FUNCTIONS

Scala Functions

- Scala supports functional programming approach.
- It provides rich set of built-in functions and allows you to create user defined functions also.
- In scala, functions are first class values.
- You can store function value, pass function as an argument and return function as a value from other function.
- You can create function by using def keyword.
- You must mention return type of parameters while defining function and return type of a function is optional.
- If you don't specify return type of a function, default return type is Unit.

Synatax:

```
def functionName(parameters : typeofparameters) : return typeoffunction = {
// statements to be executed
}
```

- can create function with or without = (equal) operator. If you use it, function will return value.
- If we don't use it, function will not return anything and will work like subroutine.

Scala Function Example without using = Operator

- The function defined below is also known as non parameterized function.

```
object MainObject {
  def main(args: Array[String]) {
      functionExample()          // Calling function
   }
   def functionExample()  {        // Defining a function
        println("This is a simple function")
   }
}
```

Scala Function Example with = Operator

```
object MainObject {
   def main(args: Array[String]) {
       var result = functionExample()        // Calling function
       println(result)
    }
    def functionExample() = {       // Defining a function
         var a = 10
         a
    }
}
```

Scala Parameterized Function Example

when using parameterized function you must mention type of parameters explicitly otherwise compiler throws an error and your code fails to compile.

```
object MainObject {
   def main(args: Array[String]) = {
       functionExample(10,20)
    }
    def functionExample(a:Int, b:Int) = {
         var c = a+b
         println(c)
    }
}
```

Scala Recursion Function

- In scala, you can create recursive functions also.
- There must be a base condition to terminate program safely.

```
object MainObject {
   def main(args: Array[String]) = {
       var result = functionExample(15,2)
       println(result)
    }
    def functionExample(a:Int, b:Int):Int = {
       if(b == 0)          // Base condition
        0
       else
        a+functionExample(a,b-1)
    }
}
```

- Recursion is a method which breaks the problem into smaller sub problems and calls itself for each of the problems.
- That is, it simply means function calling itself.
- We can use recursion instead of loops. Recursion avoids mutable state associated with loops.
- Recursion is quite common in functional programming and provides a natural way to describe many Algorithms.
- Recursion is considered as to be important in functional programming. Scala supports Recursion very well.

```
object FactorialRecursion {

    def main(args: Array[String]) {
        println(factorial(5))
    }

    def factorial(n: Int): Int = {
      if (n == 0)
          return 1
      else
          return n * factorial(n-1)
    }
}
```

5 * factor(4) = 5 * 24 = 120

4 * factor(3) = 4 * 6 = 24

3 * factor(2) = 3 * 2 = 6

2 * factor(1) = 2

Tail Recursion

- Tail recursion is a subroutine (function, method) where the last statement is being executed recursively. Simply saying, the last statement in a function calls itself multiple times.
- These functions are more perform-ant as they take advantage of tail call optimization. They simply keep calling a function (the function itself) instead of adding a new stack-frame in memory.
- Tail recursive functions should pass the state of the current iteration through their arguments.They are immutable (they are passing values as argument, not re-assigning values to the same variables).

- The **tail recursive functions** better than non tail recursive functions because tail-recursion can be optimized by compiler.
- A recursive function is said to be tail recursive if the recursive call is the last thing done by the function.
- There is no need to keep record of the previous state.
- For tail recursion function a package import **scala.annotation.tailrec** will be used in the program.

- **Syntax:**

```
@tailrec
def FuntionName(Parameter1, Parameter2, ...): type = …
```

```
import scala.annotation.tailrec

// Creating object
object tailrec
{
    // Function defined
    def factorial(n: Int): Int =
    {
        // Using tail recursion
        @tailrec def factorialTail(acc: Int, n: Int): Int =
        {
            if (n <= 1)
                acc
            else
                factorialTail(n * acc, n - 1)
        }
        factorialtail(1, n)
    }

    // Main method
    def main(args:Array[String])
    {
        println(factorial(5))
    }
}
```

- A tail recursive function call allows the compiler to perform a special optimization which it normally can not with regular recursion.
- In a tail recursive function, the recursive call is the very last thing to be executed.
- In this case, instead of allocating a stack frame for each call, the compiler can rework the code to simply reuse the current stack frame, meaning a tail-recursive function will only use a single stack frame as opposed to hundreds or even thousands.
- This optimization is possible because the compiler knows that once the tail recursive call is made, no previous copies of variables will be needed, because there is no more code to execute.
- If, for instance, a print statement followed a recursive call, the compiler would need to know the value of the variable to be printed after the recursive call returns, and thus the stack frame cannot be reused.

Anonymous Functions in Scala

- In Scala, An **anonymous function** is also known as a function literal. A function which does not contain a name is known as an anonymous function. An anonymous function provides a lightweight function definition. It is useful when we want to create an inline function.
- **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.

- When a function literal is instantiated in an object is known as a function value.
- Or in other words, when an anonymous function is assigned to a variable then we can invoke that variable like a function call.
- We can define multiple arguments in the anonymous function.

```
object Main
{
    def main(args: Array[String])
    {

var myfc1 = (str1:String, str2:String) => str1 + str2

var myfc2 = (_:String) + (_:String)

println(myfc1(“Good", “Morning"))
        println(myfc2(“Hello", “World"))
    }
}
```

SCALA CLOSURE

- 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.
- So the difference between a closure function and a normal function is the free variable.
- A free variable is any kind of variable which is not defined within the function and not passed as the parameter of the function.
- A free variable is not bound to a function with a valid value.
- The function does not contain any values for the free variable.

- def example(a:double) = a*p / 100
- Now on running the above code we'll get an error starting not found p. So now we give a value to **p** outside the function.

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

// define this closure.
def example(a:double) = a*p / 100
```

- The closure function takes the most recent state of the free variable and changes the value of the closure function accordingly.
- A closure function can further be classified into pure and impure functions, depending on the type of the free variable.
- If we give the free variable a type var then the variable tends to change the value any time throughout the entire code and thus may result in changing the value of the closure function.
- Thus this closure is a impure function.
- On the other-hand if we declare the free variable of the type val then the value of the variable remains constant and thus making the closure function a pure one

```
object closureeg{
    // Main method
    def main(args: Array[String])
    {
        println( "Final_Sum(1) value = " + sum(1))
        println( "Final_Sum(2) value = " + sum(2))
        println( "Final_Sum(3) value = " + sum(3))
    }
      var a = 4

    // define closure function
    val sum = (b:Int) => b + a
}
```