

Stairway to Scala - Flight 5

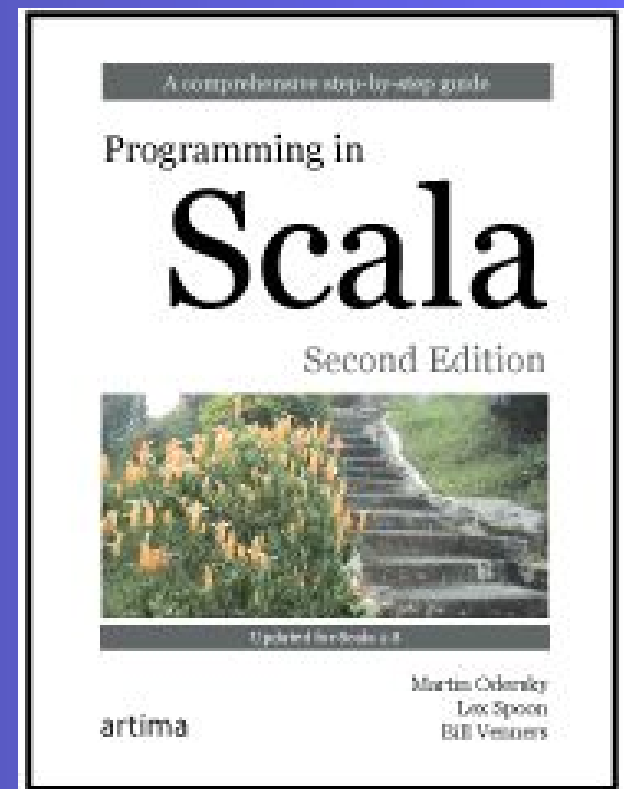
Functions and closures

Bill Venners

Dick Wall

escalatesoft.com

Copyright (c) 2010-2014 Escalate Software, LLC.
All Rights Reserved.



Flight 5 goal

Familiarize you with scala closures, and
functions as first class citizens.
(chapter 8)

Private methods

- Just like Java, Scala can have private methods

```
import scala.io.Source
object LongLines {
  def processFile(filename: String, width: Int): Unit = {
    val source = Source.fromFile(filename)
    for (line <- source.getLines)
      processLine(filename, width, line)
  }
  private def processLine(filename: String, width: Int,
    line: String): Unit = {
    if (line.length > width)
      println(s"$filename: ${line.trim}")
  }
}
```

Local functions

- But, with Scala there is another option:

```
def processFile(filename: String, width: Int): Unit = {
  def processLine(filename: String, width: Int, line: String) = {
    if (line.length > width)
      print(s"$filename: ${line.trim}")
  }
  val source = Source.fromFile(filename)
  for (line <- source.getLines) {
    processLine(filename, width, line)
  }
}
```

Local functions and scope

- But now, filename and width are in scope!

```
def processFile(filename: String, width: Int): Unit = {  
  def processLine(line: String) {  
    if (line.length > width)  
      print(s"$filename: ${line.trim}")  
  }  
  val source = Source.fromFile(filename)  
  for (line <- source.getLines) {  
    processLine(line)  
  }  
}
```

First class functions

- Why use method names at all? Function Literals:

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

- `=>` indicates that this function converts the thing on the left: `(x: Int)` into the thing on the right: `x + 1`
- *Function Values* are compiled *Function Literals* (and are consequently objects in the runtime)
- You can store function values in variables:

```
scala> val increase = { (x: Int) => x + 1 }
```

```
scala> increase(10)
```

```
res0: Int = 11
```

Can assign functions to variables

- For more than 1 statement per function literal, use { }

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

- The expression on the last line is what is evaluated and returned

Can pass functions as arguments to other functions

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

```
scala> someNumbers.foreach((x: Int) => println(x))
```

```
-5  
0  
5  
10
```

- Functions that take other functions are called higher-order functions

Short forms of function literals

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

- In this case, the type qualifier for x is redundant because the type of the someNumbers list is known, can omit it:

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

or

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

Placeholder syntax

- If each param appears only one time in the function literal, you can use placeholder syntax:

```
scala> someNumbers.filter(_ > 0)
res9: List[Int] = List(5, 10)
```

- Specifying type information for placeholder syntax:

```
scala> someNumbers.filter((_: Int) > 0)
res9: List[Int] = List(5, 10)
```

```
scala> val f = (_: Int) + (_: Int)
f: (Int, Int) => Int = <function>
scala> f(5, 10)
res11: Int = 15
```

Converting a method into a function value

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

```
scala> sum(1, 2, 3)  
res0: Int = 6
```

```
scala> val addem = { (a: Int, b: Int, c: Int) => sum(a, b, c) }  
a: (Int, Int, Int) => Int = <function3>
```

```
scala> addem(1, 2, 3)  
res1: Int = 6
```

Using underscore to represent parameters

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

```
scala> b(1, 2, 3)  
res2: Int = 6
```

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

```
scala> c(1, 2, 3)  
res3: Int = 6
```

Partially applied function

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

```
scala> d(2)  
res4: Int = 6
```

And if typing an underscore is too much

```
scala> val someNumbers = List(-11, -10, -5, 0, 5, 10)
```

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

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

```
scala> someNumbers.foreach(println)
```

```
scala> someNumbers foreach println
```

Free variables and closures

```
scala> (x: Int) => x + more
```

```
<console>:6: error: not found: value more
```

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

^

```
scala> var more = 1
```

```
more: Int = 1
```

```
scala> val addMore = { (x: Int) => x + more }
```

```
addMore: (Int) => Int = <function1>
```

```
scala> addMore(10)
```

```
res1: Int = 11
```

Visibility of changes

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

```
scala> addMore(10)  
res2: Int = 10009
```


What about parameters and local variables?

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

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

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

```
scala> inc1(10)  
res3: Int = 11
```

```
scala> inc9999(10)  
res4: Int = 10009
```

Repeated parameters (varargs)

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

```
scala> echo()
```

```
scala> echo("hi")  
hi
```

```
scala> echo("hi", "there")  
hi  
there
```

Argument expansion

```
scala> val arr = Array("hi", "there", "grandma")  
arr: Array[java.lang.String] = Array(hi, there, grandma)
```

```
scala> echo(arr)  
<console>:8: error: type mismatch;  
found   : Array[java.lang.String]  
required: String  
    echo(arr)  
      ^
```

```
scala> echo(arr: _*)  
hi  
there  
grandma
```

Named arguments

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

```
scala> speed(100, 10)
```

```
scala> speed(distance=100, time=10)
```

```
scala> speed(time=10, distance=100)
```

Default parameter values

```
def printTime(out: java.io.PrintStream = Console.out) {  
    out.println(s"time = ${System.currentTimeMillis()}")  
}
```

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

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

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

Tail recursion

@tailrec

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

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

Exercises for Flight 5