SCALA

Methods

```
package org.tech.scalaprograms

object methodfuncs5 {
        def main(args:Array[String]):Unit =
                {
// Methods, accept/returns arguments or not (Unit), overloading , currying , default params
// Functions -> Ananymous, lambda, literals, value functions - quick functionalities
// without considering reuability accross , cant have return keyword

/* Methods in Scala
   Syntax:
     def functionName(parameters : typeofparameters) : returntypeoffunction =
     {
                                        statements to be executed
                }
* */
                // FYI, the method add1/2/3…. Are declare below.
                //function called with arguments inside println function.
                        println(add1(100,200));
                        println(add2(100,300));
                        println(add3(100,400));
                        println(add4(20,3));
                        println(add5(200,500))
                        println("add5 curly braces is optional if there is no more than 1 expression")
                        println(add6(100));
                        println(add6("tech"));
                        add8();

                        //Function call with named arguments
                        add8(b = 20,a = 10);
                        add8;
                        add9(15)(30);

                        //Anonymous function or Lambda function or Function literal
                        println(add10(25,40));

                        println(sub(20,50))
                        println(add11(50,10))

                }

        // no return values
        def add1(a:Int,b:Int):Unit =
```

```
		{
			println("add1 No return values")
			val c = a + b;
			println(c)
			//return c;
		}

	// return an expression
	def add2 (a:Int,b:Int):Int =
		{
			println("add2 Return an expression")
			return {
				val x=a
				a + b
			       };
		}

	//return keyword optional
	def add3(a:Int,b:Int):Int =
		{
			println("add3 Return keyword in optional")
			a + b;

		}


	//return datatype is optional
def add4(a:Int,b:Int)=
{
println("add4 Return datatype is optional")
val x  = a.toFloat/b.toFloat;
println(x)
//return x
//x
}

	//for single statement curly braces are optional
	def add5(a:Int,b:Int) = a + b
```

Method over loading:

Method overloading is,

- Same method names with different arguments.

Consider when we need to create a method in a class which will be used by different users.
Different kind of users should be able to access the same method for different functionalities
But the name should be same.

For example,
Lets consider we have a requirement to show the project synopsis to different team.

- We create a method “projInfo” with an argument high_level_det:string to give it to non tech team . def projInfo(highdet:string)
- For a tech team we create a method with same name “projInfo” with different aruguments such

as high_level_det , tech_det. Def projInfo(highdet:String, techdet:String)

- For a accounting team, def projInfo(highdet:String,techdet:String,billingamt:doubt)

So all the team members will access “projInfo” to get the data respective to their inputs.

```
                                // Overloading
def add6 (a:Int):Int =
{
println("add6 Function overloading")
return a + 10
}

def add6 (a:Any):Any =
{
println("add6 Function overloading with any")
return a + a.toString()
}

        // Overloading
def add6(a:String):String =
{
println("add6 Function Type overloading")
return "Hello " + a;
}


        //Parameter example with default value

def add8(a:Int=30,b:Int=10) =
{
println("add7 Example for Default Arguments")
println(a + b)
}



        // Method with no arguments
def add8 =
{
println("add8 with no arguments")
println(100 + 50)
}

        //add6(20,30);

        //currying
        def add9(a:Int) (b:Int):(Int,Int) =
                {
                println("Currying method with multiple input and output arguments")
                val c = a + b;
```

```
                println(c)
                return (c,b);
                }
//add9(10)(20);

        //store function in a variable
        // Anonymous function or Lambda function or Function literal

val add10 = (a:Int,b:Int) =>
        {
                println("Anonymous add10 function call, cant have return, only last expression returns")
                a - b
                a + b
        }


 //Also we can define the same function in the short form
        var sub = (a:Int,b:Int) => a - b

        //Also we can define the same function in the short form
        var add11 = (_:Float) + (_:Float)
```

Pattern matching:

```
package org.tech.scalaprograms

object patternmatch6 {

 /*Pattern matching is a way of checking the given sequence of tokens for the presence of the
  * specific pattern.
  * It is a technique for checking a value against a pattern.
 */

 def main(args:Array[String]):Unit =
 {

   /*Exception handling is a mechanism which is used to handle abnormal conditions.
 * Avoid termination of your program unexpectedly.
*/

//here is the example for try/catch/finally.
//among try/catch/finaly, the logic inside try will be executed by default .
//if any exception needs to be handled, that logic will be placed inside catch.. the logic inside catch will be
//executed only when there is any corresponding exceptional errors occurred in try block logic.
//the logic finally will be executed at last by default..
```

```
  try
  {

      val y = 10/1
      println("first statement executed and result is : " + y);
      val z=Array(10,20,30);
      //z(2);

      //val x = 10/0
      println("All statements executed");
      println(y)


  }
  catch
  {
     case ex: java.lang.Exception =>
     {
        println("Some exception occured")
        println("I will be executed when error occured");
        println("calling test method with param as 2")
        println(testmatch(2));
     }
     case ex: java.lang.ArrayIndexOutOfBoundsException =>
      {
        println("Array index should be given properly")
        println("I will be executed when error occured");
        println("calling test method with param as 1")
        println(testmatch(1));
      }


  }
  finally
  {
    println("I will be executed at any cost");
    println("calling test method with param as 1")
    println(testmatch(1));
    //println(testcaseconditional(x=11))
  }


 }

// method containing match keyword
// its similar to SWITCH statement in java or EVALUATE statement in cobol.
//if the input value matches the value specified in the case statement, corresponding logic written inside
//the case statement will be executed.
//In the below example, if the input is passed as 0 thru x, then “Hello, Techies” will be printed.

def testmatch(x:Int) = x match
```

```
  {
    // if value of x is 0,
    // this case will be executed
    case 0 => "Hello, Techies"

    // if value of x is 1,
    // this case will be executed
    case 1 => "Are you learning Scala?"

    // if x doesnt match any sequence,
    // then this case will be executed
    case _ => "Good Luck!!"
  }

  //similarly the matching variable can be as, one of the input.. here op is the matching variable and its one
//of the input for the method.
//here the condition can be as OR condition.
 def calcuator(a:Int,b:Int,op:String):Any =
     op match
      {
       case "add" | "addition" =>
        {
         println("Add Numbers")
         a + b
        }
       case "sub" | "subtract" =>
       {
        println("Sub Numbers")
        a - b
       }
       case "mul" | "multiply" =>
       {
        println("Multiply Numbers")
        a * b
       }
       case _ =>
       {
        println("Operation Not matched")
        "No Match"
       }
      }


 val a = 100
 val b = 100

// method containing match keyword
// here based on the input match, we determine a condition.
  def testcaseconditional(x:Int) = x match
  {
```

```
    // if value of x is 0,
    // this case will be executed
    case i if (x == 0)  => {if (a == b) {println(a)}}

    // if value of x is 1,
    // this case will be executed
    case j if x > 0 & x < 10 => {if (a != b) {println(b)}}

    // if x doesnt match any sequence,
    // then this case will be executed
    case _ => "Good Luck!!"
  }

  }
```

Collection types:

```
package org.tech.scalaprograms

//to work with array and array buffer we need to import below classes.
//both array and array buffer are mutable. Array is of fixed length. Array Buffer is resizable
import scala.collection.mutable.ArrayBuffer
import scala.collection.mutable.Seq
import Array._

object collections7 {

 /*
  * Scala has rich set of collection library
  * Collection are the containers that holds elements

  * 1. Seq - linear collection of element may be indexed(array) or linear list(list)
  * 2. Map - contains a collection of key value pairs
  * 3. Set - collection that contains no duplicate values

  */
 def main(args:Array[String]):Unit =
 {
  /*
   * Seq -
   *                          - mutable Array
   *                          - immutable List
   *
   * */

    //LIST
    // Also we can define as
    var s2 = Seq(10,20,30,40,50)
// or var s2 = List(10,20,30,40,50)
```

```
//if we declare a variable as Seq, by default it will be created as list.
// while accessing the data, it should be based on offset value.. starts from zero.
      //s2(4) =50
      println("This shows Seq is mutable " + s2(4))

//ARRAY
      //An array is sequential and is of a fixed size.
      //Array - Array in scala is sequential, fixed in size and mutable

      //Declare Array with 1 element with type of Int

      //val ar = Array(Array(5,"hi"),Array(6,"Hello"))
      val ar = Array[Float](5);
      println("result of Array[Int](5) is : " + ar(0))

      //or

      val ar1 = Array(1,2,3,4,5)

      println("Fourth element of Array(1,2,3,4,5) is : " + ar1(3))

      //Creating array with range
      //var ar2 = range(start_value, end_value,incremental_value)
      val ar2= range(2,15,2)
      println("Second element of range(2,15,2) is : " + ar2(1))
      val ar3 = range(15,2,-2)
     println("Second element of range(15,2,-2) " + ar3(1))
      println(ar2(1));
      println(ar1.length)

      ar1.sorted.take(3).foreach(println)
       ar1.tail
       println(ar1.length)
       println(ar1.isEmpty)
       ar1.sorted
       ar1(2)=10;

       ar1(3)=100;

 /*
  * A list of Scala Collections is much like a Scala array.
  * Except that, it is immutable and represents a linked list.
  * An Array, on the other hand, is flat and mutable.
 */

      val lst = List(10,20,30,40)

      val fruits = List("apples", "oranges", "mangoes")

     val g = fruits(0)
```

```
    var list = List(1,8,5,6,9,58,23,15)
    //var list11 = List(List(1,"a",100000),List(2,"b",200000))

    //Access value from the list
    //list(2) = 10
    val lstval = list(0)

    //Merging 2 list
    var list3 = list ++ lst
    //or
    var list4 = list ::: lst

    println(list3)
    println(list4)

    println(list.contains(2))

    println(list4.head)
    list4.tail.foreach(println)
    println(list4.length)
    println(list4.isEmpty)
    list4.sorted.foreach(println)
    list4.reverse.foreach(println)


/*
  Tuples are immutable, contains fixed length of different type elements
 */

    val emp = (101,"Karthik",200.00)
    val empid = emp._1
    val empname = emp._2
    val empsal = emp._3

    val emp1 = (101,"Karthik",200.00,("New Street","Chennai","TN"))

    val empcity = emp1._4._2


    //// ARRAY PROGRAMS

    val ar222=Array[Int]();
      if (ar222.isEmpty)
      {
        println("Array is empty");
      }

 //1. write a program to create an Int array with 5 different value and sum all the values


/*
```

```
 * A Map in Scala is a collection of key-value pairs, and is also called a hash table.
 * We can use a key to access a value.
 * Keys are unique and values may be in common
 * Map is by default immutable
 * */

   var m = Map("Mani" -> 10000,"Karthik" -> 20000)

   //Add an element
   m += ("Raj" -> 30000)

   //Remove an element
   m -= ("Mani")

   //println(m)
   var m2 = Map.empty[String,String]

    //Add multiple Map elements

    m2 += ("1"->"A","2"->"B")
   println(m2)

   //Immutable map, doesn't allow to modify but allows to add or remove by recreating

   var immutablemap = scala.collection.immutable.Map(1 -> "Alto",2 -> "Swift")

   immutablemap -= (2)

   //Below update is not possible
   //immutablemap(1)="Alto k10"

   var mutablemap = scala.collection.mutable.Map(1 -> "Alto",2 -> "Swift")
   mutablemap(1)="Alto k10"

   m.keys
   m.values
   m.isEmpty

   val lst1 = m.toList
   val arr = m.toArray


/* A Scala Set is a collection that won't take duplicates.
 * By default, Scala uses immutable sets.
* */

   var mutablegames = scala.collection.mutable.Set("Cricket","VollyBall","BaseBall","Hockey")

   println("Due to mutable, I am able to modify")
   mutablegames.add("Chess");
   mutablegames.add("Hockey");
```

```
    mutablegames.remove("Cricket");
    println(mutablegames)

    var games = scala.collection.immutable.Set("Cricket","Football","Hockey","Golf","Cricket","Football")
    println(games)

    println("Due to Immutable, I am not able to modify, uncomment the below code and check")
    //games.add("Tennis");
    //games.remove("Cricket");

    println("Either mutable or immutable, I can reassign if its of type var");
    //val games = scala.collection.immutable.Set("Cricket","Football","Hockey","Golf","Cricket","Football")

    games += "Tennis"
    games -= "Cricket"

    println(games);
    println(games.head)           // Returns first element present in the set
    println(games.tail)       // Returns all elements except first element.
    println(games.isEmpty)        // Returns either true or false

    println(games.max)
    println(games.min)
    println(games.contains("Tennis"))
    println(games ++ mutablegames);
    println(games.intersect(mutablegames))
    println(games.union(mutablegames))
    println("Set difference");
    println(games.diff(mutablegames))

  }
}
```

Higher-Order Methods:

```
package org.tech.scalaprograms

object highordcaseclassclosure8 {
case class emp(id: Int, name: String, address: String)
  def main(args:Array[String])
  {

/*   Higher-Order Methods - scala treats functions/methods as a variable, hence it can be passed
 *   as a param for another function/method
A method that takes a function as an input parameter is called a higher-order method. Similarly, a
highorder function is a function that takes another function as input. Higher-order methods and
functions help reduce code duplication. In addition, they help you write concise code. The following
example shows a simple higher-order function.*/
```

```
def bonus(a:Int):Double = ((a*1.5))
val salaries = Array(20000, 70000, 40000)
val normalmethod = salaries.map (a=>(a*1.5))
val higherordermethod = salaries.map(bonus)
println(higherordermethod(0));

/*Case Classes
A case class is a class with a case modifier. An example is shown next.
All input parameters defined implicitly treated as Val, Useful for immutable objects and pattern
matching , Creates Factory method automatically*/

val request = emp(1, "Sam", "1, castle point blvd, nj")
println(request.name)
//request.address
```

Closures:

```
/*Closures
A closure is a function, whose return value depends on the value of one or
more variables declared outside this function.*/

var bonuspercent = .10

def bonus1(i:Int) =
{i+(i * bonuspercent)}

println( "Bonus value is = " + bonus1(100) )

    }
}
```

Few other SCALA methods:

```
package org.tech.scalaprograms

object fewscalamethods9 {
  def main (args:Array[String])
{
     //Map - Evaluates a function over each element in the list, returning a list with the same number of
elements.

   val lst = List(1,2,3,4,5,6,7,8,9,10)
   lst.map( x => x*2 )
   println(lst)

   //Filter - Removes any elements where the function you pass in evaluates to false.
   lst.filter((i: Int) => i % 2 == 0)

   //write a program to find prime numbers in the list
```

```
  //foreach is like map but returns nothing.
  lst.foreach(x => println(x))

  val Icecreamprice = List(25,13,45,34,25)
  val total = Icecreamprice.reduce((a,b) => (a + b))

  val donutPrices: List[Double] = List(1.5, 2.0, 2.5)
  val sum: Double = donutPrices.reduce(_ + _)

  val prices: Seq[Double] = Seq(1.5, 2.0, 2.5)
  println(s"Icecream prices = $prices")

  println("\nHow to sum all the icecream prices using fold function")
  val sum1 = prices.fold(0.0)(_ + _)
  println(s"Sum = $sum1")

   //flattern
  val lst1 = List(List(1,2,3,4),List(5,6),List(7,8)).flatten
  lst1.foreach(println)

  }
}
```

OOPS:

```
package org.tech.scalaprograms

/* OOPS
 * SPARK DEPENDENT
    which must be known for Spark Develpment
 * PACKAGE
   its collection of methods/functions/objects/variables etc .
 *  CLASS
   Its template which has set of methods/functions/variable which is of particular functionality
 * OBJECT
   singleton obj/class – its similar to class. Only difference is its naturally instantiated. No need to create instance separately.
   Normal obj – it's the name which is used to point out the instance memory created for its corresponding class.
 * main or custom METHODs/FUNCTIONs
   Main method – it's the method inside a class which is the start point of execution. Only one main method per class is allowed.
   Customized  method – is the method created inside a class other than main method. Many methods can be created in a class based on different logics for each method.
 * CONSTRUCTOR
   Primary constructor- when an instance is create by constructor for a class, we pass arguments if the class supports input arguments.
   Auxiliary constructor – when the instance is create by contructor for a class, if we don't want to pass arguments even if the class supports input arguments, then that class should have an inbuilt input value defined using the "this" keyword . so that the value defined in the method using "this" keyword will be used inside that class.
This concept will be explained in the below example.
```

* CASE CLASS
It's a template which can accept constant values as input and it will remain immutable.

* HIGHER ORDER METHOD/FUNC
If a function/method is passed as an argument to another function/method, then its called higher order function/method.
For eg: **sc.foreach(println)**
Foreach is a function and println is another function.
* CLOSURE
* if the output of a method is dependent of a value declared outside that method, its closure.

* OOPS Concepts made simple with Scala program implementation:
*
* All these are Classes -> Class, singleton object (instantiated class),case class, abstract class, traits
*
* Polymorphism - Polymorphism means that a function type comes "in many forms".
* the type can have instances of many types.
* Example - Method with different type of arguments

```
Eg:     var a = 100
        Var b = 50
        Var c = "sub"
            def operations(a:Int,b:Int,c:String) :Int =
              { a match {
                    case "add" -> return (a+b)
                    case "sub" -> return (a-b)
                    case "div" -> return (a/b)
                    case _ -> return (a*b)
                          }
               }
```

The output of the method "operation" is dependent on the variables a,b and c declared outside that method.

*
* Overloading - Scala Method overloading is when one class has more than one method with the
* same name but different signature.
* This means that they may differ in the number of parameters, data types, or both.
* Example : Method with different number of arguments
*
* Abstraction - Abstraction is the process to hide the internal details
* and showing only the functionality.
* abstraction is achieved by using an abstract class.
* Example : Abstract class and Traits
*
* Inheritance -
* Inheritance is the process of inheriting the feature of the parent class
* Multiple Inheritance: In Multiple inheritance ,one class can have more than one superclass and inherit features from all parent classes.
* Scala does not support multiple inheritance with classes, but it can be achieved by traits.
* Example - Abstract Class and Traits
*
* Overriding - Scala overriding method provides your own implementation of it. When a class inherits
* from another, it may want to modify the definition for a method of the superclass or
* provide a new version of it.
* This is the concept of Scala method overriding and we use the 'override' modifier to implement this.
* Example : Method or vals override with different implementations
*

```
 * Encapsulation - Specify access specified/modifier for providing access control to the objects or
 * values
 * Example : private var a=100;
 *
 *  Companion Object - It is again a singleton object called as companion if we create the object in the same name of
the instantiating class
 *  using companion object we can able to access the encapsulated members of the class.
*/

 /*
 Class -
                    A class is a user defined blueprint or prototype from which objects are created.
                    It represents the set of properties or methods
 * */
//Primary Constructor
class bankclass(actype:String,intpct:Double) {

 //Auxilary Constructor
 def this(actype:String)
 {
    this(actype,0.0);
 }

 def cust(amt:Double):Double=
 {
  return {
    if (actype == "SB")
    amt+(amt*intpct)
    else
    amt
    }
 }
}

/*
 Singleton object is an object which is declared by using object keyword instead by class.
 No object is required to call methods declared inside singleton object.
 In scala, there is no static concept.
 So scala creates a singleton object to provide entry point for your program execution. * */
  object singletonbankobj{
   val welcome= "Singleton Object Initialized with his members"
    }

/*Object -
 An object is an instance of a class. Objects have states and behaviors.*/
//object can't have params
object bankclassobj
{

 def main(args:Array[String])
 {
 println(singletonbankobj.welcome);

 println("Primary constructor will be initialized");
  //Primary constructor will be initialized
 val bankclassobjinstance1= new bankclass("SB",8.5);
```

```
  println("Auxilary constructor will be initialized");
  //Auxilary constructor will be initialized
  val bankclassobjinstance2= new bankclass("CU");


  }
}

//Abstraction, Inheritance
//Trait can't have params
trait cardtrait
{
  def cardtype(ctype:String,withdrawlimit:Long):Int;
}

trait banktypetrait
{
  def banktype(btype:String):Int= btype match
  {
    case "Investment" => 10
    case "Retail" => 20
    case _ => 30
  }

  class bankclassinherit extends cardtrait with banktypetrait
  {
   def cardtype (ctype:String,withdrawlimit:Long):Int= ctype match
   {
     case "Credit" => 100000
    case "Debit" => 20000
    case _ => 10000
   }

   val btype=banktype("Investment");
  }

//Can pass arguments, only 1 abstract class can be extended, can have implementation or not.
abstract class cardtrait1(inputargispossible:Int)
{
  val argassign=inputargispossible;
  def cardtype(ctype:String,withdrawlimit:Long):Int;
}

abstract class  banktypetrait1
{
  def banktype(btype:String):Int= btype match
  {
    case "Investment" => 10
    case "Retail" => 20
    case _ => 30
  }
}

  class bankclassinherit1 extends cardtrait1(100) //with banktypetrait1
  {
```

```
  def cardtype (ctype:String,withdrawlimit:Long):Int= ctype match
  {
    case "Credit" => 100000
   case "Debit" => 20000
   case _ => 10000
  }

  val btype=banktype("Investment");
 }


/// Overriding, Companion Object, Encapsulation
   class bankclassobj
 {
    private def pvtmethod=
    {
      println("I am a private method, only inline or companion can access me");
    }

 private val pvtval=1;
 val FDMaturity = 5+pvtval;
 }

  //Companion object has same name of the class, which can access private variable
  object bankclassobj extends bankclassobj
  {
    val objbankclassobj=new bankclassobj;
    override val FDMaturity=7;
    println(objbankclassobj.pvtval);
    objbankclassobj.pvtmethod;
  }
```