Programming in Scala

based on the book
“Programming in Scala Third Edition”

Agenda

- Chapter VIII - Functions and Closures
    - Methods
    - Local functions
    - First-class functions
    - Partially applied functions
    - Closures
    - Special function call
    - Tail recursion
- Chapter IX - Control Abstraction
    - Currying
    - Writing new control structures

Chapter VIII - Functions and Closures

Methods

- Scala also has the construct functions, but it has various types of functions, like methods, nested functions, function literals, etc...
- Methods are functions what are members of objects

```
object ObjectName {
    def methodName(param: String) = {
        // method body
    }
}
```

Nested (or local) functions

- Just like local variables, local functions are visible only in their enclosing block
- It is a tool for hiding logic
- You can also have private methods for the same purpose
- Local functions can access the parameters of their enclosing function

```
object ObjectName {
    def enclosingFunction(param: Int) = {
        def nestedFunction(nestedParam: Int) =
            param + nestedParam
        nestedFunction(3)
    }
}
```

First-class functions (function literals)

- You can write down functions as unnamed literals and then pass them around as values.
- A function literal is compiled into a *function class* that when instantiated at runtime is a *function value* (Function0, Function1...)

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

// increase(10)

list.filter((x) => x == 10) // target typing!!

list.filter(x => x == 10)
```

Placeholder syntax

- To make a function literal even more concise, you can use underscores as placeholders for one or more parameters
- You can think of the underscore as a “blank” in the expression that needs to be “filled in.”
- **Multiple underscores mean multiple parameters, not reuse of a single parameter repeatedly!**

`list.filter(_ > 0)` vs. `list.filter(x => x > 0)`

Partially applied functions

- you can also replace an entire parameter list with an underscore

`println(_)` vs. `println _` vs. `x => println(x)`

- When you use an underscore in this way, you are writing a partially applied function.
- In Scala, when you invoke a function, passing in any needed arguments, you apply that function to the arguments.
- A partially applied function is an expression in which you don’t supply all of the arguments needed by the function

```
1.  def sum(a: Int, b: Int, c: Int) = a + b + c
2.  sum(1, 2, 3) // 6
3.  val a = sum _
```

`a` is a function value. A function value class is generated by the compiler at this point which has an apply method with 3 args in this case (it implements the Function3 trait). This generated method calls `sum` with its args values

```
4.  a(1, 2, 3) // 6
```

The compiler translates this to `a.apply(1,2,3)`

```
5.  val b = sum(1, _ : Int, 3)
6.  b(2) // 6
```

Closures

- Closures are function literals that use *free variables*

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

- `someValue` is a *free variable* and `x` is a *bound variable*
- You have to provide value for all the free variables when invoking the function literal
- The *function value* created from this type of function literal is called a *closure*
- *A function literal* without any free variable is called a *closed term*, with free variables it is called an *open term (open term -> closure)*

Closures #2

- Closures capture variables themselves, not the value to which variables refer. (variable value changes are reflected when evaluating closures)
- Changes made by a closure to a captured variable are visible outside the closure
- The generated function value always contains a captured value for all of its free variables

Repeated parameters

- It is like varargs in Java
- Scala allows you to indicate that the last parameter to a function may be repeated
- Inside the function, the type of the repeated parameter is an Array of the declared type of the parameter.

```
def methodName(args: String*) = {
    // method body
}
```

Named arguments

- Normally the actual parameter list is matched to the formal parameter list based on the order of the actual parameters
- Named arguments allows to match actual parameters to formal parameters based on their name and not their order

```
def methodName(a : Int, b: Int) = {...}

methodName(b = 2, a = 3)
```

Default parameter values

- You can define default value for function parameters
- The argument for such a parameter can optionally be omitted from a function call, in which case the corresponding argument will be filled in with the default.

```
def printTime(out: java.io.PrintStream = Console.out) =

    out.println("time = " + System.currentTimeMillis())

printTime()

printTime(Console.err)
```

Tail recursion

- In FP you should prefer recursion over while loops
- The performance can be a question because recursive function call is usually expensive (new stack frame creation, etc..)
- Tail recursion is a special recursion when the recursive call is the last call in the function body
- Scala compiler can make a special optimization for tail recursive functions, it replaces the last call with a jump back to the beginning of the function body without doing a new method call!

```
def boom(x: Int): Int =
    if (x == 0) throw new Exception("boom!")
    else boom(x - 1)
```

vs

```
def boom(x: Int): Int =
    if (x == 0) throw new Exception("boom!")
    else boom(x - 1) + 1 // this is not tail recursive!
```

Chapter IX - Control Abstraction

Higher-order functions

- Functions that take functions as parameters
- One benefit of higher-order functions is they enable you to create control abstractions that allow you to reduce code duplication.
- Really similar to Java 8 new API’s that accepts lambda expressions (closures) as attributes, like filter, map, etc...

```
def filesMatching(
    query: String,
    matcher: (String, String) => Boolean) = { … }
```

Currying

- A curried function is applied to multiple argument lists, instead of just one

```
def curriedSum(x: Int)(y: Int) = x + y
```

- When you invoke a function like this you actually make 2 function invocations
  - The first function invocation takes a single Int parameter named x, and returns a function value for the second function.
  - This second function takes the Int parameter y.
  - To illustrate what happens:

```
def first(x: Int) = (y: Int) => x + y
val second = first(1)
```

Currying

- This is what happens actually:

```
val onePlus = curriedSum(1)_ // a reference function
onePlus(2) // 3
val twoPlus = curriedSum(2)_
twoPlus(2) // 4
```

Writing new control structures

- With the help of all these techniques you can easily create structures (functions) that feel like native language control structures
- A popular way to do this is when passing functions as arguments to methods

```
def twice(op: Double => Double, x: Double) = op(op(x))

twice(_ + 1, 5) // 7.0
```

- To make your code looks more like a native element, you can use curly braces instead of parenthesis when you have only 1 argument

```
increment(1) vs increment {1}
```

Writing new control structures

- What if you have more than one argument? Use currying!

```
def withPrintWriter(file: File)(op: PrintWriter => Unit)=
{ … }

val file = new File("date.txt")
withPrintWriter(file) {
    writer =>    writer.println(new java.util.Date)
}
// withPrintWriter(
//     file,
//     writer =>    writer.println(new java.util.Date))
```

Writing new control structures

- You can do it even better! Let's combine all this with **by-name parameters**
- By-name parameter is a different thing than named arguments!
- To make a by-name parameter, you give the parameter a type starting with `=>` instead of `() =>.` (a.k.a: you can left out the empty input param notation)
- By declaring a parameter as `a: => A` we are telling Scala to evaluate `a` only when it is used (which may be never).

  `a: Boolean` vs `a: => Boolean`
- A by-name type is only allowed for parameters. There is no such thing as a by-name variable or a by-name field.

```
def myAssert(predicate: () => Boolean) = {...}
myAssert(() => 5 > 3)
```

vs.

```
def byNameAssert(predicate: => Boolean) = {...}
byNameAssert(5 > 3)
```

+1 The Homework :)

This time it is going to be easy.

Try to practice all these things with simple examples!