What is Scala?

Scala stands for Scalable Language. It is a multi-paradigm programming language. Scala language includes features of functional programming and object-oriented programming. It is a statically typed language. Its source code is compiled into bytecode and executed by Java virtual machine(JVM).

Scala is object-oriented

Scala is an object-oriented language. In Scala, every value is an object.

Scala execution

Scala uses Java Virtual Machine (JVM) to execute bytecode. Its code is compiled into bytecode and executed by Java Virtual Machine. So you need only JVM to start development with it. Scala can also use all java classes and allows us to create our custom class.

Why use Scala?

It is designed to grow with the demands of its user, from writing small scripts to building a massive system for data processing.

Scala is used in Data processing, distributed computing, and web development. It powers the data engineering infrastructure of many companies.

Who uses Scala?

Scala language is mostly used by Software engineers and Data Engineers. You'll see some data scientists using it with Apache Spark to process huge data.

How Scala is different from Java?

- Nested functions – It allows us to define a function inside another function.
- Closures – A function whose return value depends on variables declared outside it of function.
- Every value is an object.
- Every operation is a method call.

For example:

```
1   val sumX = 1.+(3)
2   val sumY = 1 + 3
```

Output of both is the same.

Popular frameworks of Scala

Let’s understand how we can set up a development environment for Scala. As we know, Scala uses JVM. So in order to execute a Scala program, you need to have java installed on your machine.

sudo apt-get update

Now type: **sudo apt-get install openjdk-8-jdk**

Scala Basics

Scala is very similar to Java Language, and both use Java Virtual Machine(JVM) to execute code. So, learning Scala would be super easy for you if you have a solid understanding of the Java language. But it’s not mandatory to know Java before learning the Scala Language.

Let’s write our first Scala program!

```
object HelloWorld {
  def main(args: Array[String]) {
    println("Hello world!")
  }
}
```

Output:

Note if you want to execute this using terminal, then follow the below steps.

Save this file as **HelloWorld.scala**

To compile and run, use the below commands:

\> scalac HelloWorld.scala

\> scala HelloWorld

Let’s understand the above code.

It can be noticed that I used the **object** keyword instead of class. Scala allows us to create a singleton class using the **object** keyword.

object – Scala doesn't use static keywords like Java, instead it allows us to create a singleton object.

def main(args: Array[String]) – main() method is compulsory for any Scala Program. Scala starts execution from here.

Case Sensitivity − It is case-sensitive.

Comments in Scala

This language supports single-line and multi-line comments. Multi-line could be nested.

```
1
2     def main(args: Array[String]) {
3
4       //println(f"my name is $name%s")
5       val name:String="Rhaul roy"
6
7       /*
8       println(f"my name is $name%s")
9       println(f"my name is $name%s")
10      */
11
12
13    }
```

Output: If you run above code, it will just create a string variable and will not produce any output.

The Scala Interpreter

The Scala interpreter, an alternative way to run Scala code. Type **scala** on your terminal to launch Scala interpreter.

In this Scala tutorial, I will not be using the interactive mode to execute any Scala code.

Scala Identifiers

Identifiers are nothing but names used for objects, classes, variables and methods. The important thing is, a keyword cannot be used as an identifier.

Scala has four types of identifiers.

Alphanumeric Identifiers

These kinds of identifiers start with a letter or an underscore, which can be followed by letters, digits, or underscores.

Note – The '$' character is a reserved keyword.

Valid identifiers example

sum, _count, __1_salary

Invalid illegal identifiers example

$sum, 1count, -index

Operator Identifiers

These kinds of operator identifiers consist of one or more operator characters.

Valid operator identifiers

+ ++ :::

Mixed Identifiers

It consists of an alphanumeric identifier, which is followed by an underscore and an operator identifier.

Valid mixed identifiers

unary_-

unary_- used as a method name defines a unary **–** operator. Scala supports unary prefix operators (-, +, ~, !) only.

Literal identifier

A literal identifier is an arbitrary string enclosed in backticks (` . . . `).

Example:

`xyz`

Scala Keywords

Keywords are nothing but reserved words in Scala. The following table shows the reserved words in Scala.

abstract	case	catch	class
def	do	else	extends
false	final	finally	for
forSome	if	implicit	import
lazy	match	new	Null
object	override	package	private
protected	return	sealed	super
this	throw	trait	Try
true	type	val	Var
while	with	yield	
_	:	=	=>

<-	<:	<%	>:
#	@		

Data Types

A data type is a categorization of data that tells the compiler which type of value a variable holds. For example, a variable may hold an integer value or float. Like java, Scala doesn't have any primitive data types. It comes with the standard numeric data types. In Scala, all data types are full-blown objects.

Basic data types example:

```
1
2     object Main {
3
4       def main(args: Array[String]) {
5
6           val a: Byte = 2
7           val b: Int = 444
8           val c: Long = 324324
9           val d: Short = 5
10          val e: Double = 2.0
11
12      }
13
    }
```

In the above example, **val** is a keyword which means that the value of variable **"a"** cannot be changed. Just after ":" data type of variable "a" has been mentioned.

If you don't mention data type. The compiler will automatically detect its data type.

val j =10 // defaults to Int

Scala basic literals

Integral Literals – The following are Integral literals:

02

21

0777L

Floating Point Literal – The following are floating point literals:

11.0

7.14159f

1.0e100

.1

Boolean Literals – Only **true** and **false** are members of Boolean.

Data types and its range

Below table shows data type and its possible values or range.

Data Type	**Possible Values**
Boolean	true or false
Byte	8 bit signed value. Range from -128 to 127
Short	16 bit signed value. Range -32768 to 32767
Int	32 bit signed value. Range -2147483648 to 2147483647
Long	64 bit signed value. -9223372036854775808 to 9223372036854775807

Float	32 bit IEEE 754 single-precision float
Double	64 bit IEEE 754 double-precision float
Char	16 bit unsigned Unicode character. Range from U+0000 to U+FFFF
String	A sequence of Char
Unit	Corresponds to no value
Null	null or empty reference
Nothing	The subtype of every other type; includes no values
Any	The supertype of any type; any object is of type *An*

Escape Sequences

Below is the escape sequence in Scala.

Escape Sequences	**Unicode**	**Description**
\b	\u0008	backspace BS
\t	\u0009	horizontal tab HT
\n	\u000c	formfeed FF

\f	\u000c	formfeed FF
\r	\u000d	carriage return CR
\”	\u0022	double quote “
\’	\u0027	single quote .
\\	\u005c	backslash \

String and Char

String is nothing but a sequence of characters.

Multi- line String in Scala

Example

“”” This is

multi-line

string”””

Variables in Scala

Variables are reserved memory locations where values are stored. Which can be referred later in the program.

Scala has two kinds of variables.

1. Immutable
2. Mutable

Immutable variables – These kinds of variables can’t be changed. It is declared using the Val keyword.

Example

val name:String=”Rahul”

Mutable variables – These kinds of variables can be changed. It is declared using **var** keyword.

Example

var name:String=”Rahul”

Data Type

Data type of any variable is mentioned just after the variable name. In this case, if you don’t mention the data type. The Scala compiler can detect the type of the variable based on the value assigned to it. This is called variable type inference.

Syntax

val VarName : DataType = [Initial Value]

or

var VarName : DataType = [Initial Value]

Example:

var sum :Int=0;

Here, sum is a variable and Int is a data type of the variable.

var sum =0;

Here, Scala compiler will automatically detect the type of sum variable. This feature is called variable type inference.

Scope of Variables

There are three types of scope in Scala-

Fields – These variables are accessible from every method in the object. And even from outside the object if the access modifier is not private for that variable. Access modifiers will be covered later in detail. Variables could be mutable or immutable depending upon the **var** and **Val** keywords.

Method Parameters – Whenever we call a method, we might pass a few values as parameters. These variables can only be accessed inside and outside the method if there is any reference to the object from outside the method else not. These variables are always mutable.

Local Variables – These kinds of variables are declared inside a method. And it can only be accessed within the method. These could be both mutable & immutable.

If else statement

If else is used to make a decision based on conditions. In this, a piece of code is executed or not executed based on a specified condition. This controls the flow of execution of the program.

The following are flow control statements in Scala :

- if
- if-else
- Nested if-else
- if-else if ladder

if statement – This is the simplest decision-making statement. It only contains “if block” which is executed If the specified condition is true and If the specified condition is false then “If block” will not be executed.

```
object Main {
  def main(args: Array[String]): Unit = {
    val age:Int=11
    if(age>10)
      {
        print("true")
      }
  }
}
```

Output:

if else statement – It contains two different blocks “if block” and “else block”. If the specified condition is true then “if block” is executed and if condition is not true then “else block” is executed.

Example:

```
object Main {
  def main(args: Array[String]): Unit = {
    val age:Int=10
    if(age>10)
      {
        print("true")
      }else
      print("false")
  }
}
```

Output:

Nested if-else **–** Scala allows us to define **if-else** statement inside an if statement or in a else statement.

Example:

```
object Main {
  def main(args: Array[String]): Unit = {
    val age:Int=10;
    if(age>10)
      {
        if(age==10)
          {
            print("Age is 10")
          }
          else {
            print("Not equal to 10");
       }
        }else {
       print("Age is not grater than 10")

    }

  }
}
```

Output:

if-else if ladder **–** This is useful when you have multiple conditions. And at a time, only one condition could be true. In this, the *if* statements are executed from the top down. As soon as one of the conditions is true, the statement associated with that *if* is

executed, and the rest of the ladder is skipped. In case if none of them is true, then “else block” is executed.

Example:

```
object Main {
  def main(args: Array[String]): Unit = {
    val age:Int=10;

     if(age==10)
      {
       print("age is 10")
      }else if(age==18) {
      print("age is 19")
      }
     else
      print("not matched ")
  }
}
```

Output:

Loops

Loops are a very important part of any programming language. Which allows us to execute a block of code several number of times.

The following are loop control statements in Scala:

1. While
2. Do while
3. For

While – It executes a statement or group of statements while a given condition is true. The condition is tested before executing any statement of the loop body.

Example:

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

    var index: Int = 0
      while(index<10)
      {
        print(index);
        index=index+1
      }
```

```
8
9       }
10    }
11
12
```

Output:

Do While – It is similar to while loop the only difference is it executes the body once and then it tests the condition.

```
1
2     object Main {
3       def main(args: Array[String]): Unit = {
4
5           var index: Int = 0
6            do
7            {
8              print(index);
9              index=index+1
10           }while(index<10)
11
12      }
13    }
```

Output:

For – It is often used with collection objects. It loops through a block of code a number of times. I will cover the collection library later in detail.

Here ← is an operator which is called a generator.

```
1
2     object Main {
3       def main(args: Array[String]): Unit = {
4
5         // for with range
6         for( a <- 1 to 20){
7           println( "Value : " + a );
8         }
9
10        // print all elements of a list
11        val list1:List[String]=List("Apple","banana")
12        for(ele <- list1){
13          println( "ele : " + ele );
14        }
15
16
17      }
18    }
```

Output:

Class & object

In this chapter, we are going to cover class & objects in Scala. If you are familiar with any object-oriented programming(OOPs) language then it would be super easy for you to understand concepts related to object & class.

Let's get started with the definition of class and object.

Class – A class is a blueprint of an object.

Object – Object is nothing but an instance of a class.

So, the first step is to define a class and then create an object of that class using a new keyword. Once you have an object, you can call any method of the class.

Let's understand this by taking an example of **Employee** class. As you can see the Employee class has ID and Name as member variables. It also has **setID()** and **setName()** as Member methods.

Here, Rahul and Chandan are objects of Employee class.

Example:

```
class Employee(idc:Int,namec:String) {
    var ID:Int=idc
    var name:String=namec

    def getName(): String =
    {
       name
    }

}

object Main {
  def main(args: Array[String]): Unit = {
    val emp1:Employee =new Employee(11,"rahul")
    val emp2:Employee =new Employee(11,"Chandan")
    print(emp1.getName())

  }
}
```

```
17
18
19
20
```
Output:

It printed “Rahul” because we only checked the name of one employee.

Here, we created an Employee class with two data members and one member method.

Employee(idc:Int, namec:String) – This works as a constructor for the class.

new Employee(11,”Rahul ”) – This statement creates an object**.**

getName() – Returns name of the employee.

Note – There is no return keyword inside **getName()** method. Actually the last expression becomes the return value for the method.

Inheritance

This is one of the important OOPS concepts. It allows us to inherit properties of another class using an extended keyword.

Super class or parent class – A class which is extended called super or parent class.

Base class or derived class – A class which extends a class is called derived or base class.

Let’s take a real life example to understand this.

```
1
2
3
4
5
6
7
8
9
10
11
```
```
class Google {

 def search(keyword: String): Unit ={
   println("Your Search keyword:"+keyword);
   println("Searching please wait....");
 }

 def youtube(url: String): Unit ={
   println("Your Video URL:"+url);
   println("Playing....");
 }
```

```
}

class User(serviceT:String,inputT:String) extends Google{

 var service:String = serviceT
 var input:String = inputT

 def action(): Unit =
 {
     if(service.equals("search")){
       search(input);
     }else{
       youtube(input);
     }

 }

}

object Main {
 def main(args: Array[String]): Unit = {
   val user1: User = new User("search","what is Scala?");
   user1.action()
   val user2: User = new User("youtube","https://www.youtube.com/watch?v=i9o70PMqMGY");
   user2.action()

 }
}
```

Output:

In the above example, we have considered two classes Google class and User class. User is a derived class. This class will inherit all non-private members of the Google class, such as **search** and **youtube** methods. So when we call the “action” method of User class, then it uses Google class methods to perform the action. This is possible because of the inheritance feature supported by OOPs languages.

Scala supports different types of inheritance, including single, multilevel, multiple, and hybrid.

Singleton Objects

As we know, Scala is a **pure object–oriented language** because every value is an **object** in Scala. Instead of static members Scala offers **singleton objects.** A

singleton is nothing but a class that can have only one instance. In this type of class you need not to create an object to call methods declared inside a singleton object.

It is possible to create **singleton** objects by just using **object** keyword Instead of a class keyword. You have already seen the singleton **class,** which we call Scala's main method. Below is an example of **singleton class.**

Example:

```
object Main {

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

    print("singleton object")
}}
```

Output:

Access Modifiers

Access Modifiers are nothing but a set of keywords in Scala which controls the **accessibility** of classes, methods, and other members. Scala provides the following keywords.

1.Private

2.Protected

3.Public (not a keyword)

Private – When any member is declared as private, it can only be used inside defining class.

Example:

```
class Employee {
  private val name:String="Rahul kr"
  def getName(): String ={
    name
  }

}
object Main {
```

```
    def main(args: Array[String]): Unit = {
      val emp: Employee = new Employee();
      print(emp.getName())
      print(emp.name) // not accessible
    }
}
```

Output:

Protected – Protected members are only accessible in the class itself and its subclasses.

Example:

```
class Employee {
  protected val name:String="Rahul kr"
}
class Programmer extends Employee {

  def getName(): String ={
    name
  }
}
object Main {
  def main(args: Array[String]): Unit = {
    val emp: Programmer = new Programmer();
    print(emp.getName())

    val emp1: Employee = new Employee();
    print(emp1.name) // not accessable
  }
}
```

Output:

Public – Public is not a keyword. By default, all members are public and can be accessed from anywhere.

Example:

```
class Employee {
   val name:String="Rahul kr"
}
object Main {
  def main(args: Array[String]): Unit = {
    val emp1: Employee = new Employee();
```

```
6         print(emp1.name)
7       }
8     }
9
```

Output:

*Top-level protected and private members are accessible from inside the package.

Companion – It is a singleton object named same as the class.

Scala Functions

Scala is a pure Object-oriented programming language. But it also supports a functional programming approach. So, It provides built-in functions and allows us to create user-defined functions as well. Functions are first-class values in Scala. It means, it is possible to store function value, pass a function as an argument, and return a function as a value from another function.

def keyword is used to create a function in Scala.

Basic Syntax:

def funName(parameters : type) : return type = {
// statements
}

Let’s create a simple Scala function!

```
1
2     object Main {
3
4       def getAge():Int   = {
5         val age:Int=10
6         age
7       }
8
9       def main(args: Array[String]): Unit = {
10       print(getAge())
11
12      }
    }
```

Above code, created a function called **getAge()** which returns a value.

Anonymous function – Any function without a name is called an anonymous function. Below is an example of an anonymous function. It takes two integers and prints the sum.

Example:

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

  val res1=(a:Int,b:Int) => a+b
  val res2=(_:Int )+( _:Int)// same as above
  print(res1(1,2))
  print(res2(1,2))

}
```

Output: 3 3

Nested function – Scala allows us to define a function within another function.

Let's understand this by an example.

```
object Main {

  def multiply(a:Int,b:Int,c:Int): Int ={

      def multi(x:Int,y:Int): Int =
      {
        x*y
      }
      c* multi(a,b);
  }

  def main(args: Array[String]): Unit = {
    println(multiply(1,2,3))
  }
}
```

Output: 6

Here, multi is a function defined inside another function.

Currying Functions – Currying function is a technique of transforming a function that takes multiple parameters into a chain of functions, each taking a single parameter.

Example:

```
object Main {

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

    def add(a:Int)=(b:Int)=> a+b;
     // can also use below
    def add1(a:Int)(b:Int) = a+b;

    println(add(1)(2))
    println(add1(1)(2))
  }
}
```

Output:

Recursion function – Recursion is a very important concept of pure functional programming. Recursion happens when a function can call itself repeatedly.

Example:

```
object Main {
  def printFun(n:Int): Unit ={
    if(n<0)
      return ;
    println(n)
    printFun(n-1)
  }

  def main(args: Array[String]): Unit = {
    printFun(3);
  }
}
```

Output:

Functions with Named Arguments – Generally, while calling a function the order of argument is very important. But a named argument allows us to pass arguments to a function in any order.

Example:

```
object Demo {
  def printInt( a:Int, b:Int ) = {
    println("a : " + a );
    println("b : " + b );
```

```
    }
    def main(args: Array[String]) {
      printInt(b = 5, a = 7);
    }

}
```

Output:

Partially Applied Functions – This is a technique to generate a brand new function from an existing function. As we know, whenever we call any function we need to provide all parameters to the function. But Scala allows us to provide only some of them, leaving remaining parameters to be passed later.

Once you give the required initial parameters, you get back a new function whose parameter list only contains those parameters from the original function that were left blank.

Let's understand this by taking an example.

```

object Main {

   def getPrice(discount:Float,original_price:Float): Float =
   {
     original price - (original_price * discount / 100)
   }
  def main(args: Array[String]) {
     println(getPrice(20,100))
     /*
     if 10 % discount is fixed for all product
     then we can create another function with fixed discount
     */
     val getPriceWithFixedDiscount=getPrice(10,_:Float)
     println(getPriceWithFixedDiscount(100))
  }

}
```

Output:

Here, we created a function **getPrice()** which returns the price after subtracting the discount from the original price. Now, suppose the discount is fixed for all products then probably we can create another function where you need to pass only original_price.

Below statement creates **getPriceWithFixedDiscount()** function from **getPrice().**

val getPriceWithFixedDiscount=getPrice(10,_:Float);

Now, call it like this and provide only the original price.

getPriceWithFixedDiscount(100)

Call-by-name – Generally, parameters to functions are call-by-value parameters. It means the value of the parameter is calculated before it is passed to the function. But in call-by-name a code block is passed to the function and each time the function accesses the parameter, the code block is executed and the value is calculated.

```
1
2     object Main {
3
4       def byName(x: => Long)
5       {
6         println(x)
7         println(x)
8       }
9       def main(args: Array[String]) {
10        byName(System.nanoTime())
11      }
12
    }
```

Output :

29951397835608

29951400165187

Here, we created a **byName()** function and used => symbol to perform call-by-name and this function contains two print statements. Now, the question is when we called **byName** function only once. Then why it printed different values. The reason is whenever we assess the value of x it is called **System.*nanoTime*()** method which was passed while calling the function.

Function with Variable Arguments – This is useful when you don't know how many arguments you are going to pass to a function.

Example:

```
1     object Main {
2       def sum(args: Int *):Int=
3       {
```

```
4         var sum:Int=0;
5          for(value <- args) {
6            sum=sum+value
7          }
8          sum
9       }
10      def main(args: Array[String]) {
11          println(sum(1,2,3,4))
12          println(sum(4,5,6))
13      }
14
15  }
```

Output:

Just use * just after the data type to denote more than one argument.

Default Parameter Values for a Function – It allows to set default values for parameters

Example:

```
1
2   object Main {
3
4     //with default value
5     def sub(a:Int=0,b:Int=0):Int= {
6       a-b;
7     }
8
9     def main(args: Array[String]): Unit = {
10
11       print(sub(1,2))
12       print(sub(1))
13    }
14
15  }
```

Output:

In the above example, **the sub ()** function has been defined with two parameters a and b. The default value for these parameters is zero. You can also notice that there is no return keyword. Because the last expression becomes the return value for the method.

Note – If you don't use "=" your function will not return anything and will work like a subroutine.

Higher-order functions

Higher order functions are those functions that can either receive a function as an argument

or returns a function. This feature is not available in java.

Let’s see a couple of examples.

Example: Function as an argument

```
object Main {

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

  def higherOrderFunction(f:(Int,Int)=>Int):Unit = {
      println(f(11,11))
  }

  def add(a:Int,b:Int):Int = {
    a+b
  }
}
```

Output: 22

In the above code, we created a function called **higherOrderFunction** and instead of passing a value, we passed a function called add() as an argument**.** And this **higherOrderFunction** doesn’t return anything.

f:(Int,Int)=>Int – It means the function will take two integers and will return an integer. This definition should match with the function that you are going to pass as an argument to another function.

Example: Function returns a function

```
object Main {

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

    val add = higherOrderFunction()
```

```
6         add(1,4)
7       }
8
9       def higherOrderFunction()= {
10        (a:Int,b:Int) => println(a+b)
11      }
12
13    }
```

Output – 5

In the above example, we created a **higherOrderFunction** which returns a function which can add two elements.

val add = *higherOrderFunction*()- This line stores the return of higherOrderFunction into a variable called add.

add(1,4)– As the return type of higherOrderFunction function is function. So add() function is called with arguments.

(a:Int,b:Int) => println(a+b) – This is a function without a name called anonymous function. It takes two integers and prints the sum.

Closure

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

Example:

```
1
2   object Main {
3     var increment  = 1
4     val add = (i:Int) => i + increment
5     def main(args: Array[String]) {
6       println( add(2) )
7     }
8
9   }
```

Output:

Here, increment is a free variable(defined outside) **and add()** is a **closure** function. If the closure function changes increment value then the change becomes visible outside the function.

String

Sting is a very common data structure. It is nothing but a sequence of characters. In Scala, objects of String are immutable which means we can't change once created.

Let's see how to create a string and its common functions.

indexOf(int index) – Returns the index of the specified character.

charAt(int index) – Returns the character at the specified index.

Length – Returns the length of the string.

String[] split(String regex) – Splits this string around matches of the given regular expression.

string1.concat(string2)- this returns a new string that is string1 with string2 added to it at the end.

string1.equals(string2)- compares two strings, returns true if both strings are same else false

Example:

```
1
2     object Main {
3
4       def main(args: Array[String]) {
5         val name1:String="Shyam kumar"
6         val name2:String="Shyam"
7         println(name1.indexOf('S'))
8         println(name1.length)
9         val str:Array[String]=name1.split(' ')
10        str.foreach(x=>println(x))
11        println(name1.charAt(0))
12        println(name1.toLowerCase)
13        println(name2.concat(" kumar") )
14        println(name1 )
15        println(name1.equals(name2) )
16
17      }
18
    }
```

19
Output :

0

11

Shyam

kumar

S

shyam kumar

Shyam kumar

Shyam kumar

False

String Interpolation

String Interpolation is the new way to create Strings in Scala programming language. This feature supports the versions of Scala-2.10 and later.

The ‘s’ String Interpolator – The literal ‘s’ allows us to use a variable directly in a string, when you prepend ‘s’ to it.

Example:

```
object Main {

  def main(args: Array[String]) {
    val name:String="Rhaul roy"
    println(s"my name is $name")

  }

}
```

Output: my name is Rahul Roy

The f Interpolator – Prepending **f** to any string literal allows the creation of simple formatted strings, similar to printf in other languages. When using the f interpolator, all variable references should be followed by a printf-style format string, like %s for string.

Example:

```
object Main {

  def main(args: Array[String]) {
    val name:String="Rhaul roy"
    println(f"my name is $name%s")
  }

}
```

Output: my name is Rhaul roy

'raw' Interpolator – The 'raw' interpolator is similar to 's' interpolator except that it performs no escaping of literals within a string.

Example:

```
object Main {

  def main(args: Array[String]) {
    val name:String="Rhaul roy"
    println(raw"my \n name is $name")
  }

}
```

output: my \n name is Rhaul roy

Array

Scala provides an array data structure. which stores a sequential collection of elements of the same type. The size of an array is fixed.

Syntax

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

or

var temp = new Array[String](3)

Example:

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

    // print all elements of a array
    val arr:Array[String]=Array("Apple","banana")
    for(ele <- arr){
      println( "ele : " + ele );
    }

  }
}
```

Output:

Two-Dimensional Arrays – Two-Dimensional is nothing but an array of arrays. Data in this kinds of arrays are stored in tabular format.

var matrix = ofDim[Int](3,3)

```
import Array._
object Main {
  def main(args: Array[String]) {
    var matrix = ofDim[Int](4,4)

    for (i <- 0 to 3) {
      for ( j <- 0 to 3) {
        matrix(i)(j) = j;
      }
    }

    for (i <- 0 to 3) {
      for ( j <- 0 to 3) {
        print(" " + matrix(i)(j));
      }
      println();
    }
  }
}
```

Output:

0 1 2 3

0 1 2 3

0 1 2 3

0 1 2 3

Concatenate Arrays – Below example shows how to concatenate two arrays in Scala.

```
object Main {
  def main(args: Array[String]) {
    var list1 = Array("a", "b")
    var list2 = Array("c","d")
    var list = concat( list1, list2)

    for ( x <- list ) {
      print( x )
    }
  }
}
```

Output: abcd

Create Array with Range – In Scala, it is possible to use range() method to generate an array containing a sequence of increasing integers in a given range. We will cover range() in detail.

Example:

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

    var list:Array[Int] = range(1,3,1)
    for ( x <- list ) {
      println( x )
    }
  }
}
```

Output

1

2

Common Array Methods

The following are commonly used methods of Arrays:

def concat[T](xss: Array[T]*): Array[T] – Concatenates all arrays into a single array.

def empty[T]: Array[T] – Returns an array of length 0.

def ofDim[T](n1: Int): Array[T] – Creates array with given dimensions.

Collections

Scala provides a rich set of collection libraries. It contains classes as well as traits. These collections could be mutable or immutable. You can use them according to your requirement.

Scala.collection.mutable package has all the mutable collections. It means if you import this package you will be able to add, remove and update data.

Scala.collection.immutable has all the immutable collections. It does not allow you to modify data. Scala will import this package by default. If you want mutable collection, then you need import **scala.collection.mutable** package in your code.

Let's get started with the first collection object.

List – List is nothing but LinkedList in Scala. It is an interface in Java. But in Scala it is a class. Lists are immutable in Scala, which means elements of a list cannot be changed by assignment operator.

Let's see an example:

```
object Main {
  def main(args: Array[String]) {
    var list:List[String] = List("Scala","Java","Python","C++")
    for ( x <- list ) {
      println( x )
    }
  }
}
```

9
Output :

Scala

Java

Python

C++

Scala List Methods – The following are commonly used List methods:

def length: Int → Returns the length of the list.

def reverse[Y >: X]: List[X] → Reverses the list.

def sorted[Y >: X]: List[X] → Sorts the list according to an Ordering.

def toString(): String → Converts the list to a string.

Example:

```
1
2     object Main {
3       def main(args: Array[String]) {
4         var list:List[String] = List("Scala","Java","Python","C++")
5
6         println(list.length)
7         println(list.reverse)
8         println(list.sorted)
9         println(list.indexOf("Java"))
10        println(list.toString())
11
11      }
12    }
13
```

Output:

4

List(C++, Python, Java, Scala)

List(C++, Java, Python, Scala)

1

List(Scala, Java, Python, C++)

Basic Operations on Lists – There are basic three different basic operations that can be performed on a list.

Head – It returns the head node.

tail – It returns all elements except first

isEmpty – It returns true if list is empty

```
object Main{
  def main(args: Array[String]) {
    var list:List[String] = List("Scala","Java","Python","C++")

    println(list.head)
    println(list.tail)
    println(list.isEmpty)

  }
}
```

Output:

Scala

List(Java, Python, C++)

false

Concatenating Lists – **:::** operator or **List.:::()** method or **List.concat()** methods can be used to concatenate more than one Scala lists.

Example:

```
object Main {
  def main(args: Array[String]) {
    var list1:List[String] = List("Scala","Java")
```

```
3         var list2:List[String] = List("Python","C++")
4
5         println(list1:::list2)
6         println(List.concat(list1,list2))
7         println(list1.:::(list2))
8
9       }
10    }
11
```

Outpt:

List(Scala, Java, Python, C++)

List(Scala, Java, Python, C++)

List(Python, C++, Scala, Java)

Set

In a list duplicates elements are allowed. But Set is a collection that contains no duplicate elements. There are two different type of set **immutable** and **mutable.**

By default, Scala uses the immutable Set. In case, if you need a mutable Set. You can import **scala.collection.mutable.Set** class explicitly.

Example:

```
1
2   object Main {
3     def main(args: Array[String]) {
4       var set1:Set[String] = Set("Scala","Java","C++","C++");
5       for(ele<-set1)
6         println(ele)
7
8     }
9   }
```

Output:

Scala

Java

C++

Scala set methods – The following are commonly used set methods:

def contains(element: A): Boolean – Returns true if element is present in this set, false otherwise.

def min: A – Returns the smallest element.

def max: A – Returns the largest element.

def +(element: A): Set[A] – Creates a new set with an additional element, unless the element is already present.

def size: Int → Returns the size of the set.

Example:

```
object Mains {
  def main(args: Array[String]) {
    var set1: Set[Int] = Set(1, 2, 3 ,3,4);
    println(set1.size)
    println(set1.contains(3));
    println(set1.min);
    println(set1.max);
    //creates new set with new element
    println(set1.+(20))

  }
}
```

Output:

4

true

1

4

Set(20, 1, 2, 3, 4)

Basic Operations on set – There are basic three different basic operations that can be performed on a list.

Head – It returns the first element.

tail – It returns all elements except first.

isEmpty –It returns true if the set is empty.

```
object Demo5 {
  def main(args: Array[String]) {
    var set1: Set[Int] = Set(1, 2, 3 ,3,4);
    println(set1.head)
    println(set1.tail);
    println(set1.isEmpty);


  }
}
```

Output:

1

Set(2, 3, 4)

False

Concatenating sets – **++** operator or **List.++()** method can be used to concatenate more than one Scala sets.

Example:

```
object Main {
  def main(args: Array[String]) {
    var set1: Set[Int] = Set(1, 2, 3 ,3,4);
    var set2: Set[Int] = Set(10, 20, 3 );
    var set3=set1++set2
    var set4=set1.++(set2)
    println(set3)
    println(set4)
  }
}
```

```
9
10
```
Output:

Set(10, 20, 1, 2, 3, 4)

Set(10, 20, 1, 2, 3, 4)

Find common values in sets

Example:

```
1
2
3
4
5
6
7
8
```
```
object Demo5 {
  def main(args: Array[String]) {
    var set1: Set[Int] = Set(1, 2, 3 ,3,4);
    var set2: Set[Int] = Set(10, 20, 3 ,4);
    println(set1.intersect(set2));

  }
}
```
Output:

Set(3, 4)

As you can see both contain 3 and 4.

Map

Map is data structure which contains collection of key/value pairs. It is possible to fetch the value if we know the key. Keys are unique in map but value may not be unique.

.There are two kinds of Maps **immutable** and **mutable**.

By default, Scala uses the immutable Map. For mutable map please import **scala.collection.mutable.Map**.

Example:

```
1
2
3
```
```
object Main {
  def main(args: Array[String]) {

    var map1:Map[Int,String]= Map(1->"one",2->"two",3->"three")
```

```
println(map1);

  }
}
```

Output:

Map(1 -> one, 2 -> two, 3 -> three)

In the above example, 1-> “one” is one element where 1 is a key and “one” its value.

Scala map methods – The following are commonly used map methods:

def get(key: A): Option[B] – Optionally returns the value associated with a key.

def contains(key: A): Boolean – Returns true if there is a binding for key in this map, false otherwise

def **– (elem1: A, elem2: A, elems: A*): Map[A, B]** – Returns a new map containing all the mappings of this map except mappings with a key equal to elem1, elem2 or any of elems.

def empty: Map[A, B] – Returns the empty map of the same type.

Example:

```

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

    var map1:Map[Int,String]= Map(1->"one",2->"two",3->"three")
    println(map1.get(1).getOrElse("Not found"));
    println(map1.get(8).getOrElse("Not found"));
    println(map1.contains(1));;
   // removing one element
    println(map1.-(1))

    println(map1.isEmpty)

  }
}

```

Output:

one

Not found

true

Map(2 -> two, 3 -> three)

false

Basic Operations on Map – There are basic three different basic operations that can be performed on a list.

key – This method returns an iterable containing each key in the map.

values –It returns an iterable which contains all values of the map.

isEmpty –It returns true if the set is empty.

```
object Main {
  def main(args: Array[String]) {
    var map1:Map[Int,String]= Map(1->"one",2->"two",3->"three")
    var keys=map1.keys
    for(key<-keys)
      println(key)

    var values=map1.values
    for(value<-values)
      println(value)

  }
}
```

Output:

1

2

3

one

two

three

Concatenating maps – **++** operator or **Map.++()** method can be used to concatenate more than one Scala Maps.

Example:

```
object Main {
  def main(args: Array[String]) {
    var map1:Map[Int,String]= Map(1->"one",2->"two",3->"three")
    var map2:Map[Int,String]= Map(4->"four",5->"five")
    println(map1++map2)
    println(map1.++(map2))

  }
}
```

Output:

Map(5 -> five, 1 -> one, 2 -> two, 3 -> three, 4 -> four)

Map(5 -> five, 1 -> one, 2 -> two, 3 -> three, 4 -> four)

Scala tuple

Tuple is a unique data structure where a fixed number of items can be stored together and can be passed around as a whole. It allows us to store objects with different types.

Example:

```
object Main {
  def main(args: Array[String]) {
    val tuple1=(1,2);
    val tuple2=Tuple3(1,2,3);
    println(tuple1._1+tuple1._2)
    println(tuple2._1+tuple2._2)

  }
}
```

```
9
10
```
Output:

```
3


3
```

In the above example, we created a tuple with three values and stored it into tuple2. It is very easy to pass this to a function as a parameter and values can be accessed using **._1** or **._2** so on.

Iterate over the Tuple

There is direct method to iterate over all elements like List. You need to use **Tuple.productIterator()** method to iterate over all the elements of a Tuple.

Example:

```
1
2
3
4
5
6
```
```
object Main {
  def main(args: Array[String]) {
    val tuple2=Tuple3(1,2,3);
    tuple2.productIterator.foreach{ i =>println( i )}
  }
}
```
Output:

```
1


2


3
```

Converting to String

It is possible to convert a Tuple into a string.

```
1
2
3
4
```
```
object Main {
  def main(args: Array[String]) {
    val tuple2=Tuple3(1,2,3);
    println(tuple2.toString())
  }
```

```
5    }
6
```
Output:

(1,2,3)

Scala Option

Scala Option[T] is a container for zero or one element of a given type. An Option[T] can be either **Some[T]** or **None** object, which represents a missing value.

Example:

```
1    object Main {
2      def main(args: Array[String]) {
3
4        var map1:Map[Int,String]= Map(1->"one",2->"two",3->"three")
5        println(map1.get(1));
6
7      }
8    }
```
Output:

Some(one)

Map1.get(1) returns Some[T] . If you want to get the value you can use the get**()** method or **getOrElse().**

Example:

```
1    object Main {
2      def main(args: Array[String]) {
3
4        var map1:Map[Int,String]= Map(1->"one",2->"two",3->"three")
5        println(map1.get(1).get);
6
7      }
8    }
```
Output: one

Here, we used the get method to fetch value from Some[T].

Suppose if the key doesn't exist then it returns **get** returns an exception**.** To avoid this you can use **getOrelse().** See below example.

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

    var map1:Map[Int,String]= Map(1->"one",2->"two",3->"three")
    println(map1.get(5).getOrElse("Not found"));

  }
}
```

Output : Not found

It printed "Not found" because key 5 doesn't exist in the map1.

isEmpty() – you can use this method to check whether an Option[T] is None or not.

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

    val t1:Option[Int] = Some(5)
    val t2:Option[Int] = None

    println(t1.isEmpty)
    println(t2.isEmpty)

  }
}
```

Output:

false

true

Iterators

An iterator allows us to access the elements of a collection one by one. But iterator itself is not a collection.

Let's see one simple example:

```
object Main {
  def main(args: Array[String]) {
    val it = Iterator(1, 2, 3, 5)
    while (it.hasNext){
      println(it.next())

    }
  }
}
```

Here, hasNext is a method that returns true if the next element is present or false. Whereas next() method moves the pointer to the next element.

Few commonly used methods –

def size: Int – Returns the number of elements in this traversable or iterator.

def min: A – Finds the minimum element. The iterator is at its end after this method returns.

def contains(elem: Any): Boolean – Tests whether this iterator contains a given value as an element.

Example:

```
object Demo6 {
  def main(args: Array[String]) {
    val it1 = Iterator(1, 2, 3, 5)
    println(it1.size)
    val it2 = Iterator(1, 2, 3, 5)
    println(it2.min)
    val it3 = Iterator(1, 2, 3, 5)
    println(it3.contains(1))

  }
}
```

Output:

4

1

true

In the above example, we created three iterator because it can be traversed only once.

Trait

Unlike a class, Scala traits cannot be instantiated. It cannot have arguments or parameters. However, you can inherit it using classes and objects

A Trait can have both **abstract** and **non-abstract** methods, fields as its members. If you initialize any method then that method becomes not-abstract and those methods that are not initialized are called abstract.

But the important thing is, In case of abstract methods, the class that implements the trait takes care of the initialization part. If not then the compiler will throw an error.

Now, let's quickly see how it works!

Scala Trait Syntax

trait Trait_Name{

// Variables

// Methods

}

Example:

```
1   trait Course{
2
3     def Java()
4     def Scala()
5
6   }
7
8   class GreatLearning extends{
9
10    def Java(): Unit ={
11      println("Welcome to Java Course")
12
13    }
14
    def Scala(): Unit ={
```

```
      println("Welcome to Scala Course")

    }

}

object Demo6 {
  def main(args: Array[String]) {
   val gl:GreatLearning =new GreatLearning();
    gl.Java()

  }
}
```

Output:

In the above example, the method Java() and Scala() were an abstract method. Hence, the declaration was made in the class that inherited that trait. But in case if you have a non-abstract method then the class which extends need not to implement the method.

Pattern Matching

It is similar to java switch. But Pattern matching of Scala is more powerful. It allows us to match a value against a pattern. Pattern matching is the most commonly used feature of Scala. You can use this to identify object's type.

Let's take a simple example:

```

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

    val x: Int=4
    var res= x match {
      case 1 => "one"
      case 2 => "two"
      case 3 => "three"
      case _ => "other"
    }
    println(res)

  }
}

```

15
Output: other

Here, X is 4 and it will be matched against all cases and the result will be stored in res variable .

Another example:

```
1
2     object Demo6 {
3       def main(args: Array[String]) {
4
5         def matchNow(x: Int): String = x match {
6           case 1 => "one"
7           case 2 => "two"
8           case _ => "other"
9         }
10        println( matchNow(3))
11        println( matchNow(1) )
12
13
14      }
15    }
16
```

Output:

Here, **matchNow()** is a function which will return matched value.

Matching on case classes

Let’s understand this by taking an example of Employee class.

```
1     abstract class Employee
2
3     case class SalariedEmployee(name:String,annualSalary:Int) extends  Employee
4
5
6     case class HourlyEmployee(name:String,perHour:Int) extends Employee
7
8     object Main {
9       def processPayment(employee: Employee): String = {
10        employee match {
11          case SalariedEmployee(name, annualSalary) =>
12            s"Name:$name \nannual salary : $annualSalary  "
13          case HourlyEmployee(name, perHour) =>
14            s"Name:$name \nPer hour :$perHour  "
15        }
16    }
```

```
15      def main(args: Array[String]) {
16
17       var emp1=SalariedEmployee("Rahul",800000)
18        var emp2=HourlyEmployee("Shayam",89)
19
20        println(processPayment(emp1))
21        println(processPayment(emp2))
22
23      }
24    }
25
26
```

Output :

Name:Rahul

annual salary : 800000

Name:Shayam

Per hour :89

The function **processPayment()** takes an abstract class Employee as a parameter. And this is matched against the type
of Employee (i.e *HourlyEmployee*, *SalariedEmployee*).

Exception Handling

An exception is an event that disturbs the normal flow of a program. Exception handling is a mechanism which allows us to handle abnormal conditions.

Scala makes “checked vs unchecked” very simple. It doesn’t have checked exceptions.

Scala Try Catch

Scala has a try and catch block to handle exceptions. If you think that some part of your code might throw an exception then you keep that part in the try block. The catch block exception occurred in the try block. It is possible to add any number of try catch blocks in your program according to need.

Example:

```
object Main{

  def divide(first:Int, second:Int) = {
    try{
      first/second
    }catch{
      case e: ArithmeticException => println(e)
    }
    println("executing code after exception")
  }
  def main(args:Array[String]){

    divide(100,0)

  }
}
```

Output :

java.lang.ArithmeticException: / by zero

executing code after exception

Finally Block

If your program acquires some resources then you can use finally block to release resources during an exception. Here, resources means a file, database connection etc. The finally block is always executed.

Example:

```
object Main11{

  def divide(first:Int, second:Int) = {
    try{
      first/second
    }catch{
      case e: ArithmeticException => println(e)
    }
    finally {
      println("Finally block")
    }
    println("executing code after exception")
  }
  def main(args:Array[String]){

    divide(100,0)
```

```
15
16      }
17    }
18
19
```

Output:

java.lang.ArithmeticException: / by zero

Finally block

executing code after exception

Throwing Exceptions

You can throw exception explicitly in your code. Scala provides a **throw** keyword to throw an exception. This throw keyword is mainly used to throw custom exception.

Example:

```
1
2     object Main{
3
4       def subscribe(coins:Int)={
5         if(coins<200)
6           throw new ArithmeticException("you have less coins")
7         else println("Thanks for subscribing!")
8       }
9       def main(args:Array[String]){
10
11        subscribe(100)
12
13      }
14    }
```

Output:

Exception in thread “main” java.lang.ArithmeticException: you have less coin

at com.learnscala.hr.Main11$.subscribe(Main.scala:184)

at com.learnscala.hr.Main11$.main(Main.scala:189)

at com.learnscala.hr.Main11.main(Main.scala)

File handling

Scala provides predefined methods to deal with file. You can create, open, write and read file. Scala offers a package called scala.io which contains all functions to deal with all IO operations.

Example:

```
object Main{


  def main(args:Array[String]){

    var fileSource:BufferedSource=null
    try {
       fileSource = Source.fromFile("/user/loc/temp.txt")
    }catch {
      case e:FileNotFoundException=> println(s" Please check file path \n $e")
    }

    for(line<-fileSource.getLines){
      println(line)
    }
    fileSource.close()
  }


}
```

output:

Java

scala

c++

python

In the above example, we are reading a file called temp.txt if the file does not exist in the specified location then it throws **FileNotFoundException** exception.

Extractor

An extractor object is nothing but an object with an **unapply** method. Whereas the **apply** method is like a constructor which takes arguments and creates an object. But the **unapply** takes an object and tries to give back the arguments.

Example:

```
object Name {

  def apply(firstName: String, lastName: String): String = {
    firstName +" "+lastName
  }

  def unapply(str: String): Option[(String, String)] = {
    var temp = str.split(" ");
    if (temp.length == 2)
      Some(temp(0), temp(1)) else None
  }


  def main(args: Array[String]) {
    var name= Name("Ram","kumar");
    println( Name.unapply(name))
    println( Name.unapply(name).get)
    println( Name.unapply(name).get._1)


  }
}

```

Output:

Here, The **apply** method createsa Name string from a first name & Last name.
The **unapply** does the inverse.

Regular expression

Regular expressions are strings which can be used to find patterns in data. In scala, any string can be converted to a regular expression using the .r method.

Example:

```
object Main {
```

```
2
3       def main(args: Array[String]) {
4         val numberPattern: Regex = "[0-9]".r
5
6         numberPattern.findFirstMatchIn("mypassword") match {
7           case Some(_) => println("Password set")
8           case None => println("Password must contain a number")
9         }
10
11      }
12    }
13
```