Scala

- Scala is an *object-oriented* and *functional programming* language.
- It is a strong static type language.
- In scala, everything is an object whether it is a function or a number. It does not have concept of primitive data.
- Scala is influenced by Java.
- File extension of scala source file may be either .scala or .sc.
- You can create any kind of application like web application, enterprise application, mobile application, desktop based application etc.
- Scala programs can convert to bytecodes and can run on the JVM(Java Virtual Machine). Scala stands for Scalable language.

Features of scala

- **Object- Oriented**: Every value in Scala is an object so it is a purely object-oriented programming language.
- **Functional:** It is also a functional programming language as every function is a value and every value is an object. It provides the support for the high-order functions, nested functions, anonymous functions etc.
- **Statically Typed**: The process of verifying and enforcing the constraints of types is done at compile time in Scala.
- **Extensible:** New language constructs can be added to Scala in form of libraries.
- **Concurrent & Synchronize Processing:** Scala allows the user to write the codes in an immutable manner that makes it easy to apply the parallelism(Synchronize) and concurrency.

- **Type inference**
  - Don't require to mention data type and function return type explicitly.
- **Singleton object**

  In Scala, there are no static variables or methods. Scala uses singleton object, which is essentially class with only one object in the source file. Singleton object is declared by using object instead of class keyword.
- **Immutability**

  NO modification
- **Lazy computation**

  Scala evaluates expressions only when they are required
- **Concurrency control**

  Standard library support for concurrency control
- **String interpolation**

  Scala offers a new mechanism to create strings from your data. It is called string interpolation.
- **Rich collection set**

  rich set of collection library to support data collection.

Simple scala program

```
object ScalaExample{
   def main(args:Array[String]){
      println "Hello Scala”;
   }
}
```

Command to compile this code is: **scalac ScalaExample.scala**
Command to execute the compiled code is: **scala ScalaExample**
After executing code it yields the following output.

Output:

```
Hello Scala
```

Functional approach to write scala code

```
def scalaExample{
    println("Hello Scala")
}
```

Scala Variables and Data Types

You can create mutable and immutable variable in scala.

Mutable Variable

You can create mutable variable using var keyword. It allows you to change value after declaration of variable.

Example

```
var data = 100
```

Data = 101 // It works, No error.

In the above code, var is a keyword and data is a variable name. It contains an integer value 100. Scala is a type infers language so you dont need to specify data type explicitly. You can also mention data type of variable explicitly as we have used in below.

var data:Int = 100

Immutable Variable

val data = 100

data = 101 // Error: reassignment to val

The above code throws an error because we have changed content of immutable variable, which is not allowed. So if you want to change content then it is advisable to use var instead of val.

Operators

- Arithmetic(+,-,*,/,%,**)
- Relational(==,!=,>,<,>=,<=)
- Logical(&&,||,!)
- Assignment operators(=,+=,-=,*= etc.)
- Bitwise operators(&,|,<<,>>, etc)

```
// Scala program to demonstrate
// the Arithmetic Operators
object Arithop
{
def main(args: Array[String]) {
        // variables
        var a = 50;
        var b = 30;
        // Addition
        println("Addition of a + b = " + (a + b));
        // Subtraction
        println("Subtraction of a - b = " + (a - b));
        // Multiplication
        println("Multiplication of a * b = " + (a * b));
        // Division
        println("Division of a / b = " + (a / b));
        // Modulus
        println("Modulus of a % b = " + (a % b));}
}
```

Data Types in Scala:

Data types in scala are much similar to java in terms of their storage, length, except that in scala there is no concept of primitive data types every type is an object and starts with capital letter. A table of data types is given below.

Data Type	Default Value	Size
Boolean	False	True or false
Byte	0	8 bit signed value (-2^7 to 2^7-1)
Short	0	16 bit signed value(-2^{15} to $2^{15}-1$)
Char	'\u0000'	16 bit unsigned Unicode character(0 to $2^{16}-1$)
Int	0	32 bit signed value(-2^{31} to $2^{31}-1$)
Long	0L	64 bit signed value(-2^{63} to $2^{63}-1$)
Float	0.0F	32 bit IEEE 754 single-precision float
Double	0.0D	64 bit IEEE 754 double-precision float
String	Null	A sequence of characters

Scala Conditional Expressions

- If statement
- If-else statement
- Nested if-else statement
- If-else-if ladder statement

Example:

```
var age:Int = 20;
if(age > 18){
    println ("Age is greate than 18")
}
```

If else Example

```
var number:Int = 21
if(number%2==0){
    println("Even number")
}else{
    println("Odd number")
}
```

```
var number:Int = 85
if(number>=0 && number<50){
   println ("fail")
}
else if(number>=50 && number<60){
   println("D Grade")
}
else if(number>=60 && number<70){
   println("C Grade")
}
else if(number>=70 && number<80){
   println("B Grade")
}
else if(number>=80 && number<90){
   println("A Grade")
}
else if(number>=90 && number<=100){
   println("A+ Grade")
}
else println ("Invalid")
```

In scala, you can assign if statement result to a function. Scala does not have ternary operator concept like C/C++ but provides more powerful if which can return value.

```
object MainObject {
   def main(args: Array[String]) {
      val result = checkIt(-10)
      println (result)
   }
    def checkIt (a:Int)  =  if (a >= 0) 1 else -1    // Passing a if expression value to function
}
```

Scala - for Loops

- Control structure that allows you to execute set of statements certain number of times.
- Syntax − for loop with ranges

The simplest syntax of for loop with ranges in Scala is −

```
for( var x <- Range ){
   statement(s);
}
```

<- left arrow operator called generator(because it's generating individual values from a range.)

Range could be a range of numbers and that is represented as **i to j** or sometime like **i until j**

Example

```
object Demo {
   def main(args: Array[String]) {
      var a = 0;

      // for loop execution with a
range
      for( a <- 1 to 10){
         println( "Value of a: " + a );
      }
   }
}
```

Output

```
value of a: 1
value of a: 2
value of a: 3
value of a: 4
value of a: 5
value of a: 6
value of a: 7
value of a: 8
value of a: 9
value of a: 10
```

```
// Scala program of while loop

// Creating object
object GFG
{
	// Main method
	def main(args: Array[String])
	{
		// variable declaration (assigning 5 to a)
		var a = 5
	// loop execution
		while (a > 0)
		{
			println("a is : " + a)
			a = a - 1;
		}
	}
}
```

```
// Scala program of do-while loop
// Creating object
object GFG
{
	// Main method
	def main(args: Array[String])
	{
		// variable declaration (assigning 5 to a)
		var a = 5;
		// loop execution
		do
		{
			println("a is : " + a);
			a = a - 1;
		}
		while (a > 0);
	}
}
```

Arrays in Scala

Syntax:

var z:Array[String] = new Array[String](3)

or

var z = new Array[String](3)

Here, z is declared as an array of Strings that may hold up to three elements. Values can be assigned to individual elements or get access to individual elements.

Example:

z(0) = "Zara"; z(1) = "Nuha"; z(4/2) = "Ayan"

Another way of declaring array:

```
var z = Array("Zara", "Nuha", "Ayan")
```

```
object Demo {
  def main(args: Array[String]) {
    var myList = Array(1.9, 2.9, 3.4, 3.5)

    // Print all the array elements
    for ( x <- myList ) {
      println( x )
    }

    // Summing all elements
    var total = 0.0;

    for ( i <- 0 to (myList.length - 1)) {
      total += myList(i);
    }
    println("Total is " + total);
```

```
// Finding the largest element
    var max = myList(0);

    for ( i <- 1 to (myList.length - 1) ) {
      if (myList(i) > max) max = myList(i);
    }

    println("Max is " + max);
  }
}
```

Multi dimensional arrays

Syntax:

```
myMatrix = ofDim[Int](3,3)
```

```
import Array._
object Demo {
   def main(args: Array[String]) {
      var myMatrix = ofDim[Int](3,3)
      // build a matrix
      for (i <- 0 to 2) {
         for ( j <- 0 to 2) {
            myMatrix(i)(j) = j;
         }
      }
           // Print two dimensional array
      for (i <- 0 to 2) {
         for ( j <- 0 to 2) {
            print(" " + myMatrix(i)(j));
         }
         println();
      }
   }
}
```

Scala - Functions

Function Declarations

- A Scala function declaration has the following form :

```
def functionName ([list of parameters]) : [return type]
```

Function Definitions

A Scala function definition has the following form −

Syntax:

```
def functionName ([list of parameters]) : [return type] = {
   function body
   return [expr]
}
```

Example:

```
object add {
   def addInt( a:Int, b:Int ) : Int = {
      var sum:Int = 0
      sum = a + b
      return sum
   }
}
```

```
object Demo {
   def main(args: Array[String]) {
      println( "Returned Value : " + addInt(5,7) );
   }

   def addInt( a:Int, b:Int ) : Int = {
      var sum:Int = 0
      sum = a + b

      return sum
   }
}
```

Pattern matching/Switch in scala

```
import scala.util.Random
val x: Int = Random.nextInt(10)
 val description=x match {
case 0 => "zero"
case 1 => "one“
 case 2 => "two“
 case _ => "other“
}
Println(description)
```

Maps in scala

- Map is a collection of key-value pairs. In other words, it is similar to dictionary.
- Keys are always unique while values need not be unique.
- Key-value pairs can have any data type. However, data type once used for any key and value must be consistent throughout.
- Maps are classified into two types: mutable and immutable. By default Scala uses immutable Map.
- In order to use mutable Map, we must import scala.collection.mutable.Map class explicitly.

- You can construct a map as :

  val scores = Map("Alice" -> 10, "Bob" -> 3, "Cindy" -> 8)

This constructs an immutable Map[String, Int] whose contents can't be changed.

- If you want a mutable map, use

  var scores = scala.collection.mutable.Map("Alice" -> 10, "Bob" -> 3, "Cindy" -> 8)

- You can also create an empty mutable map initially and add elements to it later:

  Var scores = new scala.collection.mutable.HashMap[String, Int]

Or

  var scores =collection.mutable.HashMap[String, Int]()

  Scores+=("Tom"->25)

Accessing Map Values

- In Scala, the analogy between functions and maps is particularly close because you use the () notation to look up key values.

```
val bobsScore = scores("Bob")
```

- If the map doesn’t contain a value for the requested key, an exception is thrown. To check whether there is a key with the given value, call the contains method:

```
val bobsScore = if (scores.contains("Bob")) scores("Bob") else 0
```

- Since this call combination is so common, there is a shortcut:

```
val [redacted]= scores.getOrElse("Bob", 0)
```

Updating Map Value

- In a mutable map, you can update a map value, or add a new one, with a () to the left of an = sign:

```
scores("Bob") = 10
// Updates the existing value for the key "Bob" (assuming scores is mutable)
scores("Fred") = 7
// Adds a new key/value pair to scores (assuming it is mutable)
```

Alternatively, you can use the += operation to add multiple associations:

```
scores += ("Bob" -> 10, "Fred" -> 7)
```

- To remove a key and its associated value, use the -= operator:

```
scores -= "Alice"
```

- You can't update an immutable map, but you can do something that's just as useful—obtain a new map that has the desired update:

```
val newScores = scores + ("Bob" -> 10, "Fred" -> 7) // New map with update
```

- The newScores map contains the same associations as scores, except that "Bob" has been updated and "Fred" added.
- Similarly, to remove a key from an immutable map, use the - operator to obtain a new map without the key:

```
scores = scores - "Alice"
```

Iterating over Maps

- The following simple loop iterates over all key/value pairs of a map:

  ```
  for ((k, v) <- map) process k and v
  ```

If for some reason you just want to visit the keys or values, use the keySet and values methods, as you would in Java. The values method returns an Iterable that you can use in a for loop.

```
scores.keySet // A set such as Set("Bob", "Cindy", "Fred", "Alice")
for (v <- scores.values)
println(v) // Prints 10 8 7 10
```

To reverse a map—that is, switch keys and values—use

```
for ((k, v) <- map) yield (v, k)
```

Sorted Maps

- When working with a map, you need to choose an implementation—a hash table or a balanced tree. By default, Scala gives you a hash table.
- To get an immutable tree map instead of a hash map, use

```
val scores = scala.collection.immutable.SortedMap("Alice" -> 10,
"Fred" -> 7, "Bob" -> 3, "Cindy" -> 8)
```

- Unfortunately, there is (as of Scala 2.9) no mutable tree map

Interoperating with Java

If you get a Java map from calling a Java method, you may want to convert it to a Scala map so that you can use the pleasant Scala map API. This is also useful if you want to work with a mutable tree map, which Scala doesn't provide.

Simply add an import statement:

```
import scala.collection.JavaConversions.mapAsScalaMap
```

Then trigger the conversion by specifying the Scala map type:

```
val scores: scala.collection.mutable.Map[String, Int] = new java.util.TreeMap[String, Int]
```

Tuples

- Maps are collections of key/value pairs. Pairs are the simplest case of tuples- aggregates of values of different types.
- A tuple value is formed by enclosing individual values in parentheses. For example,

  (1, 3.14, "Fred")

is a tuple of type

  Tuple3[Int, Double, java.lang.String]

- which is also written as

  (Int, Double, java.lang.String)

- If you have a tuple, say,

  val t = (1, 3.14, "Fred")

- then you can access its components with the methods _1, _2, _3, for example:

  val second = t._2 // Sets second to 3.14

- Unlike array or string positions, the component positions of a tuple start with 1, not 0

Zipping

- One reason for using tuples is to bundle together values so that they can be processed together. This is commonly done with the zip method. For example,the code

```
val symbols = Array("<", "-", ">")
val counts = Array(2, 10, 2)
val pairs = symbols.zip(counts)
```

- yields an array of pairs

```
Array(("<", 2), ("-", 10), (">", 2))
```

- The pairs can then be processed together:

```
for ((s, n) <- pairs) Console.Print(s * n)
// Prints <<---------->>
```

Classes in scala

```
class Counter {
 private var value = 0 // You must initialize the field
 def increment() { value += 1 } // Methods are public by default
 def current() = value
}
```

In Scala, a class is not declared as public. A Scala source file can contain multiple classes, and all of them have public visibility. To use this class, you construct objects and invoke methods in the usual way:

```
val myCounter = new Counter // Or new Counter()
myCounter.increment()
println(myCounter.current)
```

You can call a parameterless method (such as current) with or without parentheses:

```
myCounter.current // OK
myCounter.current() // Also OK
```

Which form should you use? It is considered good style to use () for a mutator method (a method that changes the object state), and to drop the () for an accessor method (a method that does not change the object state).

example:

```
myCounter.increment() // Use () with mutator
println(myCounter.current) // Don’t use () with accessor
```

```
class Student {
 var name: String = _
 var id: Int = _
 var test1: Double = _
 var test2: Double = _
 var test3: Double = _
 def setName(studentName: String): Unit = {
  name = studentName }
 def setId(studentId: Int): Unit = {
  id = studentId }
 def setTestMarks(t1: Double, t2: Double, t3: Double): Unit = {
  test1 = t1
  test2 = t2
  test3 = t3   }
 def displayDetails(): Unit = {
  println(s"Student Name: $name")
  println(s"Student ID: $id")
  println(s"Test 1: $test1")
  println(s"Test 2: $test2")
  println(s"Test 3: $test3")  }
 def calculateAndDisplayAverage(): Unit = {
  val allTests = List(test1, test2, test3)
  val bestTwoTests = allTests.sorted.takeRight(2)
```

```
object StudentMain {
 def main(args: Array[String]): Unit = {
  val student1 = new Student
  student1.setName("John Doe")
  student1.setId(1)
  student1.setTestMarks(85.5, 92.0, 78.5)

  val student2 = new Student
  student2.setName("Jane Smith")
  student2.setId(2)
  student2.setTestMarks(90.0, 88.5, 95.0)

  // Display details and calculate average for each student
  println("Details for Student 1:")
  student1.displayDetails()
  student1.calculateAndDisplayAverage()

  println("\nDetails for Student 2:")
  student2.displayDetails()
  student2.calculateAndDisplayAverage()
 }
}
```

Scala constructors

- Used to initialize 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 our 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.

Syntax:

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

In the above syntax, the primary constructor and the class share the **same body** so, anything defined in the body of the class except method declaration is the part of the primary constructor

```
// Scala program to illustrate the  concept of primary constructor
class GFG(Aname: String, Cname: String, Particle: Int)
{
def display()
	{
		println("Author name: " + Aname);
		println("Chapter name: " + Cname);
		println("Total published articles:" + Particle);
	}
}
object Main
{
	def main(args: Array[String])
	{

		// Creating and initializing object of GFG class
		var obj = new GFG("Ankita", "Constructors", 145);
		obj.display();
	}
}
```

- The primary constructor may contain zero or more parameters.
- If we do not create a constructor in our 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

```
// Scala program to illustrate the  concept of default primary constructor
class GFG
{
        def display()
        {
                println("Welcome to RIT");
        }
}
object Main
{
def main(args: Array[String])
        {
                var obj = new GFG();
                obj.display();
        }
}
```

- If the parameters in the constructor parameter-list are declared using var, then the value of the fields may change. And Scala also generates getter and setter methods for that field.
- If the parameters in the constructor parameter-list are declared using val, then the value of the fields cannot change. And Scala also generates a getter method for that field.
- If the parameters in the constructor parameter-list are declared without using val or var, then the visibility of the field is very restricted. And Scala does not generate any getter and setter methods for that field.
- If the parameters in the constructor parameter-list are declared using private val or var, then it prevents from generating any getter and setter methods for that field. So, these fields can be accessed by the members of that class.
- 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

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.

Syntax:

def this(.....)

- 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**

```scala
/ /Scala program to illustrate the
// concept of Auxiliary Constructor


// Primary constructor
class GFG( Aname: String, Cname: String)
{
   var no: Int = 0;;
   def display()
   {
      println("Author name: " + Aname);
      println("Chapter name: " + Cname);
      println("Total number of articles: " + no);

   }

   // Auxiliary Constructor
   def this(Aname: String, Cname: String, no:Int)
   {

      // Invoking primary constructor
      this(Aname, Cname)
      this.no=no
   }
}
```

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

        // Creating object of GFG class
        var obj = new GFG("Anya", "Constructor", 34);
        obj.display();
    }
}
```

```
Output:
Author name: Anya
Chapter name: Constructor
Total number of articles: 34
```

Scala – Pass function as a parameter

- In Scala, we can pass a function as a parameter to another function or say method. For that we have to follow the below guidelines.
  - Define your method , including the function signature that it will accepts as a parameter.
  - Define the function with the exact signature that will be passed a method parameter.
  - Pass the function as a parameter to the methods.

```
object FuncAsParam {

  def main(args: Array[String]): Unit = {
    EmpDetails("Tom",30000,yearlySalary)
  }

  def EmpDetails(name:String,salary:Int, f:Int=>Int)= {
    print(name +"'s salary per anum in INR - "+ f(salary) )
  }

  def yearlySalary(sal:Int): Int ={
    sal * 12
  }
}
```