Chapter 9

Control Abstraction

In Chapter 7, we pointed out that Scala doesn't have many built-in control abstractions because it gives you the ability to create your own. In the previous chapter, you learned about function values. In this chapter, we'll show you how to apply function values to create new control abstractions. Along the way, you'll also learn about currying and by-name parameters.

9.1 REDUCING CODE DUPLICATION

All functions are separated into common parts, which are the same in every invocation of the function, and non-common parts, which may vary from one function invocation to the next. The common parts are in the body of the function, while the non-common parts must be supplied via arguments. When you use a function value as an argument, the non-common part of the algorithm is itself some other algorithm! At each invocation of such a function, you can pass in a different function value as an argument, and the invoked function will, at times of its choosing, invoke the passed function value. These *higher-order functions*—functions that take functions as parameters—give you extra opportunities to condense and simplify code.

One benefit of higher-order functions is they enable you to create control abstractions that allow you to reduce code duplication. For example, suppose you are writing a file browser, and you want to provide an API that allows users to search for files matching some criterion. First, you add a facility to search for files whose names end in a particular string. This would enable your users to find, for example, all files with a ".scala" extension. You could provide such an API by defining a public filesEnding method inside a singleton object like this:

```
object FileMatcher {
  private def filesHere = (new java.io.File(".")).listFiles

  def filesEnding(query: String) =
    for (file <- filesHere; if file.getName.endsWith(query))
      yield file
}
```

The filesEnding method obtains the list of all files in the current directory using the private helper method filesHere, then filters them based on whether each file name ends with the user-specified query. Given filesHere is private, the filesEnding method is the only accessible method defined in FileMatcher, the API you provide to your users.

So far so good, and there is no repeated code yet. Later on, though, you decide to let people search based on any part of the file name. This is good for when your users cannot remember if they named a file phb-important.doc, stupid-phb-report.doc, may2003salesdoc.phb, or something entirely different; they just know that "phb" appears in the name somewhere. You go back to work and add this function to your FileMatcher API:

```
def filesContaining(query: String) =
  for (file <- filesHere; if file.getName.contains(query))
    yield file
```

This function works just like filesEnding. It searches filesHere, checks the name, and returns the file if the name matches. The only difference is that this function uses contains instead ofendsWith.

The months go by, and the program becomes more successful. Eventually, you give in to the requests of a few power users who want to search based on regular expressions. These sloppy guys have immense directories with thousands of files, and they would like to do things like find all "pdf" files that have "oopsla" in the title somewhere. To support them, you write this function:

```
def filesRegex(query: String) =
  for (file <- filesHere; if file.getName.matches(query))
    yield file
```

Experienced programmers will notice all of this repetition and wonder if it can be factored into a common helper function. Doing it the obvious way does not work, however. You would like to be able to do the following:

```
def filesMatching(query: String, null) =
  for (file <- filesHere; if file.getName.null(query))
    yield file
```

This approach would work in some dynamic languages, but Scala does not allow pasting together code at runtime like this. So what do you do?

Function values provide an answer. While you cannot pass around a method name as a value, you can get the same effect by passing around a function value that calls the method for you. In this case, you could add a matcher parameter to the method whose sole purpose is to check a file name against a query:

```
def filesMatching(query: String,
    matcher: (String, String) => Boolean) = {

  for (file <- filesHere; if matcher(file.getName, query))
    yield file
}
```

In this version of the method, the if clause now uses matcher to check the file name against the query. Precisely what this check does depends on what is specified as the matcher. Take a look, now, at the type of matcher itself. It is a function, and thus has a => in the type. This function takes two string arguments—the file name and the query—and returns a boolean, so the type of this function is (String, String) => Boolean.

Given this new filesMatching helper method, you can simplify the three searching methods by having them call the helper method, passing in an appropriate function:

```
def filesEnding(query: String) =
  filesMatching(query, _.endsWith(_))
```

```
  def filesContaining(query: String) =
    filesMatching(query, _.contains(_))

  def filesRegex(query: String) =
    filesMatching(query, _.matches(_))
```

The function literals shown in this example use the placeholder syntax, introduced in the previous chapter, which may not as yet feel very natural to you. So here's a clarification of how placeholders are used: The function literal _.endsWith(_), used in the filesEnding method, means the same thing as:

```
(fileName: String, query: String) => fileName.endsWith(query)
```

Because filesMatching takes a function that requires two String arguments, you need not specify the types of the arguments; you could just write(fileName, query) => fileName.endsWith(query). Since the parameters are each used only once in the body of the function (*i.e.*, the first parameter, fileName, is used first in the body, and the second parameter, query, is used second), you can use the placeholder syntax: _.endsWith(_). The first underscore is a placeholder for the first parameter, the file name, and the second underscore a placeholder for the second parameter, the query string.

This code is already simplified, but it can actually be even shorter. Notice that the query gets passed to filesMatching, but filesMatching does nothing with the query except to pass it back to the passed matcher function. This passing back and forth is unnecessary because the caller already knew the query to begin with! You might as well remove the query parameter fromfilesMatching and matcher, thus simplifying the code as shown in Listing 9.1.

```
object FileMatcher {
  private def filesHere = (new java.io.File(".")).listFiles

  private def filesMatching(matcher: String => Boolean) =
    for (file <- filesHere; if matcher(file.getName))
      yield file

  def filesEnding(query: String) =
    filesMatching(_.endsWith(query))

  def filesContaining(query: String) =
    filesMatching(_.contains(query))

  def filesRegex(query: String) =
    filesMatching(_.matches(query))
}
```

Listing 9.1 - Using closures to reduce code duplication.

This example demonstrates the way in which first-class functions can help you eliminate code duplication where it would be very difficult to do so without them. In Java, for example, you could create an interface containing a method that takes one String and returns a Boolean, then create and pass anonymous inner class instances that implement this interface tofilesMatching. Although this approach would remove the code duplication you are trying to eliminate, it would, at the same time, add as much or more new code. Thus the benefit is not worth the cost, and you may as well live with the duplication.

Moreover, this example demonstrates how closures can help you reduce code duplication. The function literals used in the previous example, such as _.endsWith(_) and _.contains(_), are instantiated at runtime into function values that are *not* closures because they don't capture any free variables. Both variables used in the expression, _.endsWith(_), for example, are represented by underscores, which means they are taken from arguments to the function. Thus, _.endsWith(_) uses two bound variables, and no free variables. By contrast, the function literal _.endsWith(query), used in the most recent example, contains one bound variable, the argument represented by the underscore, and one free variable named query. It is only because Scala supports closures that you were able to remove the query parameter fromfilesMatching in the most recent example, thereby simplifying the code even further.

9.2 SIMPLIFYING CLIENT CODE

The previous example demonstrated that higher-order functions can help reduce code duplication as you implement an API. Another important use of higher-order functions is to put them in an API itself to make client code more concise. A good example is provided by the special-purpose looping methods of Scala's collection types.[1] Many of these are listed inTable 3.1 in Chapter 3, but take a look at just one example for now to see why these methods are so useful.

Consider exists, a method that determines whether a passed value is contained in a collection. You could, of course, search for an element by having a var initialized to false, looping through the collection checking each item, and setting the var to true if you find what you are looking for. Here's a method that uses this approach to determine whether a passedList contains a negative number:

```
def containsNeg(nums: List[Int]): Boolean = {
  var exists = false
  for (num <- nums)
    if (num < 0)
      exists = true
  exists
}
```

If you define this method in the interpreter, you can call it like this:

```
scala> containsNeg(List(1, 2, 3, 4))
res0: Boolean = false

scala> containsNeg(List(1, 2, -3, 4))
res1: Boolean = true
```

A more concise way to define the method, though, is by calling the higher-order functionexists on the passed List, like this:

```
def containsNeg(nums: List[Int]) = nums.exists(_ < 0)
```

This version of containsNeg yields the same results as the previous:

```
scala> containsNeg(Nil)
res2: Boolean = false
```

```
scala> containsNeg(List(0, -1, -2))
res3: Boolean = true
```

The exists method represents a control abstraction. It is a special-purpose looping construct provided by the Scala library, rather than built into the Scala language like while or for. In the previous section, the higher-order function, filesMatching, reduces code duplication in the implementation of the object FileMatcher. The exists method provides a similar benefit, but because exists is public in Scala's collections API, the code duplication it reduces is client code of that API. If exists didn't exist, and you wanted to write a containsOdd method to test whether a list contains odd numbers, you might write it like this:

```
def containsOdd(nums: List[Int]): Boolean = {
  var exists = false
  for (num <- nums)
    if (num % 2 == 1)
      exists = true
  exists
}
```

If you compare the body of containsNeg with that of containsOdd, you'll find that everything is repeated except the test condition of an if expression. Using exists, you could write this instead:

```
def containsOdd(nums: List[Int]) = nums.exists(_ % 2 == 1)
```

The body of the code in this version is again identical to the body of the correspondingcontainsNeg method (the version that uses exists), except the condition for which to search is different. Yet the amount of code duplication is much smaller because all of the looping infrastructure is factored out into the exists method itself.

There are many other looping methods in Scala's standard library. As with exists, they can often shorten your code if you recognize opportunities to use them.

9.3 CURRYING

In Chapter 1, we said that Scala allows you to create new control abstractions that "feel like native language support." Although the examples you've seen so far are indeed control abstractions, it is unlikely anyone would mistake them for native language support. To understand how to make control abstractions that feel more like language extensions, you first need to understand the functional programming technique called *currying*.

A curried function is applied to multiple argument lists, instead of just one. Listing 9.2 shows a regular, non-curried function, which adds two Int parameters, x and y.

```
scala> def plainOldSum(x: Int, y: Int) = x + y
plainOldSum: (x: Int, y: Int)Int

scala> plainOldSum(1, 2)
res4: Int = 3
```

Listing 9.2 - Defining and invoking a "plain old" function.

By contrast, Listing 9.3 shows a similar function that's curried. Instead of one list of two Intparameters, you apply this function to two lists of one Int parameter each.

```
scala> def curriedSum(x: Int)(y: Int) = x + y
curriedSum: (x: Int)(y: Int)Int

scala> curriedSum(1)(2)
res5: Int = 3
```

Listing 9.3 - Defining and invoking a curried function.

What's happening here is that when you invoke curriedSum, you actually get two traditional function invocations back to back. The first function invocation takes a single Int parameter named x, and returns a function value for the second function. This second function takes theInt parameter y. Here's a function named first that does in spirit what the first traditional function invocation of curriedSum would do:

```
scala> def first(x: Int) = (y: Int) => x + y
first: (x: Int)Int => Int
```

Applying the first function to 1—in other words, invoking the first function and passing in 1—yields the second function:

```
scala> val second = first(1)
second: Int => Int = <function1>
```

Applying the second function to 2 yields the result:

```
scala> second(2)
res6: Int = 3
```

These first and second functions are just an illustration of the currying process. They are not directly connected to the curriedSum function. Nevertheless, there is a way to get an actual reference to curriedSum's "second" function. You can use the placeholder notation to usecurriedSum in a partially applied function expression, like this:

```
scala> val onePlus = curriedSum(1)_
onePlus: Int => Int = <function1>
```

The underscore in curriedSum(1)_ is a placeholder for the second parameter list.[2] The result is a reference to a function that, when invoked, adds one to its sole Int argument and returns the result:

```
scala> onePlus(2)
res7: Int = 3
```

And here's how you'd get a function that adds two to its sole Int argument:

```
scala> val twoPlus = curriedSum(2)_
twoPlus: Int => Int = <function1>

scala> twoPlus(2)
res8: Int = 4
```

9.4 WRITING NEW CONTROL STRUCTURES

In languages with first-class functions, you can effectively make new control structures even though the syntax of the language is fixed. All you need to do is create methods that take functions as arguments.

For example, here is the "twice" control structure, which repeats an operation two times and returns the result:

```
scala> def twice(op: Double => Double, x: Double) = op(op(x))
twice: (op: Double => Double, x: Double)Double

scala> twice(_ + 1, 5)
res9: Double = 7.0
```

The type of op in this example is Double => Double, which means it is a function that takes oneDouble as an argument and returns another Double.

Any time you find a control pattern repeated in multiple parts of your code, you should think about implementing it as a new control structure. Earlier in the chapter you saw filesMatching, a very specialized control pattern. Consider now a more widely used coding pattern: open a resource, operate on it, and then close the resource. You can capture this in a control abstraction using a method like the following:

```
def withPrintWriter(file: File, op: PrintWriter => Unit) = {
  val writer = new PrintWriter(file)
  try {
    op(writer)
  } finally {
    writer.close()
  }
}
```

Given such a method, you can use it like this:

```
withPrintWriter(
  new File("date.txt"),
  writer => writer.println(new java.util.Date)
)
```

The advantage of using this method is that it's withPrintWriter, not user code, that assures the file is closed at the end. So it's impossible to forget to close the file. This technique is called the *loan pattern*, because a control-abstraction function, such as withPrintWriter, opens a resource and "loans" it to a function. For instance, withPrintWriter in the previous example loans a PrintWriter to the function, op. When the function completes, it signals that it no longer needs the "borrowed" resource. The resource is then closed in a finally block, to ensure it is indeed closed, regardless of whether the function completes by returning normally or throwing an exception.

One way in which you can make the client code look a bit more like a built-in control structure is to use curly braces instead of parentheses to surround the argument list. In any method invocation in Scala in which you're passing in exactly one argument, you can opt to use curly braces to surround the argument instead of parentheses.

For example, instead of:

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

You could write:

```
scala> println { "Hello, world!" }
Hello, world!
```

In the second example, you used curly braces instead of parentheses to surround the arguments to println. This curly braces technique will work, however, only if you're passing in one argument. Here's an attempt at violating that rule:

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

scala> g.substring { 7, 9 }
<console>:1: error: ';' expected but ',' found.
       g.substring { 7, 9 }
                     ^
```

Because you are attempting to pass in two arguments to substring, you get an error when you try to surround those arguments with curly braces. Instead, you'll need to use parentheses:

```
scala> g.substring(7, 9)
res12: String = wo
```

The purpose of this ability to substitute curly braces for parentheses for passing in one argument is to enable client programmers to write function literals between curly braces. This can make a method call feel more like a control abstraction. Take the withPrintWritermethod defined previously as an example. In its most recent form, withPrintWriter takes two arguments, so you can't use curly braces. Nevertheless, because the function passed towithPrintWriter is the last argument in the list, you can use currying to pull the first argument, the File, into a separate argument list. This will leave the function as the lone parameter of the second argument list. Listing 9.4 shows how you'd need to redefine withPrintWriter.

```
def withPrintWriter(file: File)(op: PrintWriter => Unit) = {
  val writer = new PrintWriter(file)
  try {
    op(writer)
  } finally {
    writer.close()
  }
}
```

Listing 9.4 - Using the loan pattern to write to a file.

The new version differs from the old one only in that there are now two parameter lists with one parameter each instead of one parameter list with two parameters. Look between the two parameters. In the previous version of withPrintWriter, shown here, you see ...File, op.... But in this version, you see ...File)(op.... Given the above definition, you can call the method with a more pleasing syntax:

```
val file = new File("date.txt")

withPrintWriter(file) { writer =>
  writer.println(new java.util.Date)
}
```

In this example, the first argument list, which contains one File argument, is written surrounded by parentheses. The second argument list, which contains one function argument, is surrounded by curly braces.

9.5 BY-NAME PARAMETERS

The withPrintWriter method shown in the previous section differs from built-in control structures of the language, such as if and while, in that the code between the curly braces takes an argument. The function passed to withPrintWriter requires one argument of typePrintWriter. This argument shows up as the "writer =>" in:

```
withPrintWriter(file) { writer =>
  writer.println(new java.util.Date)
}
```

But what if you want to implement something more like if or while, where there is no value to pass into the code between the curly braces? To help with such situations, Scala provides by-name parameters.

As a concrete example, suppose you want to implement an assertion construct called myAssert. [3] The myAssert function will take a function value as input and consult a flag to decide what to do. If the flag is set, myAssert will invoke the passed function and verify that it returns true. If the flag is turned off, myAssert will quietly do nothing at all.

Without using by-name parameters, you could write myAssert like this:

```
var assertionsEnabled = true

def myAssert(predicate: () => Boolean) =
  if (assertionsEnabled && !predicate())
    throw new AssertionError
```

The definition is fine, but using it is a little bit awkward:

```
myAssert(() => 5 > 3)
```

You would really prefer to leave out the empty parameter list and => symbol in the function literal and write the code like this:

```
myAssert(5 > 3) // Won't work, because missing () =>
```

By-name parameters exist precisely so that you can do this. To make a by-name parameter, you give the parameter a type starting with => instead of () =>. For example, you could changemyAssert's predicate parameter into a by-name parameter by changing its type, "() => Boolean", into "=> Boolean". Listing 9.5 shows how that would look:

```
  def byNameAssert(predicate: => Boolean) =
    if (assertionsEnabled && !predicate)
      throw new AssertionError
```

Listing 9.5 - Using a by-name parameter.

Now you can leave out the empty parameter in the property you want to assert. The result is that using byNameAssert looks exactly like using a built-in control structure:

```
byNameAssert(5 > 3)
```

A by-name type, in which the empty parameter list, (), is left out, is only allowed for parameters. There is no such thing as a by-name variable or a by-name field.

Now, you may be wondering why you couldn't simply write myAssert using a plain old Booleanfor the type of its parameter, like this:

```
def boolAssert(predicate: Boolean) =
  if (assertionsEnabled && !predicate)
    throw new AssertionError
```

This formulation is also legal, of course, and the code using this version of boolAssert would still look exactly as before:

```
boolAssert(5 > 3)
```

Nevertheless, one difference exists between these two approaches that is important to note. Because the type of boolAssert's parameter is Boolean, the expression inside the parentheses inboolAssert(5 > 3) is evaluated *before* the call to boolAssert. The expression 5 > 3 yields true, which is passed to boolAssert. By contrast, because the type of byNameAssert's predicate parameter is=> Boolean, the expression inside the parentheses in byNameAssert(5 > 3) is *not* evaluated before the call to byNameAssert. Instead a function value will be created whose apply method will evaluate 5 > 3, and this function value will be passed to byNameAssert.

The difference between the two approaches, therefore, is that if assertions are disabled, you'll see any side effects that the expression inside the parentheses may have in boolAssert, but not in byNameAssert. For example, if assertions are disabled, attempting to assert on "x / 0 == 0" will yield an exception in boolAssert's case:

```
scala> val x = 5
x: Int = 5

scala> var assertionsEnabled = false
assertionsEnabled: Boolean = false

scala> boolAssert(x / 0 == 0)
java.lang.ArithmeticException: / by zero
  ... 33 elided
```

But attempting to assert on the same code in byNameAssert's case will *not* yield an exception:

```
scala> byNameAssert(x / 0 == 0)
```

9.6 CONCLUSION

This chapter has shown you how to build on Scala's rich function support to build control abstractions. You can use functions within your code to factor out common control patterns, and you can take advantage of higher-order functions in the Scala library to reuse control patterns that are common across all programmers' code. We also discussed how to use currying and by-name parameters so that your own higher-order functions can be used with a concise syntax.

In the previous chapter and this one, you have seen quite a lot of information about functions. The next few chapters will go back to discussing more object-oriented features of the language.

Footnotes for Chapter 9:

[1] These special-purpose looping methods are defined in trait Traversable, which is extended by List, Set, and Map. See Chapter 17 for a discussion.

[2] In the previous chapter, when the placeholder notation was used on traditional methods, like println _, you had to leave a space between the name and the underscore. In this case you don't, because whereas println_ is a legal identifier in Scala, curriedSum(1)_ is not.

[3] You'll call this myAssert, not assert, because Scala provides an assert of its own, which will be described in Section 14.1.