BA-BEAD Big Data Engineering for Analytics

BEAD Workshop Series

Functional Thinking in Scala

Table of Contents

Introduction

This set of instructions would guide you through the functional thinking workshop using Scala and Scala IDE.

Functions Vs Methods

***Functions** are objects, methods are not. Functions are named, reusable expressions. **Methods** are associated with a class, functions are not.*

The terms **method** and object are often used interchangeably because methods can be stored in objects quite easily. For example:

```
def getArea2(radius:Double):Double = { val PI = 3.14; PI * radius * radius }
```

Member in summary has the below properties:

1. Methods are not value types; methods cannot serve as r-values defined using def.
2. Methods are not objects, but can be converted to function objects quite easily.
3. Methods have signature, but no independent type.
4. Slightly faster and better performing.
5. Methods are not first class entities unless converted to functions.
6. Methods work fine with type parameters and parameter default values.

Function objects are first class entities on par with classes - methods are not. For example:

```
val getArea = (radius:Double) =>
 {
     val PI = 3.14;
     PI * radius * radius
 }:Double
```

Functions in summary has the below properties:

1. Functions are value types; can be stored in val and var storage units.
2. Functions are objects of type function0, function1, ...(traits descending from AnyRef)
3. Functions have both a type and a signature (type is function0,function1, ..).
4. Slightly slower, and higher overhead.
5. Functions are first class entities on par with classes.
6. Functions do not accept type parameters or parameter default values.

Statements are units of code that do not return a value. Expressions are units of code that return a value. The last expression in a block is the return value for the entire block. Functions are named, reusable expressions. Expressions can be stored in values or variables and passed into functions.

Simple Examples:

Decorate Function

```
def decorate(str: String, left: String = "[", right: String = "]") =
               left + str + right
```

Sum using tail recursion

```
def recursiveSum(args: Int*) : Int = {
  if (args.length == 0) 0
  else args.head + recursiveSum(args.tail : _*)
}
```

Currying

```
scala> def multiplier(i: Int)(factor: Int) = i * factor
```

```
multiplier: (i: Int)(factor: Int)Int
scala> val byFive = multiplier(5)
_byFive: Int => Int = <function1>
scala> val byTen = multiplier(10)
_byTen: Int => Int = <function1>
scala> byFive(2)
res8: Int = 10
scala> byTen(2)
res9: Int = 20
```

List and Maps

```
scala> List(1, 2, 3, 4, 5) foreach { i => println("Int: " + i) }
Int: 1
Int: 2
Int: 3
Int: 4
Int: 5
scala> val stateCapitals = Map(
     |   "Alabama" -> "Montgomery",
     |   "Alaska"  -> "Juneau",
     |   "Wyoming" -> "Cheyenne")
stateCapitals: scala.collection.immutable.Map[String,String] =
Map(Alabama -> Montgomery, Alaska -> Juneau, Wyoming -> Cheyenne)
// stateCapitals foreach { kv => println(kv._1 + ": " + kv._2) }
scala> stateCapitals foreach { case (k, v) => println(k + ": " + v) }
Alabama: Montgomery
Alaska: Juneau
Wyoming: Cheyenne
```

FlatMap

```
scala> val list = List("now", "is", "", "the", "time")
list: List[String] = List(now, is, "", the, time)
scala> list flatMap (s => s.toList)
res0: List[Char] = List(n, o, w, i, s, t, h, e, t, i, m, e)
```

Flatten

```
scala> val list2 = List("now", "is", "", "the", "time") map (s => s.toList)
list2: List[List[Char]] =
  List(List(n, o, w), List(i, s), List(), List(t, h, e), List(t, i, m, e))
scala> list2.flatten
res1: List[Char] = List(n, o, w, i, s, t, h, e, t, i, m, e)
```

Filter

```
scala> val stateCapitals = Map(
     |   "Alabama" -> "Montgomery",
     |   "Alaska"  -> "Juneau",
     |   "Wyoming" -> "Cheyenne")
stateCapitals: scala.collection.immutable.Map[String,String] =
  Map(Alabama -> Montgomery, Alaska -> Juneau, Wyoming -> Cheyenne)
scala> val map2 = stateCapitals filter { kv => kv._1 startsWith "A" }
map2: scala.collection.immutable.Map[String,String] =
  Map(Alabama -> Montgomery, Alaska -> Juneau)
```

Fold and Reduce

```
scala> val list = List(1,2,3,4,5,6)
list: List[Int] = List(1, 2, 3, 4, 5, 6)
scala> list reduce (_ + _)
res0: Int = 21
scala> list.fold (10) (_ * _)
res1: Int = 7200
```

Scan

```
scala> val list = List(1, 2, 3, 4, 5)
list: List[Int] = List(1, 2, 3, 4, 5)
scala> (list scan 10) (_ + _)
res0: List[Int] = List(10, 11, 13, 16, 20, 25)
```

First the seed value 10 is emitted, followed by the first element plus the seed, 11, followed by the second element plus the previous value, 11 + 2 = 13, and so on.

Exercises

1. Write a function that computes the area of a circle given its radius.
2. Provide an alternate form of the function in exercise 1 that takes the radius as a `String`. What happens if your function is invoked with an empty `String` ?
3. Write a recursive function that prints the product of all values from an array of integer numbers, without using `for` or `while` loops. Can you make it tail-recursive?
   (Hint use Sum Example as inspiration)
4. Write a function that takes a milliseconds value and returns a string describing the value in days, hours, minutes, and seconds. What's the optimal type for the input value?
5. Write a function that calculates the first value raised to the exponent of the second value. Try writing this first using `math.pow`, then with your own calculation. Did you implement it with variables? Is there a solution available that only uses immutable data? Did you choose a numeric type that is large enough for your uses?
6. Write a function that calculates the difference between a pair of 2D points (x and y) and returns the result as a point.
   (Hint: this would be a good use for tuples (see Tuples Examples for inspiration).
7. Write a function that takes a 2-sized tuple and returns it with the `Int` value (if included) in the first position.
   (Hint: this would be a good use for type parameters and the `isInstanceOf` type operation.)
8. Write a function that takes a 3-sized tuple and returns a 6-sized tuple, with each original parameter followed by its `String` representation. For example, invoking the function with `(true, 22.25, "yes")` should return `(true, "true", 22.5, "22.5", "yes", "yes")`. Can you ensure that tuples of all possible types are compatible with your function? When you invoke this function, can you do so with explicit types not only in the function result but in the value that you use to store the result?