Extracted from:

Pragmatic Scala

Create Expressive, Concise, and Scalable Applications

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Scala 2.11

Pragmatic Scala

Create Expressive, Concise, and Scalable Applications

Venkat Subramaniam

edited by Jacquelyn Carter

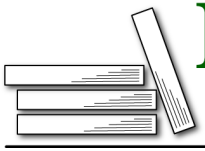
Pragmatic Scala

Create Expressive, Concise, and Scalable Applications

Venkat Subramaniam

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *https://pragprog.com*.

The team that produced this book includes:

Jacquelyn Carter (editor)
Potomac Indexing, LLC (index)
Liz Welch (copyedit)
Dave Thomas (layout)
Janet Furlow (producer)
Ellie Callahan (support)

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-054-7
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—September 2015

CHAPTER 6

Function Values and Closures

Functions are first-class citizens in functional programming. You can pass functions to functions as parameters, return them from functions, and even nest functions within functions. These higher-order functions are called *function values* in Scala. Closures are special forms of function values that close over or bind to variables defined in another scope or context.

In addition to object decomposition, you can structure applications around function values as building blocks, since Scala supports both the OO and functional styles of programming. This can lead to concise, reusable code. In this chapter, you'll learn to use function values and closures in Scala.

Limitations of Normal Functions

At the core of functional programming are functions or so-called higher-order functions. To get a feel for what these are, let's start with a familiar function.

To find the sum of values in a given range `1` to `number` we'd probably write code like this:

```
def sum(number: Int) = {
  var result = 0
  for(i <- 1 to number) {
    result += i
  }
  result
}
```

That's familiar code—we've all written code like this a million times over in different languages. That's called imperative style—you tell not only what to do, but also how to do it. That's dictating a low level of details. In Scala, you can write imperative code like this where it make sense, but you're not restricted to that.

While this code got the work done, it's not extensible. Now, if in addition, we need to count the number of even numbers and the number of odd numbers in the given range, that code will fall flat; we'd be tempted to use the infamous reuse by copy-paste-and-change pattern. Cringe! That's the best we can do with normal functions, but that'd lead to code duplication with poor reuse.

Let's take another shot at the simple problem on hand. Instead of the imperative style, we can program the same problem in functional style. We can pass an anonymous function to the function that iterates over the range. In other words, we make use of a level of indirection. The functions we pass can hold different logic to achieve different tasks over the iteration. Let's rewrite the previous code in functional style.

Extensibility with Higher-Order Functions

Functions that can take other functions as parameters are called *higher-order functions*. They reduce code duplication, increase reuse, and make code concise as well. We can create functions within functions, assign them to references, and pass them around to other functions. Scala internally deals with these so-called function values by creating them as instances of special classes. In Scala, function values are really objects.

Let's rewrite the previous example using function values. With this new version, we can perform different operations, such as summing numbers or counting the number of even numbers on a range of values.

We'll start by first extracting the common code into a method named totalResultOverRange(), by looping over the range of values:

```
def totalResultOverRange(number: Int, codeBlock: Int => Int) = {
  var result = 0
  for (i <- 1 to number) {
    result += codeBlock(i)
  }
  result
}
```

We've defined two parameters for the method totalResultOverRange(). The first one is an Int for the range of values to iterate over. The second one is special; it's a function value. The name of the parameter is codeBlock, and its type is a function that accepts an Int and returns an Int. The result of the method totalResultOverRange() is itself an Int.

The symbol => specifies and separates the function's expected input to the left and the expected return type to the right. The syntax form input => output

is intended to help us think of a function as transforming input to output without having any side effects.

In the body of the totalResultOverRange() method we iterate over the range of values, and for each element we invoke the given function, referenced by the variable codeBlock. The given function is expected to receive an Int, representing an element in the range, and return an Int as a result of computation on that element. The computation or operation itself is left to be defined by the caller of the method totalResultOverRange(). We total the results of calls to the given function value and return that total.

The code in totalResultOverRange() removed the duplication from the example in *Limitations of Normal Functions*, on page 5. Here is how we'd call that method to get the sum of values in the range:

```
println(totalResultOverRange(11, i => i))
```

We passed two arguments to the method. The first argument is the upper limit (11) of the range we want to iterate over. The second argument is actually an anonymous just-in-time function—that is, a function with no name but only parameter(s) and an implementation. The implementation, in this example, simply returns the given parameter. The symbol =>, in this case, separates the parameter list on the left from the implementation on the right. Scala was able to infer that the type of the parameter, i, is an Int from the parameter list of totalResultOverRange(). Scala will give us an error if the parameter's type or the result type does not match with what's expected.

For a simple totaling of the values, it may appear that the call to totalResultOverRange() with a number and a function as arguments was rather cumbersome, compared to the call to the normal function sum() we wrote earlier. However, the new version is extensible and we can call it in a similar way for other operations. For example, instead of finding the sum, if we'd like to total the even numbers in the range, we'd call the function like this:

```
println(totalResultOverRange(11, i => if (i % 2 == 0) i else 0))
```

The function value we pass as argument in this case returns the input parameter if it is even; otherwise it returns a 0. Thus, the function totalResultOverRange() will only total even values in the given range.

If we'd like to total the odd numbers, we can call the function as follows:

```
println(totalResultOverRange(11, i => if (i % 2 != 0) i else 0))
```

Unlike the sum() function, we saw how the totalResultOverRange() function can be extended to perform a total for a different selection of values in the range.

This is a direct benefit of the indirection we achieved using higher-order functions.

Any number of parameters of a function or a method can be function values, and they can be any parameter, not just the trailing parameter.

It was quite easy to make the code DRY (see *The Pragmatic Programmer: From Journeyman to Master [HT00]* by Andy Hunt and David Thomas for details about the Don't Repeat Yourself [DRY] principle) using function values. We gathered up the common code into a function, and the differences were rolled into arguments of method calls. Functions and methods that accept function values are commonplace in the Scala library, as we'll see in Chapter 8, *Collections*, on page ?. Scala makes it easy to pass multiple parameters to function values, and define the types of arguments as well, if you desire.

Function Values with Multiple Parameters

In the previous example, the function values received one parameter. Function values can receive zero or more parameters. Let's take a look at a few examples to learn how to define function values with different numbers of parameters.

In its simplest form, a function value may not even receive any parameter, but may return a result. Such a function value is like a factory—it constructs and returns an object. Let's take a look at an example of defining and using a zero parameter function value:

```
def printValue(generator: () => Int) = {
  println(s"Generated value is ${generator()}")
}

printValue(() => 42)
```

For the method printValue() function, we've defined the parameter generator as a function value that takes no parameters, indicated by an empty set of parentheses, and returns an Int. Within the function, we call the function value like we call any function, like so: generator(). In the call to the printValue() function, we've created a function value that takes no parameters and returns a fixed value, 42. Instead of returning a fixed value, the function value may return a random value, a new value, or a pre-cached value.

You know how to pass zero or one arguments. To pass more than one argument you need to provide a comma-separated list of parameter types in the definition. Let's look at an example of a function inject() that passes the result of the operation on one element in an array of Int to the operation on the next

element. It's a way to cascade or accumulate results from operations on each element.

```
def inject(arr: Array[Int], initial: Int, operation: (Int, Int) => Int) = {
  var carryOver = initial
  arr.foreach(element => carryOver = operation(carryOver, element) )
  carryOver
}
```

The inject() method takes three parameters: an array of Int, an initial Int value to inject into the operation, and the operation itself as a function value. In the method we set a variable carryOver to the initial value. We loop through the elements of the given array using the foreach() method. This method accepts a function value as a parameter, which it invokes with each element in the array as an argument. In the function that we send as an argument to foreach(), we're invoking the given operation with two arguments: the carryOver value and the context element. We assign the result of the operation call to the variable carryOver so it can be passed as an argument in the subsequent call to the operation. When we're done calling the operation for each element in the array, we return the final value of carryOver.

Let's look at a couple of examples of using the inject() method. Here's how we would total the elements in the array:

```
val array = Array(2, 3, 5, 1, 6, 4)
val sum = inject(array, 0, (carry, elem) => carry + elem)
println(s"Sum of elements in array is $sum")
```

The first argument to the method inject() is an array whose elements we'd like to sum. The second argument is an initial value 0 for the sum. The third argument is the function that carries out the operation of totaling the elements, one at a time. If instead of totaling the elements we'd like to find the maximum value, we can use the same inject() method:

```
val max =
  inject(array, Integer.MIN_VALUE, (carry, elem) => Math.max(carry, elem))
println(s"Max of elements in array is  $max")
```

The function value passed as an argument to the inject() function in the second call returns the maximum of the two parameters passed to it.

Here's the output of executing the previous two calls to the inject() method:

```
Sum of elements in array is 21
Max of elements in array is  6
```

The previous example helped us see how to pass multiple parameters. However, to navigate over elements in a collection and perform operations, we

don't have to really roll out our own inject() method. The Scala library already has this method built in. It is the foldLeft() method. Here's an example of using the built-in foldLeft() method to get the sum and max of elements in an array:

```
val array = Array(2, 3, 5, 1, 6, 4)

val sum = array.foldLeft(0) { (sum, elem) => sum + elem }
val max = array.foldLeft(Integer.MIN_VALUE) { (large, elem) =>
   Math.max(large, elem) }

println(s"Sum of elements in array is $sum")
println(s"Max of elements in array is $max")
```

In an effort to make the code more concise, Scala defines some shortcut names and notations for select methods. The foldLeft() method has an equivalent operator /:. We can use foldLeft() or the equivalent /: to perform the previous actions. Methods that end with a colon (:) are treated special in Scala, as you'll learn in *Method Name Convention*, on page ?. Let's quickly take a look at using the equivalent operators instead of foldLeft():

```
val sum = (0 /: array) { (sum, elem) => sum + elem }
val max =
  (Integer.MIN_VALUE /: array) { (large, elem) => Math.max(large, elem) }
```

As an observant reader, you probably noticed the function value was placed inside curly braces instead of being sent as an argument to the foldLeft() method. That looks a lot better than sending those functions as arguments within parentheses. However, if we attempt the following on the inject() method, we will get an error:

FunctionValuesAndClosures/Inject3.scala

```
val sum = inject(array, 0) {(carryOver, elem) => carryOver + elem}
```

The previous code will result in the following error:

```
Inject3.scala:9: error: not enough arguments for method inject: (arr:
Array[Int], initial: Int, operation: (Int, Int) => Int)Int.
Unspecified value parameter operation.
val sum = inject(array, 0) {(carryOver, elem) => carryOver + elem}
                ^
one error found
```

That was not quite what we'd like to see. Before you can get the same benefit of using the curly braces that the library method enjoyed, you have to learn one more concept—currying.