Functions & Closures

Principles of Programming Languages

Scala Function

A Scala method is a part of a class that has a name and a signature. A function in Scala is a complete object that can be assigned to a variable.

```
def function_name ([parameter_list]) : [return_type] = {

  // function body

}
```

If the user will not use the equals sign and body then implicitly method is declared abstract.

Function without parameters

- To define a function in Scala, use the def keyword followed by the method name and the method body.

```
object Main {
  def main(args: Array[String]) {
      def hello() = {"Hello World!"}
      println(hello );
  }
}
```

We can invoke this function using either hello() or hello.

- Other alternatives

```
def hello():String = {"Hello World!"}
```

```
def hello() = "Hello World!"
```

```
def hello = "Hello World!"
```

Function with Parameters

- The body of the functions are expressions, where the final line becomes the return value of the function.

```
def square (i:Int) = {i*i}
```

```
object Main {
  def main(args: Array[String]) {
      def square (i:Int) = {i*i}
      println(square(2) );
  }
}
```

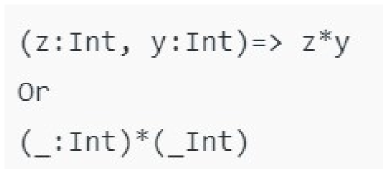
- Multiple parameters are separated by commas

```
object Main {
  def main(args: Array[String]) {
      def add(x: Int, y: Int): Int = { x + y }

      println(add(5, 5) );
  }
}
```

Anonymous Function

- A function literal is also called an anonymous function.
- Syntax:-



```
(z:Int, y:Int)=> z*y
Or
(_:Int)*(_Int)
```

- **=>** 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.
- _ character is known as a wildcard is a shorthand way to represent a parameter who appears only once in the anonymous function.

Function Literals

- A function literal starts with a parenthesized comma-separated list of arguments followed by an arrow and the body of the function.

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

- A function literal is instantiated into objects called function values.
- Because the function value is an object, it could be stored in a variable and it could be invoked using the parentheses function-call.

```
object Main extends App {
   val add = (a: Int, b: Int) => a + b
   println(add(1, 2));
}
```

Partially Applied Function

- When all the parameters are passed to the function we have fully applied the function to all the parameters.

```
val add = (x: Int, y: Int) => x + y
add(1,2)
```

- But when we give only a subset of the parameters to the function, the result of the expression is a partially applied function.

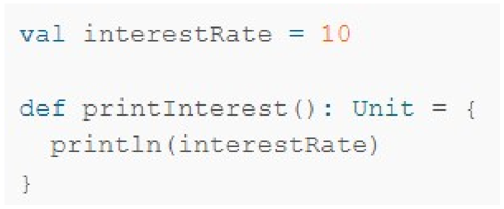
```
val partiallyAdd = add(1, _:Int)
```

- When you give partiallyAdd an Int value 2, you get the sum of the Int number passed into the add and partiallyAdd functions

```
partiallyAdd(2)
```

Using closures

- You want to pass a function around like a variable, and while doing so, you want that function to be able to refer to one or more fields that were in the same scope as the function when it was declared.
- The different between a normal function and a closure is that a closure is dependent on one or more free variables.
- The concept of a free variable comes from mathematics and functional programming theory. They refer to variables which are used locally but enclosed in some scope such as a class of method.



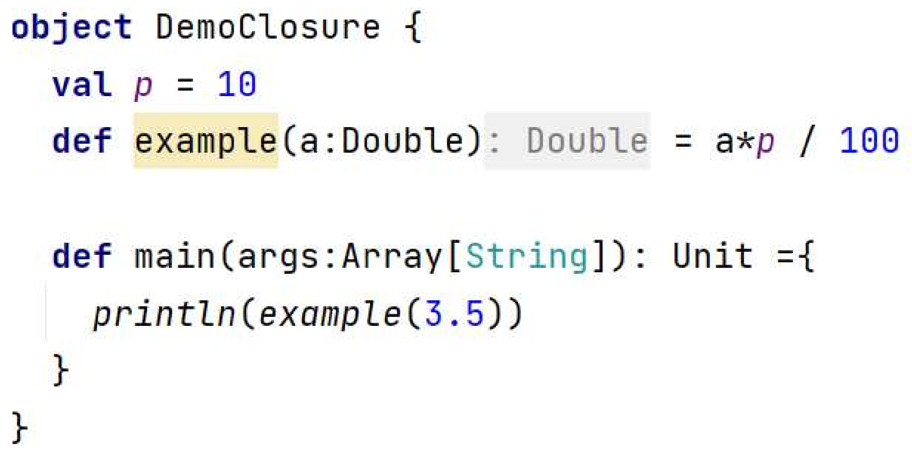
```
val interestRate = 10

def printInterest(): Unit = {
  println(interestRate)
}
```

The variable interest is defined for the class scope.

Free variable

- The free variables are defined outside of the Closure Function and is not included as a parameter of this function.
- 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.



```
object DemoClosure {
  val p = 10
  def example(a:Double): Double = a*p / 100

  def main(args:Array[String]): Unit ={
    println(example(3.5))
  }
}
```

Example - Multiplier

- Consider a multiplier function.

```
var y = 3
val multiplier = (x:Int) => x * y
```

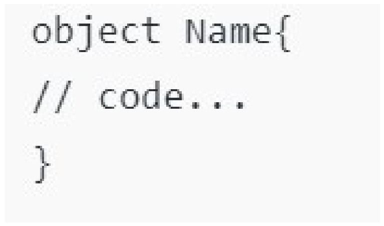
- Below code: y has a reference to a variable outside the function but in the enclosing scope.

```
object Main extends App {
    var y = 3
    val multiplier = (x:Int) => x * y
    println(multiplier(3))

}
```

Singleton Object

- Scala is more object oriented language than Java so, Scala does not contain any concept of static keyword.
- Instead of static keyword Scala has singleton object.
- A Singleton object is an object which defines a single object of a class.
- A singleton object provides an entry point to your program execution.



```
object Name{
// code...
}
```

Properties of Singleton Object

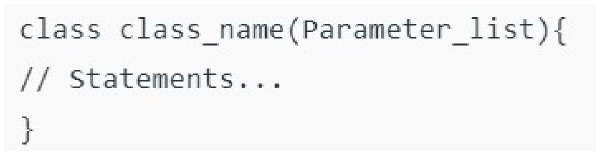
- The method in the singleton object is globally accessible.
- You are not allowed to create an instance of singleton object.
- You are not allowed to pass parameter in the primary constructor of singleton object.
- In Scala, a singleton object can extend class and traits.
- In Scala, a main method is always present in singleton object.
- The method in the singleton object is accessed with the name of the object(just like calling static method in Java), so there is no need to create an object to access this method.

Scala Constructors

- Constructors are used to initializing the object’s state.
- Like methods, a constructor also contains a collection of statements(i.e., instructions) that are executed at the time of Object creation.
- Scala supports two types of constructors:
  - Primary Constructor
  - Auxiliary Constructor

Primary Constructor

- When Scala program contains only one constructor, then that constructor is known as a primary constructor.
- The primary constructor and the class share the same body, means we need not to create a constructor explicitly.

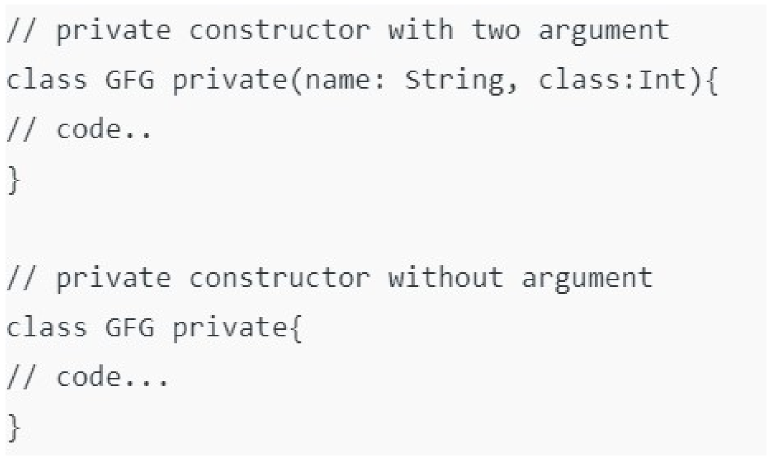


```
class class_name(Parameter_list){
// Statements...
}
```

- The primary constructor may contain zero or more parameters.
- If we do not create a constructor in the Scala program, then the compiler will automatically create a primary constructor when we create an object of your class, this constructor is known as a default primary constructor. It does not contain any parameters.

Primary Constructor

- In Scala, only a primary constructor is allowed to invoke a superclass constructor.
- In Scala, we are allowed to make a primary constructor private by using a private keyword in between the class name and the constructor parameter-list.



```
// private constructor with two argument
class GFG private(name: String, class:Int){
// code..
}

// private constructor without argument
class GFG private{
// code...
}
```

Auxiliary Constructor

- In a Scala program, the constructors other than the primary constructor are known as auxiliary constructors.
- We are allowed to create any number of auxiliary constructors in our program, but a program contains only one primary constructor.

```
def this(......)
```

Auxiliary Constructor

- In a single program, we are allowed to create multiple auxiliary constructors, but they have different signatures or parameter-lists.
- Every auxiliary constructor must call one of the previously defined constructors.
- The invoke constructor may be a primary or another auxiliary constructor that comes textually before the calling constructor.
- The first statement of the auxiliary constructor must contain the constructor call using this.