Chapter 8

Functions and Closures

When programs get larger, you need some way to divide them into smaller, more manageable pieces. For dividing up control flow, Scala offers an approach familiar to all experienced programmers: divide the code into functions. In fact, Scala offers several ways to define functions that are not present in Java. Besides methods, which are functions that are members of some object, there are also functions nested within functions, function literals, and function values. This chapter takes you on a tour through all of these flavors of functions in Scala.

8.1 METHODS

The most common way to define a function is as a member of some object; such a function is called a method. As an example, Listing 8.1 shows two methods that together read a file with a given name and print out all lines whose length exceeds a given width. Every printed line is prefixed with the name of the file it appears in.

```
import scala.io.Source

object LongLines {

  def processFile(filename: String, width: Int) = {
    val source = Source.fromFile(filename)
    for (line <- source.getLines())
      processLine(filename, width, line)
  }

  private def processLine(filename: String,
      width: Int, line: String) = {

    if (line.length > width)
      println(filename + ": " + line.trim)
  }
}
```

Listing 8.1 - LongLines with a private processLine method.

The processFile method takes a filename and width as parameters. It creates a Source object from the file name and, in the generator of the for expression, calls getLines on the source. As mentioned in Step 12 of Chapter 3, getLines returns an iterator that provides one line from the file on each iteration, excluding the end-of-line character. The for expression processes each of these lines by calling the helper method, processLine. The processLine method takes three parameters: a filename, a width, and a line. It tests whether the length of the line is greater than the given width, and, if so, it prints the filename, a colon, and the line.

To use LongLines from the command line, we'll create an application that expects the line width as the first command-line argument, and interprets subsequent arguments as filenames:[1]

```
object FindLongLines {
  def main(args: Array[String]) = {
    val width = args(0).toInt
    for (arg <- args.drop(1))
      LongLines.processFile(arg, width)
  }
}
```

Here's how you'd use this application to find the lines in LongLines.scala that are over 45 characters in length (there's just one):

```
$ scala FindLongLines 45 LongLines.scala
LongLines.scala: def processFile(filename: String, width: Int) = {
```

So far, this is very similar to what you would do in any object-oriented language. However, the concept of a function in Scala is more general than a method. Scala's other ways to express functions will be explained in the following sections.

8.2 LOCAL FUNCTIONS

The construction of the processFile method in the previous section demonstrated an important design principle of the functional programming style: programs should be decomposed into many small functions that each do a well-defined task. Individual functions are often quite small. The advantage of this style is that it gives a programmer many building blocks that can be flexibly composed to do more difficult things. Each building block should be simple enough to be understood individually.

One problem with this approach is that all the helper function names can pollute the program namespace. In the interpreter this is not so much of a problem, but once functions are packaged in reusable classes and objects, it's desirable to hide the helper functions from clients of a class. They often do not make sense individually, and you often want to keep enough flexibility to delete the helper functions if you later rewrite the class a different way.

In Java, your main tool for this purpose is the private method. This private-method approach works in Scala as well, as demonstrated in Listing 8.1, but Scala offers an additional approach: you can define functions inside other functions. Just like local variables, such *local function*s are visible only in their enclosing block. Here's an example:

```
def processFile(filename: String, width: Int) = {

  def processLine(filename: String,
      width: Int, line: String) = {

    if (line.length > width)
      println(filename + ": " + line.trim)
  }

  val source = Source.fromFile(filename)
  for (line <- source.getLines()) {
    processLine(filename, width, line)
  }
}
```

In this example, we refactored the original LongLines version, shown in Listing 8.1, by transforming private method, processLine, into a local function of processFile. To do so we removed the private modifier, which can only be applied (and is only needed) for members, and placed the definition of processLine inside the definition of processFile. As a local function,processLine is in scope inside processFile, but inaccessible outside.

Now that processLine is defined inside processFile, however, another improvement becomes possible. Notice how filename and width are passed unchanged into the helper function? This is not necessary because local functions can access the parameters of their enclosing function. You can just use the parameters of the outer processLine function, as shown in Listing 8.2.

```
import scala.io.Source

object LongLines {

  def processFile(filename: String, width: Int) = {

    def processLine(line: String) = {
      if (line.length > width)
        println(filename + ": " + line.trim)
    }

    val source = Source.fromFile(filename)
    for (line <- source.getLines())
      processLine(line)
  }
}
```

Listing 8.2 - LongLines with a local processLine function.

Simpler, isn't it? This use of an enclosing function's parameters is a common and useful example of the general nesting Scala provides. The nesting and scoping described in Section 7.7 applies to all Scala constructs, including functions. It's a simple principle, but very powerful, especially in a language with first-class functions.

8.3 FIRST-CLASS FUNCTIONS

Scala has *first-class functions*. Not only can you define functions and call them, but you can write down functions as unnamed *literals* and then pass them around as *values*. We introduced function literals in Chapter 2 and showed the basic syntax in Figure 2.2 here.

A function literal is compiled into a class that when instantiated at runtime is a *function value*.[2] Thus the distinction between function literals and values is that function literals exist in the source code, whereas function values exist as objects at runtime. The distinction is much like that between classes (source code) and objects (runtime).

Here is a simple example of a function literal that adds one to a number:

```
(x: Int) => x + 1
```

The => designates that this function converts the thing on the left (any integer x) to the thing on the right (x + 1). So, this is a function mapping any integer x to x + 1.

Function values are objects, so you can store them in variables if you like. They are functions, too, so you can invoke them using the usual parentheses function-call notation. Here is an example of both activities:

```
scala> var increase = (x: Int) => x + 1
increase: Int => Int = <function1>

scala> increase(10)
res0: Int = 11
```

Because increase, in this example, is a var, you can assign a different function value to it later on.

```
scala> increase = (x: Int) => x + 9999
increase: Int => Int = <function1>

scala> increase(10)
res1: Int = 10009
```

If you want to have more than one statement in the function literal, surround its body by curly braces and put one statement per line, thus forming a block. Just like a method, when the function value is invoked, all of the statements will be executed, and the value returned from the function is whatever results from evaluating the last expression.

```
scala> increase = (x: Int) => {
         println("We")
         println("are")
         println("here!")
         x + 1
       }
increase: Int => Int = <function1>

scala> increase(10)
We
are
here!
res2: Int = 11
```

So now you have seen the nuts and bolts of function literals and function values. Many Scala libraries give you opportunities to use them. For example, a foreach method is available for all collections.[3] It takes a function as an argument and invokes that function on each of its elements. Here is how it can be used to print out all of the elements of a list:

```
scala> val someNumbers = List(-11, -10, -5, 0, 5, 10)
someNumbers: List[Int] = List(-11, -10, -5, 0, 5, 10)

scala> someNumbers.foreach((x: Int) => println(x))
-11
-10
-5
0
5
```

```
  10
```

As another example, collection types also have a filter method. This method selects those elements of a collection that pass a test the user supplies. That test is supplied using a function. For example, the function (x: Int) => x > 0 could be used for filtering. This function maps positive integers to true and all others to false. Here is how to use it with filter:

```
scala> someNumbers.filter((x: Int) => x > 0)
res4: List[Int] = List(5, 10)
```

Methods like foreach and filter are described further later in the book. Chapter 16 talks about their use in class List. Chapter 17 discusses their use with other collection types.

8.4 SHORT FORMS OF FUNCTION LITERALS

Scala provides a number of ways to leave out redundant information and write function literals more briefly. Keep your eyes open for these opportunities, because they allow you to remove clutter from your code.

One way to make a function literal more brief is to leave off the parameter types. Thus, the previous example with filter could be written like this:

```
scala> someNumbers.filter((x) => x > 0)
res5: List[Int] = List(5, 10)
```

The Scala compiler knows that x must be an integer, because it sees that you are immediately using the function to filter a list of integers (referred to by someNumbers). This is called *target typing* because the targeted usage of an expression (in this case, an argument tosomeNumbers.filter()) is allowed to influence the typing of that expression (in this case to determine the type of the x parameter). The precise details of target typing are not important. You can simply start by writing a function literal without the argument type, and if the compiler gets confused, add in the type. Over time you'll get a feel for which situations the compiler can and cannot puzzle out.

A second way to remove useless characters is to leave out parentheses around a parameter whose type is inferred. In the previous example, the parentheses around x are unnecessary:

```
scala> someNumbers.filter(x => x > 0)
res6: List[Int] = List(5, 10)
```

8.5 PLACEHOLDER SYNTAX

To make a function literal even more concise, you can use underscores as placeholders for one or more parameters, so long as each parameter appears only one time within the function literal. For example, _ > 0 is very short notation for a function that checks whether a value is greater than zero:

```
scala> someNumbers.filter(_ > 0)
res7: List[Int] = List(5, 10)
```

You can think of the underscore as a "blank" in the expression that needs to be "filled in." This blank will be filled in with an argument to the function each time the function is invoked. For example, given that someNumbers was initialized here to the valueList(-11, -10, -5, 0, 5, 10), the filter method will replace the blank in _ > 0 first with -11, as in-11 > 0, then with -10, as in -10 > 0, then with -5, as in -5 > 0, and so on to the end of the List. The function literal _ > 0, therefore, is equivalent to the slightly more verbose x => x > 0, as demonstrated here:

```
scala> someNumbers.filter(x => x > 0)
res8: List[Int] = List(5, 10)
```

Sometimes when you use underscores as placeholders for parameters, the compiler might not have enough information to infer missing parameter types. For example, suppose you write_ + _ by itself:

```
scala> val f = _ + _
<console>:7: error: missing parameter type for expanded
function ((x$1, x$2) => x$1.$plus(x$2))
       val f = _ + _
               ^
```

In such cases, you can specify the types using a colon, like this:

```
scala> val f = (_: Int) + (_: Int)
f: (Int, Int) => Int = <function2>

scala> f(5, 10)
res9: Int = 15
```

Note that _ + _ expands into a literal for a function that takes two parameters. This is why you can use this short form only if each parameter appears in the function literal exactly once. Multiple underscores mean multiple parameters, not reuse of a single parameter repeatedly. The first underscore represents the first parameter, the second underscore the second parameter, the third underscore the third parameter, and so on.

8.6 PARTIALLY APPLIED FUNCTIONS

Although the previous examples substitute underscores in place of individual parameters, you can also replace an entire parameter list with an underscore. For example, rather than writing println(_), you could write println _. Here's an example:

```
someNumbers.foreach(println _)
```

Scala treats this short form exactly as if you had written the following:

```
someNumbers.foreach(x => println(x))
```

Thus, the underscore in this case is not a placeholder for a single parameter. It is a placeholder for an entire parameter list. Remember that you need to leave a space between the function name and the underscore; otherwise, the compiler will think you are referring to a different symbol, such as, for example, a method named println_, which likely does not exist.

When you use an underscore in this way, you are writing a *partially applied function*. In Scala, when you invoke a function, passing in any needed arguments, you *apply* that function*to* the arguments. For example, given the following function:

```
scala> def sum(a: Int, b: Int, c: Int) = a + b + c
sum: (a: Int, b: Int, c: Int)Int
```

You could apply the function sum to the arguments 1, 2, and 3 like this:

```
scala> sum(1, 2, 3)
res10: Int = 6
```

A partially applied function is an expression in which you don't supply all of the arguments needed by the function. Instead, you supply some, or none, of the needed arguments. For example, to create a partially applied function expression involving sum, in which you supply none of the three required arguments, you just place an underscore after "sum". The resulting function can then be stored in a variable. Here's an example:

```
scala> val a = sum _
a: (Int, Int, Int) => Int = <function3>
```

Given this code, the Scala compiler instantiates a function value that takes the three integer parameters missing from the partially applied function expression, sum _, and assigns a reference to that new function value to the variable a. When you apply three arguments to this new function value, it will turn around and invoke sum, passing in those same three arguments:

```
scala> a(1, 2, 3)
res11: Int = 6
```

Here's what just happened: The variable named a refers to a function value object. This function value is an instance of a class generated automatically by the Scala compiler from sum _, the partially applied function expression. The class generated by the compiler has an applymethod that takes three arguments.[4] The generated class's apply method takes three arguments because three is the number of arguments missing in the sum _ expression. The Scala compiler translates the expression a(1, 2, 3) into an invocation of the function value'sapply method, passing in the three arguments 1, 2, and 3.
Thus, a(1, 2, 3) is a short form for:

```
scala> a.apply(1, 2, 3)
res12: Int = 6
```

This apply method, defined in the class generated automatically by the Scala compiler from the expression sum _, simply forwards those three missing parameters to sum, and returns the result. In this case apply invokes sum(1, 2, 3), and returns what sum returns, which is 6.

Another way to think about this kind of expression, in which an underscore is used to represent an entire parameter list, is as a way to transform a def into a function value. For example, if you have a local function, such as sum(a: Int, b: Int, c: Int): Int, you can "wrap" it in a function value whose apply method has the same parameter list and result types. When you apply this function value to some arguments, it in turn applies sum to those same arguments and returns the result. Although you can't assign a method or nested function to a variable, or pass it as an argument to another function, you can do these things if you wrap the method or nested function in a function value by placing an underscore after its name.

Now, although sum _ is indeed a partially applied function, it may not be obvious to you why it is called this. It has this name because you are not applying that function to all of its arguments. In the case of sum _, you are applying it to *none* of its arguments. But you can also express a partially applied function by supplying only *some* of the required arguments. Here's an example:

```
scala> val b = sum(1, _: Int, 3)
b: Int => Int = <function1>
```

In this case, you've supplied the first and last argument to sum, but not the middle argument. Since only one argument is missing, the Scala compiler generates a new function class whoseapply method takes one argument. When invoked with that one argument, this generated function's apply method invokes sum, passing in 1, the argument passed to the function, and 3. Here are some examples:

```
scala> b(2)
res13: Int = 6
```

In this case, b.apply invoked sum(1, 2, 3).

```
scala> b(5)
res14: Int = 9
```

And in this case, b.apply invoked sum(1, 5, 3).

If you are writing a partially applied function expression in which you leave off all parameters, such as println _ or sum _, you can express it more concisely by leaving off the underscore if a function is required at that point in the code. For example, instead of printing out each of the numbers in someNumbers (defined here) like this:

```
someNumbers.foreach(println _)
```

You could just write:

```
someNumbers.foreach(println)
```

This last form is allowed only in places where a function is required, such as the invocation offoreach in this example. The compiler knows a function is required in this case, becauseforeach requires that a function be passed as an argument. In situations where a function is not required, attempting to use this form will cause a compilation error. Here's an example:

```
scala> val c = sum
<console>:8: error: missing arguments for method sum;
follow this method with `_' if you want to treat it as a
partially applied function
       val c = sum
               ^
scala> val d = sum _
```

```
d: (Int, Int, Int) => Int = <function3>

scala> d(10, 20, 30)
res14: Int = 60
```

8.7 CLOSURES

So far in this chapter, all the examples of function literals have referred only to passed parameters. For example, in (x: Int) => x > 0, the only variable used in the function body, x > 0, is x, which is defined as a parameter to the function. You can, however, refer to variables defined elsewhere:

```
(x: Int) => x + more  // how much more?
```

This function adds "more" to its argument, but what is more? From the point of view of this function, more is a *free variable* because the function literal does not itself give a meaning to it. The x variable, by contrast, is a *bound variable* because it does have a meaning in the context of the function: it is defined as the function's lone parameter, an Int. If you try using this function literal by itself, without any more defined in its scope, the compiler will complain:

```
scala> (x: Int) => x + more
<console>:8: error: not found: value more
              (x: Int) => x + more
                              ^
```

WHY THE TRAILING UNDERSCORE?

Scala's syntax for partially applied functions highlights a difference in the design trade-offs of Scala and classical functional languages, such as Haskell or ML. In these languages, partially applied functions are considered the normal case. Furthermore, these languages have a fairly strict static type system that will usually highlight every error with partial applications that you can make. Scala bears a much closer relation to imperative languages, such as Java, where a method that's not applied to all its arguments is considered an error. Furthermore, the object-oriented tradition of subtyping and a universal root type accepts some programs that would be considered erroneous in classical functional languages.

For instance, say you mistook the drop(n: Int) method of List for tail(), and therefore forgot you need to pass a number to drop. You might write, "println(drop)". Had Scala adopted the classical functional tradition that partially applied functions are OK everywhere, this code would type check. However, you might be surprised to find out that the output printed by this println statement would always be <function>! What would have happened is that the expression drop would have been treated as a function object. Because println takes objects of any type, this would have compiled OK, but it would have given an unexpected result.

To avoid situations like this, Scala normally requires you to specify function arguments that are left out explicitly, even if the indication is as simple as a `_'. Scala allows you to leave off even the _ only when a function type is expected.

On the other hand, the same function literal will work fine so long as there is something available named more:

```
scala> var more = 1
more: Int = 1

scala> val addMore = (x: Int) => x + more
addMore: Int => Int = <function1>

scala> addMore(10)
res16: Int = 11
```

The function value (the object) that's created at runtime from this function literal is called a*closure*. The name arises from the act of "closing" the function literal by "capturing" the bindings of its free variables. A function literal with no free variables, such as (x: Int) => x + 1, is called a *closed term*, where a *term* is a bit of source code. Thus a function value created at runtime from this function literal is not a closure in the strictest sense, because (x: Int) => x + 1is already closed as written. But any function literal with free variables, such as(x: Int) => x + more, is an *open term*. Therefore, any function value created at runtime from(x: Int) => x + more will, by definition, require that a binding for its free variable, more, be captured. The resulting function value, which will contain a reference to the captured morevariable, is called a closure because the function value is the end product of the act of closing the open term, (x: Int) => x + more.

This example brings up a question: What happens if more changes after the closure is created? In Scala, the answer is that the closure sees the change. For example:

```
scala> more = 9999
more: Int = 9999

scala> addMore(10)
res17: Int = 10009
```

Intuitively, Scala's closures capture variables themselves, not the value to which variables refer.[5] As the previous example shows, the closure created for (x: Int) => x + more sees the change to more made outside the closure. The same is true in the opposite direction. Changes made by a closure to a captured variable are visible outside the closure. Here's an example:

```
scala> val someNumbers = List(-11, -10, -5, 0, 5, 10)
someNumbers: List[Int] = List(-11, -10, -5, 0, 5, 10)

scala> var sum = 0
sum: Int = 0

scala> someNumbers.foreach(sum +=  _)

scala> sum
res19: Int = -11
```

This example uses a roundabout way to sum the numbers in a List. Variable sum is in a surrounding scope from the function literal sum += _, which adds numbers to sum. Even though it is the closure modifying sum at runtime, the resulting total, -11, is still visible outside the closure.

What if a closure accesses some variable that has several different copies as the program runs? For example, what if a closure uses a local variable of some function, and the function is invoked many times? Which instance of that variable gets used at each access?

Only one answer is consistent with the rest of the language: the instance used is the one that was active at the time the closure was created. For example, here is a function that creates and returns "increase" closures:

```
def makeIncreaser(more: Int) = (x: Int) => x + more
```

Each time this function is called it will create a new closure. Each closure will access the morevariable that was active when the closure was created.

```
scala> val inc1 = makeIncreaser(1)
inc1: Int => Int = <function1>

scala> val inc9999 = makeIncreaser(9999)
inc9999: Int => Int = <function1>
```

When you call makeIncreaser(1), a closure is created and returned that captures the value 1 as the binding for more. Similarly, when you call makeIncreaser(9999), a closure that captures the value 9999 for more is returned. When you apply these closures to arguments (in this case, there's just one argument, x, which must be passed in), the result that comes back depends on how more was defined when the closure was created:

```
scala> inc1(10)
res20: Int = 11

scala> inc9999(10)
res21: Int = 10009
```

It makes no difference that the more in this case is a parameter to a method call that has already returned. The Scala compiler rearranges things in cases like these so that the captured parameter lives out on the heap, instead of the stack, and thus can outlive themethod call that created it. This rearrangement is all taken care of automatically, so you don't have to worry about it. Capture any variable you like: val, var, or parameter.

8.8 SPECIAL FUNCTION CALL FORMS

Most functions and function calls you encounter will be as you have seen so far in this chapter. The function will have a fixed number of parameters, the call will have an equal number of arguments, and the arguments will be specified in the same order and number as the parameters.

Since function calls are so central to programming in Scala, however, a few special forms of function definitions and function calls have been added to the language to address some special needs. Scala supports repeated parameters, named arguments, and default arguments.

Repeated parameters

Scala allows you to indicate that the last parameter to a function may be repeated. This allows clients to pass variable length argument lists to the function. To denote a repeated parameter, place an asterisk after the type of the parameter. For example:

```
scala> def echo(args: String*) =
         for (arg <- args) println(arg)
echo: (args: String*)Unit
```

Defined this way, echo can be called with zero to many String arguments:

```
scala> echo()

scala> echo("one")
one

scala> echo("hello", "world!")
hello
world!
```

Inside the function, the type of the repeated parameter is an Array of the declared type of the parameter. Thus, the type of args inside the echo function, which is declared as type "String*" is actually Array[String]. Nevertheless, if you have an array of the appropriate type, and you attempt to pass it as a repeated parameter, you'll get a compiler error:

```
scala> val arr = Array("What's", "up", "doc?")
arr: Array[String] = Array(What's, up, doc?)

scala> echo(arr)
<console>:10: error: type mismatch;
 found   : Array[String]
 required: String
              echo(arr)
                   ^
```

To accomplish this, you'll need to append the array argument with a colon and an _* symbol,like this:

```
scala> echo(arr: _*)
What's
up
doc?
```

This notation tells the compiler to pass each element of arr as its own argument to echo, rather than all of it as a single argument.

Named arguments

In a normal function call, the arguments in the call are matched one by one in the order of the parameters of the called function:

```
scala> def speed(distance: Float, time: Float): Float =
         distance / time
speed: (distance: Float, time: Float)Float

scala> speed(100, 10)
res27: Float = 10.0
```

In this call, the 100 is matched to distance and the 10 to time. The 100 and 10 are matched in the same order as the formal parameters are listed.

Named arguments allow you to pass arguments to a function in a different order. The syntax is simply that each argument is preceded by a parameter name and an equals sign. For example, the following call to speed is equivalent to speed(100,10):

```
scala> speed(distance = 100, time = 10)
res28: Float = 10.0
```

Called with named arguments, the arguments can be reversed without changing the meaning:

```
scala> speed(time = 10, distance = 100)
res29: Float = 10.0
```

It is also possible to mix positional and named arguments. In that case, the positional arguments come first. Named arguments are most frequently used in combination with default parameter values.

Default parameter values

Scala lets you specify default values for function parameters. The argument for such a parameter can optionally be omitted from a function call, in which case the corresponding argument will be filled in with the default.

An example is shown in Listing 8.3. Function printTime has one parameter, out, and it has a default value of Console.out.

```
  def printTime(out: java.io.PrintStream = Console.out) =
    out.println("time = " + System.currentTimeMillis())
```

Listing 8.3 - A parameter with a default value.

If you call the function as printTime(), thus specifying no argument to be used for out, then outwill be set to its default value of Console.out. You could also call the function with an explicit output stream. For example, you could send logging to the standard error output by calling the function as printTime(Console.err).

Default parameters are especially helpful when used in combination with named parameters. In Listing 8.4, function printTime2 has two optional parameters. The out parameter has a default of Console.out, and the divisor parameter has a default value of 1.

```
def printTime2(out: java.io.PrintStream = Console.out,
               divisor: Int = 1) =
  out.println("time = " + System.currentTimeMillis()/divisor)
```

Listing 8.4 - A function with two parameters that have defaults.

Function printTime2 can be called as printTime2() to have both parameters filled in with their default values. Using named arguments, however, either one of the parameters can be specified while leaving the other as the default. To specify the output stream, call it like this:

```
printTime2(out = Console.err)
```

To specify the time divisor, call it like this:

```
printTime2(divisor = 1000)
```

8.9 TAIL RECURSION

In Section 7.2, we mentioned that to transform a while loop that updates vars into a more functional style that uses only vals, you may sometimes need to use recursion. Here's an example of a recursive function that approximates a value by repeatedly improving a guess until it is good enough:

```
def approximate(guess: Double): Double =
  if (isGoodEnough(guess)) guess
  else approximate(improve(guess))
```

A function like this is often used in search problems, with appropriate implementations forisGoodEnough and improve. If you want the approximate function to run faster, you might be tempted to write it with a while loop to try and speed it up, like this:

```
def approximateLoop(initialGuess: Double): Double = {
  var guess = initialGuess
  while (!isGoodEnough(guess))
    guess = improve(guess)
  guess
}
```

Which of the two versions of approximate is preferable? In terms of brevity and var avoidance, the first, functional one wins. But is the imperative approach perhaps more efficient? In fact, if we measure execution times, it turns out that they are almost exactly the same!

This might seem surprising because a recursive call looks much more "expansive" than a simple jump from the end of a loop to its beginning. However, in the case of approximate above, the Scala compiler is able to apply an important optimization. Note that the recursive call is the last thing that happens in the evaluation of function approximate's body. Functions likeapproximate, which call themselves as their last action, are called *tail recursive*. The Scala compiler detects tail recursion and replaces it with a jump back to the beginning of the function, after updating the function parameters with the new values.

The moral is that you should not shy away from using recursive algorithms to solve your problem. Often, a recursive solution is more elegant and concise than a loop-based one. If the solution is tail recursive, there won't be any runtime overhead to be paid.

Tracing tail-recursive functions

A tail-recursive function will not build a new stack frame for each call; all calls will execute in a single frame. This may surprise a programmer inspecting a stack trace of a program that failed. For example, this function calls itself some number of times then throws an exception:

```
def boom(x: Int): Int =
  if (x == 0) throw new Exception("boom!")
  else boom(x - 1) + 1
```

This function is not tail recursive, because it performs an increment operation after the recursive call. You'll get what you expect when you run it:

```
scala>  boom(3)
java.lang.Exception: boom!
      at .boom(<console>:5)
      at .boom(<console>:6)
      at .boom(<console>:6)
      at .boom(<console>:6)
      at .<init>(<console>:6)
...
```

TAIL CALL OPTIMIZATION

The compiled code for approximate is essentially the same as the compiled code forapproximateLoop. Both functions compile down to the same thirteen instructions of Java bytecodes. If you look through the bytecodes generated by the Scala compiler for the tail recursive method, approximate, you'll see that although both isGoodEnough and improveare invoked in the body of the method, approximate is not. The Scala compiler optimized away the recursive call:

```
public double approximate(double);
  Code:
   0:   aload_0
   1:   astore_3
   2:   aload_0
   3:   dload_1
   4:   invokevirtual   #24; //Method isGoodEnough:(D)Z
   7:   ifeq    12
   10:  dload_1
   11:  dreturn
   12:  aload_0
   13:  dload_1
   14:  invokevirtual   #27; //Method improve:(D)D
   17:  dstore_1
   18:  goto    2
```

If you now modify boom so that it does become tail recursive:

```
def bang(x: Int): Int =
    if (x == 0) throw new Exception("bang!")
    else bang(x - 1)
```

You'll get:

```
scala> bang(5)
java.lang.Exception: bang!
      at .bang(<console>:5)
      at .<init>(<console>:6) ...
```

This time, you see only a single stack frame for bang. You might think that bang crashed before it called itself, but this is not the case. If you think you might be confused by tail-call optimizations when looking at a stack trace, you can turn them off by giving the following argument to the scala shell or to the scalac compiler:

```
-g:notailcalls
```

With that option specified, you will get a longer stack trace:

```
scala> bang(5)
java.lang.Exception: bang!
      at .bang(<console>:5)
      at .bang(<console>:5)
      at .bang(<console>:5)
      at .bang(<console>:5)
      at .bang(<console>:5)
      at .bang(<console>:5)
      at .<init>(<console>:6) ...
```

Limits of tail recursion

The use of tail recursion in Scala is fairly limited because the JVM instruction set makes implementing more advanced forms of tail recursion very difficult. Scala only optimizes directly recursive calls back to the same function making the call. If the recursion is indirect, as in the following example of two mutually recursive functions, no optimization is possible:

```
def isEven(x: Int): Boolean =
  if (x == 0) true else isOdd(x - 1)
def isOdd(x: Int): Boolean =
  if (x == 0) false else isEven(x - 1)
```

You also won't get a tail-call optimization if the final call goes to a function value. Consider for instance the following recursive code:

```
val funValue = nestedFun _
def nestedFun(x: Int) : Unit = {
  if (x != 0) { println(x); funValue(x - 1) }
}
```

The funValue variable refers to a function value that essentially wraps a call to nestedFun. When you apply the function value to an argument, it turns around and applies nestedFun to that same argument, and returns the result. Therefore, you might hope the Scala compiler would perform a tail-call optimization, but in this case it would not. Tail-call optimization is limited to situations where a method or nested function calls itself directly as its last operation, without going through a function value or some other intermediary. (If you don't fully understand tail recursion yet, see Section 8.9).

8.10 CONCLUSION

This chapter has given you a grand tour of functions in Scala. In addition to methods, Scala provides local functions, function literals, and function values. In addition to normal function calls, Scala provides partially applied functions and functions with repeated parameters. When possible, function calls are implemented as optimized tail calls, and thus many nice-looking recursive functions run just as quickly as hand-optimized versions that use whileloops. The next chapter will build on these foundations and show how Scala's rich support for functions helps you abstract over control.

Footnotes for Chapter 8:

[1] In this book, we usually won't check command-line arguments for validity in example applications, both to save trees and reduce boilerplate code that can obscure the example's important code. The trade-off is that instead of producing a helpful error message when given bad input, our example applications will throw an exception.

[2] Every function value is an instance of some class that extends one of several FunctionNtraits in package scala, such as Function0 for functions with no parameters, Function1 for functions with one parameter, and so on. Each FunctionN trait has an apply method used to invoke the function.

[3] A foreach method is defined in trait Traversable, a common supertrait of List, Set, Array, andMap. See Chapter 17 for the details.

[4] The generated class extends trait Function3, which declares a three-arg apply method.

[5] By contrast, Java's inner classes do not allow you to access modifiable variables in surrounding scopes at all, so there is no difference between capturing a variable and capturing its currently held value.