frequently useful in cases like the previous example. One branch of an ifcomputes a value, while the other throws an exception and computes Nothing. The type of the whole if expression is then the type of that branch which does compute something. TypeNothing is discussed further in Section 11.3.

Catching exceptions

You catch exceptions using the syntax shown in Listing 7.11 The syntax for catch clauses was chosen for its consistency with an important part of Scala: *pattern matching*. Pattern matching, a powerful feature, is described briefly in this chapter and in more detail inChapter 15.

```
import java.io.FileReader
import java.io.FileNotFoundException
import java.io.IOException

try {
  val f = new FileReader("input.txt")
  // Use and close file
} catch {
  case ex: FileNotFoundException => // Handle missing file
  case ex: IOException => // Handle other I/O error
}
```

Listing 7.11 - A try-catch clause in Scala.

The behavior of this try-catch expression is the same as in other languages with exceptions. The body is executed, and if it throws an exception, each catch clause is tried in turn. In this example, if the exception is of type FileNotFoundException, the first clause will execute. If it is of type IOException, the second clause will execute. If the exception is of neither type, the try-catchwill terminate and the exception will propagate further.

Note

One difference you'll quickly notice in Scala is that, unlike Java, Scala does not require you to catch checked exceptions or declare them in a throws clause. You can declare a throws clause if you wish with the @throwsannotation, but it is not required. See Section 31.2 for more information on@throws.

The finally clause

You can wrap an expression with a finally clause if you want to cause some code to execute no matter how the expression terminates. For example, you might want to be sure an open file gets closed even if a method exits by throwing an exception. Listing 7.12 shows an example.[3]

```
import java.io.FileReader

val file = new FileReader("input.txt")
try {
  // Use the file
} finally {
  file.close()  // Be sure to close the file
}
```

Listing 7.12 - A try-finally clause in Scala.

Note

Listing 7.12 shows the idiomatic way to ensure a non-memory resource, such as a file, socket, or database connection, is closed. First you acquire the resource. Then you start a try block in which you use the resource. Lastly, you close the resource in a finally block. This idiom is the same in Scala as in Java; alternatively, in Scala you can employ a technique called the *loan pattern* to achieve the same goal more concisely. The loan pattern will be described in Section 9.4.

Yielding a value

As with most other Scala control structures, try-catch-finally results in a value. For example,Listing 7.13 shows how you can try to parse a URL but use a default value if the URL is badly formed. The result is that of the try clause if no exception is thrown, or the relevant catchclause if an exception is thrown and caught. If an exception is thrown but not caught, the expression has no result at all. The value computed in the finally clause, if there is one, is dropped. Usually finally clauses do some kind of clean up, such as closing a file. Normally, they should not change the value computed in the main body or a catch clause of the try.

```
import java.net.URL
import java.net.MalformedURLException

def urlFor(path: String) =
  try {
    new URL(path)
  } catch {
    case e: MalformedURLException =>
      new URL("http://www.scala-lang.org")
  }
```

Listing 7.13 - A catch clause that yields a value.

If you're familiar with Java, it's worth noting that Scala's behavior differs from Java only because Java's try-finally does not result in a value. As in Java, if a finally clause includes an explicit return statement, or throws an exception, that return value or exception will "overrule" any previous one that originated in the try block or one of its catch clauses. For example, given this, rather contrived, function definition:

```
def f(): Int = try return 1 finally return 2
```

calling f() results in 2. By contrast, given:

```
def g(): Int = try 1 finally 2
```

calling g() results in 1. Both of these functions exhibit behavior that could surprise most programmers, so it's usually best to avoid returning values from finally clauses. The best way to think of finally clauses is as a way to ensure some side effect happens, such as closing an open file.

7.5 MATCH EXPRESSIONS

Scala's match expression lets you select from a number of *alternatives*, just like switchstatements in other languages. In general a match expression lets you select using arbitrary*patterns*, which will be described in Chapter 15. The general form can wait. For now, just consider using match to select among a number of alternatives.

As an example, the script in Listing 7.14 reads a food name from the argument list and prints a companion to that food. This match expression examines firstArg, which has been set to the first argument out of the argument list. If it is the string "salt", it prints "pepper", while if it is the string "chips", it prints "salsa", and so on. The default case is specified with an underscore (_), a wildcard symbol frequently used in Scala as a placeholder for a completely unknown value.

```
val firstArg = if (args.length > 0) args(0) else ""

firstArg match {
  case "salt" => println("pepper")
  case "chips" => println("salsa")
  case "eggs" => println("bacon")
  case _ => println("huh?")
}
```

Listing 7.14 - A match expression with side effects.

There are a few important differences from Java's switch statement. One is that any kind of constant, as well as other things, can be used in cases in Scala, not just the integer-type, enum, and string constants of Java's case statements. In Listing 7.14, the alternatives are strings. Another difference is that there are no breaks at the end of each alternative. Instead the break is implicit, and there is no fall through from one alternative to the next. The common case—not falling through—becomes shorter, and a source of errors is avoided because programmers can no longer fall through by accident.

The most significant difference from Java's switch, however, may be that match expressions result in a value. In the previous example, each alternative in the match expression prints out a value. It would work just as well to yield the value rather than print it, as shown in Listing 7.15. The value that results from this match expression is stored in the friend variable. Aside from the code getting shorter (in number of tokens anyway), the code now disentangles two separate concerns: first it chooses a food and then prints it.

```
val firstArg = if (!args.isEmpty) args(0) else ""

val friend =
  firstArg match {
    case "salt" => "pepper"
    case "chips" => "salsa"
    case "eggs" => "bacon"
    case _ => "huh?"
  }

println(friend)
```

Listing 7.15 - A match expression that yields a value.

7.6 LIVING WITHOUT BREAK AND CONTINUE

You may have noticed that there has been no mention of break or continue. Scala leaves out these commands because they do not mesh well with function literals, a feature described in the next chapter. It is clear what continue means inside a while loop, but what would it mean inside a function literal? While Scala supports both imperative and functional styles of programming, in this case it leans slightly towards functional programming in exchange for simplifying the language. Do not worry, though. There are many ways to program withoutbreak and continue, and if you take advantage of function literals, those alternatives can often be shorter than the original code.

The simplest approach is to replace every continue by an if and every break by a boolean variable. The boolean variable indicates whether the enclosing while loop should continue. For example, suppose you are searching through an argument list for a string that ends with ".scala" but does not start with a hyphen. In Java you could—if you were quite fond of whileloops, break, and continue—write the following:

```
int i = 0;                // This is Java
boolean foundIt = false;
while (i < args.length) {
  if (args[i].startsWith("-")) {
    i = i + 1;
    continue;
  }
  if (args[i].endsWith(".scala")) {
    foundIt = true;
    break;
  }
  i = i + 1;
}
```

To transliterate this Java code directly to Scala, instead of doing an if and then a continue, you could write an if that surrounds the entire remainder of the while loop. To get rid of the break, you would normally add a boolean variable indicating whether to keep going, but in this case you can reuse foundIt. Using both of these tricks, the code ends up looking as shown in Listing 7.16.

```
var i = 0
var foundIt = false

while (i < args.length && !foundIt) {
  if (!args(i).startsWith("-")) {
    if (args(i).endsWith(".scala"))
      foundIt = true
  }
  i = i + 1
}
```

Listing 7.16 - Looping without break or continue.

This Scala code in Listing 7.16 is quite similar to the original Java code. All the basic pieces are still there and in the same order. There are two reassignable variables and a while loop. Inside the loop, there is a test that i is less than args.length, a check for "-", and a check for".scala".

If you wanted to get rid of the vars in Listing 7.16, one approach you could try is to rewrite the loop as a recursive function. You could, for example, define a searchFrom function that takes an integer as an input, searches forward from there, and then returns the index of the desired argument. Using this technique the code would look as shown in Listing 7.17:

```
def searchFrom(i: Int): Int =
  if (i >= args.length) -1
  else if (args(i).startsWith("-")) searchFrom(i + 1)
  else if (args(i).endsWith(".scala")) i
  else searchFrom(i + 1)

val i = searchFrom(0)
```

Listing 7.17 - A recursive alternative to looping with vars.

The version in Listing 7.17 gives a human-meaningful name to what the function does, and it uses recursion to substitute for looping. Each continue is replaced by a recursive call with i + 1as the argument, effectively skipping to the next integer. Many people find this style of programming easier to understand, once they get used to the recursion.

Note

The Scala compiler will not actually emit a recursive function for the code shown inListing 7.17. Because all of the recursive calls are in *tail-call* position, the compiler will generate code similar to a while loop. Each recursive call will be implemented as a jump back to the beginning of the function. Tail-call optimization is discussed inSection 8.9.

If after all this discussion you still feel the need to use break, there's help in Scala's standard library. Class Breaks in package scala.util.control offers a break method, which can be used to exit an enclosing block that's marked with breakable. Here is an example how this library-supplied break method could be applied:

```
import scala.util.control.Breaks._
import java.io._

val in = new BufferedReader(new InputStreamReader(System.in))

breakable {
  while (true) {
    println("? ")
    if (in.readLine() == "") break
  }
}
```

This will repeatedly read non-empty lines from the standard input. Once the user enters an empty line, control flow exits from the enclosing breakable, and with it the while loop.

The Breaks class implements break by throwing an exception that is caught by an enclosing application of the breakable method. Therefore, the call to break does not need to be in the same method as the call to breakable.

7.7 VARIABLE SCOPE

Now that you've seen Scala's built-in control structures, we'll use them in this section to explain how scoping works in Scala.

FAST TRACK FOR JAVA PROGRAMMERS

If you're a Java programmer, you'll find that Scala's scoping rules are almost identical to Java's. One difference between Java and Scala is that Scala allows you to define variables of the same name in nested scopes. So if you're a Java programmer, you may wish to at least skim this section.

Variable declarations in Scala programs have a *scope* that defines where you can use the name. The most common example of scoping is that curly braces generally introduce a newscope, so anything defined inside curly braces leaves scope after the final closing brace.[4] As an illustration, consider the function shown in Listing 7.18.

```
def printMultiTable() = {

  var i = 1
  // only i in scope here

  while (i <= 10) {

    var j = 1
    // both i and j in scope here

    while (j <= 10) {

      val prod = (i * j).toString
      // i, j, and prod in scope here

      var k = prod.length
      // i, j, prod, and k in scope here

      while (k < 4) {
        print(" ")
        k += 1
      }

      print(prod)
      j += 1
    }

    // i and j still in scope; prod and k out of scope

    println()
    i += 1
  }

  // i still in scope; j, prod, and k out of scope
```

```
    }
```

Listing 7.18 - Variable scoping when printing a multiplication table.

The printMultiTable function shown in Listing 7.18 prints out a multiplication table.[5] The first statement of this function introduces a variable named i and initializes it to the integer 1. You can then use the name i for the remainder of the function.

The next statement in printMultiTable is a while loop:

```
  while (i <= 10) {

    var j = 1
    ...
  }
```

You can use i here because it is still in scope. In the first statement inside that while loop, you introduce another variable, this time named j, and again initialize it to 1. Because the variablej was defined inside the open curly brace of the while loop, it can be used only within thatwhile loop. If you were to attempt to do something with j after the closing curly brace of thiswhile loop, after the comment that says j, prod, and k are out of scope, your program would not compile.

All variables defined in this example—i, j, prod, and k—are *local variables*. Such variables are "local" to the function in which they are defined. Each time a function is invoked, a new set of its local variables is used.

Once a variable is defined, you can't define a new variable with the same name in the same scope. For example, the following script with two variables named a in the same scope would not compile:

```
  val a = 1
  val a = 2 // Does not compile
  println(a)
```

You can, on the other hand, define a variable in an inner scope that has the same name as a variable in an outer scope. The following script would compile and run:

```
  val a = 1;
  {
    val a = 2 // Compiles just fine
    println(a)
  }
  println(a)
```

When executed, the script shown previously would print 2 then 1, because the a defined inside the curly braces is a different variable, which is in scope only until the closing curly brace.[6] One difference to note between Scala and Java is that Java will not let you create a variable in an inner scope that has the same name as a variable in an outer scope. In a Scala program, an inner variable is said to *shadow* a like-named outer variable, because the outer variable becomes invisible in the inner scope.

You might have already noticed something that looks like shadowing in the interpreter:

```
scala> val a = 1
a: Int = 1

scala> val a = 2
a: Int = 2

scala> println(a)
2
```

In the interpreter, you can reuse variable names to your heart's content. Among other things, this allows you to change your mind if you made a mistake when you defined a variable the first time in the interpreter. You can do this because conceptually the interpreter creates a new nested scope for each new statement you type in. Thus, you could visualize the previous interpreted code like this:

```
val a = 1;
{
  val a = 2;
  {
    println(a)
  }
}
```

This code will compile and run as a Scala script, and like the code typed into the interpreter, will print 2. Keep in mind that such code can be very confusing to readers, because variable names adopt new meanings in nested scopes. It is usually better to choose a new, meaningful variable name rather than to shadow an outer variable.

7.8 REFACTORING IMPERATIVE-STYLE CODE

To help you gain insight into the functional style, in this section we'll refactor the imperative approach to printing a multiplication table shown in Listing 7.18. Our functional alternative is shown in Listing 7.19.

```
// Returns a row as a sequence
def makeRowSeq(row: Int) =
  for (col <- 1 to 10) yield {
    val prod = (row * col).toString
    val padding = " " * (4 - prod.length)
    padding + prod
  }

// Returns a row as a string
def makeRow(row: Int) = makeRowSeq(row).mkString

// Returns table as a string with one row per line
def multiTable() = {

  val tableSeq = // a sequence of row strings
    for (row <- 1 to 10)
    yield makeRow(row)

  tableSeq.mkString("\n")
}
```

Listing 7.19 - A functional way to create a multiplication table.

The imperative style reveals itself in Listing 7.18 in two ways. First, invoking printMultiTablehas a side effect: printing a multiplication table to the standard output. In Listing 7.19, we refactored the function so that it returns the multiplication table as a string. Since the function no longer prints, we renamed it multiTable. As mentioned previously, one advantage of side-effect-free functions is they are easier to unit test. To test printMultiTable, you would need to somehow redefine print and println so you could check the output for correctness. You could test multiTable more easily by checking its string result.

The other telltale sign of the imperative style in printMultiTable is its while loop and vars. By contrast, the multiTable function uses vals, for expressions, *helper functions*, and calls tomkString.

We factored out the two helper functions, makeRow and makeRowSeq, to make the code easier to read. Function makeRowSeq uses a for expression whose generator iterates through column numbers 1 through 10. The body of this for calculates the product of row and column, determines the padding needed for the product, and yields the result of concatenating the padding and product strings. The result of the for expression will be a sequence (some subclass of scala.Seq) containing these yielded strings as elements. The other helper function,makeRow, simply invokes mkString on the result returned by makeRowSeq. mkString will concatenate the strings in the sequence and return them as one string.

The multiTable method first initializes tableSeq with the result of a for expression whose generator iterates through row numbers 1 to 10, and for each calls makeRow to get the string for that row. This string is yielded; thus the result of this for expression will be a sequence of row strings. The only remaining task is to convert the sequence of strings into a single string. The call to mkString accomplishes this, and because we pass "\n", we get an end of line character inserted between each string. If you pass the string returned by multiTable to println, you'll see the same output that's produced by calling printMultiTable.

```
 1   2   3   4   5   6   7   8   9  10
 2   4   6   8  10  12  14  16  18  20
 3   6   9  12  15  18  21  24  27  30
 4   8  12  16  20  24  28  32  36  40
 5  10  15  20  25  30  35  40  45  50
 6  12  18  24  30  36  42  48  54  60
 7  14  21  28  35  42  49  56  63  70
 8  16  24  32  40  48  56  64  72  80
 9  18  27  36  45  54  63  72  81  90
10  20  30  40  50  60  70  80  90 100
```

7.9 CONCLUSION

Scala's built-in control structures are minimal, but they do the job. They act much like their imperative equivalents, but because they tend to result in a value, they support a functional style, too. Just as important, they are careful in what they omit, thus leaving room for one of Scala's most powerful features, the function literal, which will be described in the next chapter.

Footnotes for Chapter 7:

[1] The gcd function shown in Listing 7.4 uses the same approach used by the like-named function, first shown in Listing 6.3, to calculate greatest common divisors for class Rational. The main difference is that instead of Ints the gcd of Listing 7.4 works with Longs.

[2] To be precise, the expression to the right of the <- symbol in a for expression can be any type that has certain methods (in this case foreach) with appropriate signatures. Details on how the Scala compiler processes for expressions are described in Chapter 23.

[3] Although you must always surround the case statements of a catch clause in parentheses,try and finally do not require parentheses if they contain only one expression. For example, you could write: try t() catch { case e: Exception => ... } finally f().

[4] There are a few exceptions to this rule because in Scala you can sometimes use curly braces in place of parentheses. One example of this kind of curly-brace use is the alternativefor expression syntax described in Section 7.3.

[5] The printMultiTable function shown in Listing 7.18 is written in an imperative style. We'll refactor it into a functional style in the next section.

[6] By the way, the semicolon is required in this case after the first definition of a because Scala's semicolon inference mechanism will not place one there.