A brief introduction to Functional Programming in Scala

Amanj Sherwany

20 Aug 2013

1 Let's do Scala

Translate this into Scala:

```
// in Java
class Person {
       private String name;
       private int age;

       public Person(String name, int age) {
              this.name = name;
              this.age = age;
       }

       public void setName(String name) {
              this.name = name;
       }

       public String getName() { return name; }

       public void setAge() {
              this.age = age;
       }

       public int getAge() {
              return age;
       }
}
```

```

// in Scala

class Person(name: String, age: Int) // age and name are vals not vars
```

Now what if we want to set some conditions on age? Like not allowing negative numbers.

```
class Person(name: String, age: Int) {
       require(age > 0) // no semicolon
}
```

What if we want a special message on failure?

```scala
class Person(name: String, age: Int) {
      require(age > 0, "Negative age is impossible") // no semicolon
}
```

2 Programming Paradigms:

- Imperative Programming
  - Procedural Programming
  - Object-Oriented Programming
- Logic Programming
- Functional Programming

3 Imperative Programming has:

- Modifying mutable variables (mutable vs immutable)
- Using assignments
- Control structures (If-else, loops, breaks, continue, return ...etc)

4 Functional Programming:

- Programming without mutable variables, assignments, loops, etc
- Functions as building blocks (first class functions)
  - Can be defined anywhere
  - Can be passed like any other values
  - Functions can be composed using a set of operators

Examples of Functional languages: Haskell, SML, Scala, Lisp...

4.1 Why Functional Programming?

Functional Programming is becoming increasingly popular because it offers an attractive method for exploiting parallelism for multicore and cloud computing. And testing becomes easier. Why?

4.2 Var vs val (mutable vs immutable)

4.3 Binary and unary Operations (Logic and Arithmetic)

4.4 Data types (Int, Double, Float, Boolean,)

4.5 Lists but not arrays

4.6 Tuples

```
val list1 = List(1, 2, 3, 4)
val list2 = List('a', 'b', 'c', 'd')

val zipped = list1 zip list2
val (fst, snd) = zipped unzip
```

4.7 Functions and Nested functions

4.8 No loop, but recursion:

```
def factorial(num: Int): Int = {
     if(num == 0 || num == 1) 1
     else num * factorial(num-1) // No return keyword is required
}

def sum(list: List[Int]): Int = {
     def sumAux(list2: List[Int], currentSum: Int) = {
            if(list2 == Nil) currentSum
            else (sumAux(list2.tail, list2.head + currentSum)
     }
     sumAux(list, 0)
}
```

If needed writing functions: sqrt, even and odd

4.9 Case classes and pattern matching

Now re-write the above two examples in terms of pattern matching

If time allowed talk about a simple BinOp language:

```
tpe := n      # n is any number
bop := +
```

```

class Expr

case class Num(n: Int) extends Expr

case class Add(fst: Expr, snd: Expr) extends Expr
```

```
def eval(expr: Expr): Num = {
  expr match {
                case n: Num => n

                case Add(f, s) =>
                       val fst = eval(f)
                       val snd = eval(s)
                       Num(fst.n + snd.n)
        }
}
```

Now lets do the same thing in Java

```
  public interface Expr {}

  public class Num implements Expr {
    public int n;
  }

  public class Add implements Expr {
    public Expr fst;
    public Expr snd;
  }


  public class Eval {
    public static Num eval(Expr expr) {
      if(expr instanceof Num) return (Num) expr;
      else if(expr instanceof Add) {
        Add add = (Add) expr;
        Num e1 = eval(add.fst)
        Num e2 = eval(add.snd)
        return new Num(e1.n + e2.n);
      }
    }
  }
```

4.10 Function values

```
def plus(a: Int)(b: Int): Int = a + b

val plus1 = plus(1) _
```

4.11 Currying

There is actually no function which takes multiple parameters, but there are functions that return functions.

So, *plus* is a function that takes one Int and returns a function that takes an Int and returns an Int, written like:

```
plus = Int => Int => Int
```

4.12 Anonymous functions (lambdas)

```
(x: Int, y: Int) => x + y
```

We can store lambdas inside variables/values

```
val plus = (x: Int, y: Int) => x + y
```

What is the type of the above lambda function?

```
(Int, Int) => Int
```

4.13 Higher-order functions

Lets write a function that takes a function and applies it:

```
def apply(number: Int, f: Int => Int): Int = {
       f(number)
}

apply(3, (x: Int) => x + 3))

// or

apply(3, _ + 3)
```

Fold, Map, Foreach and Filter (and a lot of examples) sum, length, add every element by 2

```
def sum(list: List[Int]): Int = list.foldLeft(0)((x: Int, y: Int) => x +
    y)

// or

def sum(list: List[Int]): Int = list.foldLeft(0)(_ + _)

def length(list: List[Int]): Int = list.foldLeft(0)((x, y) => x + 1)

def addBy2(list: List[Int]): List[Int] = list.map((x: Int) => x + 2)

// or

def addBy2(list: List[Int]): List[Int] = list.map(_ + 2)

// How to print every list element:

def printAll(list: List[Int]): Unit = list.foreach((x: Int) =>
    println(x))
```

```scala
// or

def printAll(list: List[Int]): Unit = list.foreach(println(_))

// only the odd items

def oddOnly(list: List[Int]): List[Int] = list.filter((x: Int) => x % 2
    != 0)

// or

def oddOnly(list: List[Int]): List[Int] = list.filter(_ % 2 != 0)
```