Chapter 8: Functions

Exercise:

1. Understand the difference between functions and methods in the context of functional programming in Scala.
2. Consider a use case of recursion and try applying it.
3. Try using functions within functions, i.e., local functions. Understand whether you can refer to inner functions in outer scope.
4. Understand whether variables were copied by value or by reference in Scala and what the implications of doing so are.
5. Understand the best practices of functions, in that they should be designed so that they perform one and only one task.

Answer :

- Functions and methods in Scala represent similar concepts, but there are significant differences in how we use them.

A function is a callable unit of code that can take a single parameter, a list of parameters, or no parameters at all. A function can execute one or many statements and can return a value, a list of values, or no values at all.

Methods are essentially functions that are parts of a class structure, can be overridden, and use a different syntax. Scala doesn't allow us to define an anonymous method.

- Recursion

Is the technique of making a function call itself directly or indirectly and the corresponding function is called a recursive function. This technique provides a way to break complicated problems down into simple problems which are easier to solve. Recursion may be a bit difficult to understand and it can take some time to get your head around how it works

In the recursive program, the solution to the base case is provided and the solution of the bigger problem is expressed in terms of smaller problems. Let us take an example to understand this.

Recursion plays a big role in pure functional programming and Scala supports recursion functions very well. Recursion means a function can call itself repeatedly.

Try the following program, it is a good example of recursion where factorials of the passed number are calculated.

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

def gcd(a: Int, b: Int): Int =
  if (b == 0) a else gcd(b, a % b)

// Exiting paste mode, now interpreting.

gcd: (a: Int, b: Int)Int

scala> gcd(78,63)
res0: Int = 3

scala> :paste
// Entering paste mode (ctrl-D to finish)

def factorial(n: Int): Int =
if (n == 0) 1 else n * factorial(n - 1)

// Exiting paste mode, now interpreting.

factorial: (n: Int)Int

scala> factorial(12)
res1: Int = 479001600
```

- local functions

If you understand the Scala function, the local function should be the most simple idea among all other Scala functional concepts. When we use an Object Oriented approach, it is quite common to create private methods. They are a kind of helper methods, and we make them private because we do not want them to expose to the external world.

Scala offers the same notion using the local functions. You can define functions inside other functions. That's what we call as a Local Function. The local functions are visible only in their enclosing block.

- **“**Functions should DO ONLY ONE THING. And they should do it well."

The first rule of functions is that they should be small. The second rule of functions is that they should be smaller than small. So, this means that your function should not be large enough to hold nested structures. Therefore, the indent level of a function should not be greater than one or two. This technique makes it easier to read, understand, and digest.

Your function can take also the minimum number of the argument, which is zero. Such functions do not depend on any input, so there’s always going to be some trivial output from it. However, it will be easy to test.

Next comes one (monadic), which is based on functional programming rules, followed closely by two (dyadic). Three arguments (triadic) should be avoided wherever possible. More than three (polyadic) requires very special justification, and even then, they shouldn’t be used anyway.

However, in case the number of arguments are more than three, we should group them into a meaningful object, like so: