Unit 5a: Closures

1 Introduction

The following function definition (as a lambda) depends on the value of two free variables, *i* and *factor*:

```
val multiplier = (i:Int) => i * factor
```

i is a parameter of the function, so it takes on a value whenever the function is called. However, *factor* is not a parameter – it is external to the function. A compilation error results as *factor* cannot be resolved.

If, however, there is a local variable *factor* declared within the same scope as the function definition, then the function can use its value when called.

```
val factor = 3
val multiplier = (i:Int) => i * factor

scala> multiplier(4)
res0: 12
```

This demonstrates something interesting (though it may not be immediately obvious why it's useful). A function encompasses or is "closed over" the environment where it is declared – it is a closure. The function *multiplier* is closed over any variables that are external to the function and that are in scope where it is declared. A function that has no external variables is closed over itself. This becomes useful when using HO functions where:

- you want to pass a function around like a variable
- while doing so, you want that function to be able to refer to variables that were in the same scope as the function when it was declared but are not in the scope where the function is called

2 Closure and scope demonstration

We can demonstrate with an example that has two different scopes – inside the function *main* and inside the function *printTransformedValue*.

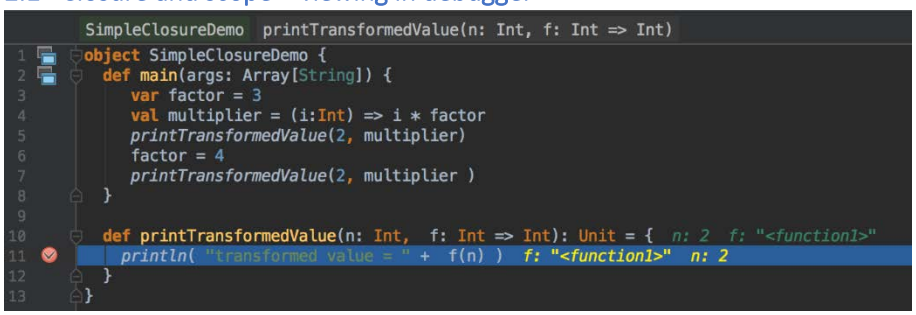
printTransformedValue is a HO function that can be composed with a function (*multiplier* in this case) that does a transformation to the value supplied:

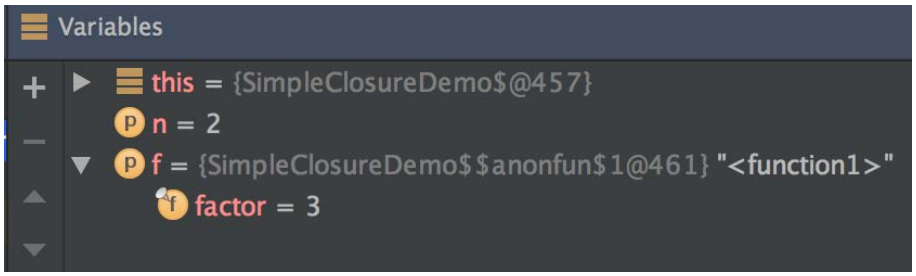
```
object SimpleClosureDemo {

  def main(args: Array[String]) {
     var factor = 3
     val multiplier = (i:Int) => i * factor

     printTransformedValue(2, multiplier)

     factor = 4
     printTransformedValue(2, multiplier )
  }

  def printTransformedValue(n:Int,f:Int => Int): Unit = {
    println("transformed value = " +  f(n))
  }
}
```

The variable *factor* is in scope when the function *multiplier* is declared; therefore, *multiplier* is a closure over it. *multiplier* is passed as a parameter to *printTransformedValue*; it is actually called inside *printTransformedValue* – the variable *factor* is not in scope where the function is called, but its value is used as it is part of the closure. The value of *factor* can be changed after the function *multiplier* is declared and the closure contains the new value.

```
transformed value = 6
transformed value = 8
```

2.1 Closure and scope – viewing in debugger





The breakpoint in *printTransformedValue* – only *n* and *f* in scope, but *factor* is in the closure scope.

3 Closures – a nice explanation

"Personally, I like to think of a closure as being like quantum entanglement, which Einstein referred to as "a spooky action at a distance." Just as quantum entanglement begins with two elements that are together and then separated — but somehow remain aware of each other — a closure begins with a function and a variable defined in the same scope, which are then separated from each other. When the function is executed at some other point in space (scope) and time, it is magically still aware of the variable it referenced in their earlier time together, and even picks up any changes to that variable."

From http://alvinalexander.com/scala/how-to-use-closures-in-scala-fp-examples, read for another example and other explanations

4 Closures – practical applications

Closures seem abstract, but are very useful for communication between different scopes; they make some things with HO functions simple that would be much more difficult without closures. For example, this filters a list with a threshold value of 3:

```
List(1,2,3,4,5,6).filter((x:Int) => x > 3)
```

What if we want a function that filters with a threshold value as a parameter?

```
def paramFilter(threshold: Int) = {
  List(1, 2, 3, 4, 5, 6).filter((x:Int) =>
        x > threshold)
}
```

This would not work without closures – parameter *threshold* is in scope within the *paramFilter* function, so *filter* function (the lambda expression) is closed over it.

However, the *filter* function is called in the *filter* method of the *List* class, which is in another scope, so the value of *threshold* would not be available to that call without the closure.

They are also useful for functions that create functions that "know" about the environment where they were created, widely used to overcome some limitations of JavaScript, for example.