Advanced Functional Thinking

Dr Surbhi Saraswat
School of Computer Science

Variables are Immutable

- In functional programming, we can’t modify a variable after it’s been initialized.
- We can create new variables – but we can’t modify existing variables, and this really helps to maintain state throughout the runtime of a program.
- Once we create a variable and set its value, we can have full confidence knowing that the value of that variable will never change.

Variables are Immutable

- The literal meaning of Immutability is unable to change.
  - In the Functional Programming world, we create values or objects by initializing them. Then we use them, but we do not change their values or their state. If we need, we create a new one, but we do not modify the existing object's state.

- Scala has two types of variables.
  - **var:** The *var* stands for a variable, and the *val* stands for value. You can initialize a *var*, and later you can reassign a new value to the *var*.

```
var s = "Hello World!"        //After initializing it once, you can change it later.
s = "Hello Scala!”
```

  - **val:** The *val* is a constant. That means, once initialized, You cannot change it.

```
val v = "You cannot change me."
v = "Let me try.”        //Output:- <console>:8: error: reassignment to val
```

Since Scala is a hybrid language, It supports *var* and *val* both.

Functions

- A function is a collection of statements that perform a certain task.
- **Function vs Method:**
  - Function is an object which can be stored in a variable.
  - A method always belongs to a class which has a name, signature etc.
  - Basically, a method is a function which is a member of some object.

Syntax:

```
def funct_name ([param_list]):[return_type] = {
   // function body
   }
```

Functions

```
object GfG {

    def main(args: Array[String]) {
       // Calling the function
       println("Sum is: " + add(5,3));
    }

    // declaration and definition of function
    def add(a:Int, b:Int) : Int = {
          var sum = 0
          sum = a + b
          // returning the value of sum
          return sum
         }
}
```

Method

Function

Pure Functions

```
def multiplyByTwo (i : Int): Int = { i * 2 }
```

Let's analyze this small code and draw some observations.

- **The Input solely determines the output.**
  - There is no other thing like a global variable or the content of a file, or input from the console that determines the output. It is only the input parameter value, nothing else.
  - No matter how many times or where do you invoke this function, as long as the input parameter value is same, you are going to get the same output.
- **The function does not change its input.**
  - The above code snippet takes *i* as an input and uses it to calculate the output. However, it does not change the value of *i*.
  - A pure function guarantees that the input value remains unchanged. It never modifies the Input value.

Pure Functions

```
def multiplyByTwo (i : Int): Int = { i * 2 }
```

Let's analyze this small code and draw some observations.

- **The Function does not do anything else except computing the output- no side effects.**
  - The above code snippet does not read anything from a file or console. It does not print anything on the console or write some data to a file.
  - It does not read or modify a global variable or for that matter anything outside the function. In fact, it does not perform any I/O.
  - If it does anything else that impacts the outside world or is visible to the outside world, we call it a side effect of the function.
  - A side effect is like doing something other than your primary purpose. So, a function is pure if it is free from side effects.

Anonymous Functions

- A standard function has a name, a list of parameters, a return type, and a body. If you do not give a name to a function, it is an anonymous function.
- Let's take a simple example in Scala.
  - `def doubler (i: Int) = { i * 2 }`
  - `(i: Int) => { i * 2 }`
- A normal function uses = symbol whereas an Anonymous function uses => symbol.
- Some people call it an anonymous function whereas others may refer to it as *lambda*.
- Assign it to a variable. `val d = (i: Int) => { i * 2 }`
- Call it using the variable. `d(3)` //Output: 6

Anonymous Functions

Syntax:

```
(z:Int, y:Int)=> z*y

 Or

(_:Int)*(_Int)
```

- In the above **first syntax, =>** is known as a transformer. The transformer is used to transform the parameter-list of the left-hand side of the symbol into a new result using the expression present on the right-hand side.

- In the above **second syntax, _** character is known as a wildcard is a shorthand way to represent a parameter who appears only once in the anonymous function.

Anonymous Functions

```
object Main
{
   def main(args: Array[String])
   {
      // Creating anonymous functions with multiple parameters
      // Assign anonymous functions to variables
      var myfc1 = (str1:String, str2:String) => str1 + str2

      // An anonymous function is created using _ wildcard instead of variable name
      // because str1 and str2 variable appear only once
      var myfc2 = (_:String) + (_:String)

      // Here, the variable invoke like a function call
      println(myfc1("Geeks", "12Geeks"))
      println(myfc2("Geeks", "forGeeks"))
   }
}
```

Scala Closures

- **Scala Closures** are functions which uses one or more free variables and the return value of this function is dependent of these variable.
- The free variables are defined outside of the Closure Function and is not included as a parameter of this function.

Example:

```
// defined the value of p as 10

val p = 10

// define this closure.

def example (a:double) = a*p / 100
```