CS 2340 Objects and Design - Scala

Functions and Closures

Christopher Simpkins
chris.simpkins@gatech.edu

Aside: `drop` and `take`

def drop (n: Int): List[A]

Selects all elements except first *n* ones.

n	the number of elements to drop from this list.
returns	a list consisting of all elements of this list except the first `n` ones, or else the empty list, if this list has less than `n` elements.

Definition Classes List → LinearSeqOptimized → IterableLike → TraversableLike → GenTraversableLike

def take (n: Int): List[A]

Selects first *n* elements.

n	Tt number of elements to take from this list.
returns	a list consisting only of the first `n` elements of this list, or else the whole list, if it has less than `n` elements.

Definition Classes List → LinearSeqOptimized → IterableLike → TraversableLike → GenTraversableLike

(Optional) homework: look at the last line in the docs for these methods. Where are `drop` and `take` defined? What if you call these methods on `Set`s? Try it in the REPL with multiple sets.

Aside: `drop` and `take` in Action

```
scala> val xs = List(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
xs: List[Int] = List(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

scala> xs.take(5)
res42: List[Int] = List(0, 1, 2, 3, 4)

scala> xs.drop(2)
res45: List[Int] = List(2, 3, 4, 5, 6, 7, 8, 9)
```

Think of above calls as

- "take the first 5 elements" and
- "drop the first 2 elements."

Methods

- Methods are functions that are members of objects

```
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 (linelength > width)
      println(fileName + ": "+ line.trim)
  }
}
```

What does `trim` do? Play with it in the REPL.

Local Functions

Scala is block structured, so we can move private helper function `processLine` inside `processFile` to reduce namespace clutter:

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

  // Notice no private modifier.
  // Visibility modifiers only for methods and fields
  def processLine(line: String) {
    if (linelength > width)
      println(fileName + ": "+ line.trim)
  }
  val source = Source.fromFile(filename)
  for (line <- source.getLines())
  processLine(filename, width, line)
}
```

Now `processLine` won't show up as a code completion outside of the `processFile` method.

Also, notice how we no longer need the `fileName` and `width` parameters to `processLine`, since nested functions see names in scope inside enclosing functions.

First-Class Functions

- First class objects can be stored in variables, passed as arguments to functions, and returned from functions
- Function literals are compiled into classes that instantiate first-class objects (called function values)
- A function literal exists in source code, a function value is an object that exists at runtime.

Function literals are analogous to classes and function values are analogous to objects (instantiated classes).

First-Class Functions

Here, `(x: Int) => x + 1` is a function literal and `addOneTo` is a val of type `(Int) => Int` that holds a reference to the instantiated function value.

```
scala> val addOneTo = (x: Int) => x + 1
addOneTo: (Int) => Int = <function1>

scala> addOneTo(1)
res52: Int = 2

scala> xs.map(addOneTo)
res55: List[Int] = List(2, 3, 4, 5, 6)
```

Function Literal Shortcuts

Preceding `map` could have been called with a function literal instead of a function value:

```
scala> xs.map( (x: Int) => x + 1)
res56: List[Int] = List(2, 3, 4, 5, 6)
```

Because Scala knows `xs` is a `List[Int]`, it can infer the type of the parameter `x` using *target typing*:

```
scala> xs.map( (x) => x + 1)
res57: List[Int] = List(2, 3, 4, 5, 6)
```

We can further shorten the code with *placeholder* syntax:

```
scala> xs.map( _ + 1)
res58: List[Int] = List(2, 3, 4, 5, 6)
```

Note that multiple _ placeholders mean multiple arguments, not repeated single arguments.

Partially Aplied Functions

- You can call a function with less than all of its arguments using the _ placeholder syntax
- Result of such a call is a *partially applied function* that can later be applied to its remaining arguments

Here's a partially applied `println` function that is passed to `foreach`:

```
scala> xs.foreach(println _)
1
2
3
4
5
```

In this example, we left out the entire argument list.

Partially Aplied Functions

Notice that previous example used target typing. This doesn't work:

```
scala> val p = println _
p: () => Unit = <function0>

scala> xs.foreach(p)
<console>:10: error: type mismatch;
 found   : () => Unit
 required: (Int) => ?
       xs.foreach(p)
```

You have to specify the types:

```
scala> val p = println(_: Int)
p: (Int) => Unit = <function1>

scala> xs.foreach(p)
1
2
3
4
5
```

Partially Aplied Functions

You can flexibly supply any number of the arguments to a function:

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

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

scala> a(1, 2, 3)
res62: Int = 6

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

scala> b(2)
res63: Int = 6
```

In a context where a function is expected, you can leave off the _

```
scala> xs.foreach(println)
1
2
3
4
```

Closures

- *Bound variables* of a function are declared in the parameter list or inside the function
- *Free variables* of a function are used inside the function but are defined in an enclosing scope
- A *closure* is a function that "closes over" or "captures" the *values* of the free variables that are in an enclosing scope at the point where the closure is defined

```
scala> var more = 10
more: Int = 10
scala> val add = (x: Int) => x + more
add: (Int) => Int = <function1>
scala> add(1)
res0: Int = 11
scala> more = 20
more: Int = 20
scala> add(1)
res1: Int = 21
```

Note that `add` closed over the variable `more`, not the particular value `more` held when the closure was defined.

Closures

When you close over a variable that's local to a function that encloses your closure, the closure retains the value the variable had when the function exited.
Here, each call to makeIncreaser creates a new closure that closes over the particular actual parameter `more` for that function call

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

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

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

scala> inc1(10)
res21: Int = 11

scala> inc9999(10)
res22: Int = 10009
```

Repeated Parameters

Append `*` to the end of th type name for the last parameter to turn it into a repeated parameter.

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

scala> echo("one")
one

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

Inside `echo`, `args` is an `Array[String]`, but you can't pass an array argument becuase the parameter is a repeated parameter. If you want to pass an array, expand it in the function call with `: _*`

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

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

Named Arguments and Default Parameters

Default parameters, which must come at the end of a parameter list, can be left off in function calls.

```
scala> def speed(distance: Float, time: Float, units: String = "mph") =
     |   (distance / time).toString + " " + units
speed: (distance: Float, time: Float, units: String)java.lang.String

scala> speed(256, 16)
res1: java.lang.String = 16.0 mph
```

Named arguments allow function calls with arguments in any order.

```
scala> speed(time=16, units = "fps", distance=256)
res2: java.lang.String = 16.0 fps
```

Note that named parameters must come oafter positionally determined parameters.

```
speed(16, units = "fps", distance=256)
<console>:9: error: parameter specified twice: distance
       speed(16, units = "fps", distance=256)
```

In the example above, the unnamed first argument was assumed to be `distance`.

Tail Recursion

In a recursive function, if the recursive call is the last operation in the function, it is said to be a *tail call*.

```
scala> def factorial(n: BigInt): BigInt =
     |   if (n < 2) 1 else n * factorial(n - 1)
factorial: (n: BigInt)BigInt

scala> factorial(5)
res11: BigInt = 120
```

Is this function above tail-recursive?

Tail Recursion

This function is not tail-recursive.

```
def factorial(n: BigInt): BigInt =
  if (n < 2) 1 else n * factorial(n - 1)
```

The last operation in the function is a multiplication, which has to wait on `factorial(n -1)` to return, generating activation records for each n...1. If we call this function with a big enough number, we overflow the stack:

```
scala> factorial(50000)
java.lang.StackOverflowError
at java.math.BigInteger.subtract(BigInteger.java:1098)
at scala.math.BigInt.$minus(BigInt.scala:165)
at .factorial(<console>:8)
at .factorial(<console>:8)
...
```

How to fix?

A Tail-Recursive Factorial Function

By adding an accumulator, we can create a tail-recursive factorial function.

```
scala> def tailFactorial(n: BigInt, accum: BigInt = 1): BigInt =
     |   if (n < 2) accum else tailFactorial(n - 1, n * accum)
tailFactorial: (n: BigInt, accum: BigInt)BigInt
```

Now the function generates an iterative, rather than a recursive process (generates only one activation record that changes for each n...1).

```
scala> tailFactorial(50000)
res19: BigInt = 33473205095971448369154760940714864779127732238
... (and, like, a finity more digits)
```

Notice that, thanks to the default parameter, we can make the function call more “natural,” leaving off the initial value for the accumulator. Is `tailFactorial` well designed?

A Better Design for `factorial`

Our previous `tailFactorial` is poorly designed, because client code can choose to pass a different initial value for `accum`, causing incorrect results.

```
scala> tailFactorial(5, 2)
res21: BigInt = 240
```

We can use a local function that implements our tail-recursive factorial, keeps the interface simple, and doesn't permit users to mess it up.

```
scala> def factorial(n: BigInt) = {
     |   def tailFactorial(n: BigInt, accum: BigInt): BigInt = {
     |     if (n < 2) accum else tailFactorial(n - 1, n * accum)
     |   }
     |   tailFactorial(n, 1)
     | }
...
scala> factorial(5)
res22: BigInt = 120

scala> factorial(50000)
res23: BigInt = 33473205095971448369154760940714864779127732238104548070
... (and many more digits)
```

Limits of Tail Recursion

Scala can't optimize mutual tail recursion.

```
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)
```

And becuase the JVM doesn't optimize tail calls, function values in tail position are not optimized.

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

Higher-Order Functions

A function that takes another function as a parameter is called a *higher-order function*.
Map is the quintessential example:

```
scala> xs
res4: List[Int] = List(1, 2, 3)

scala> xs.map(math.pow(_,2))
res5: List[Double] = List(1.0, 4.0, 9.0)
```