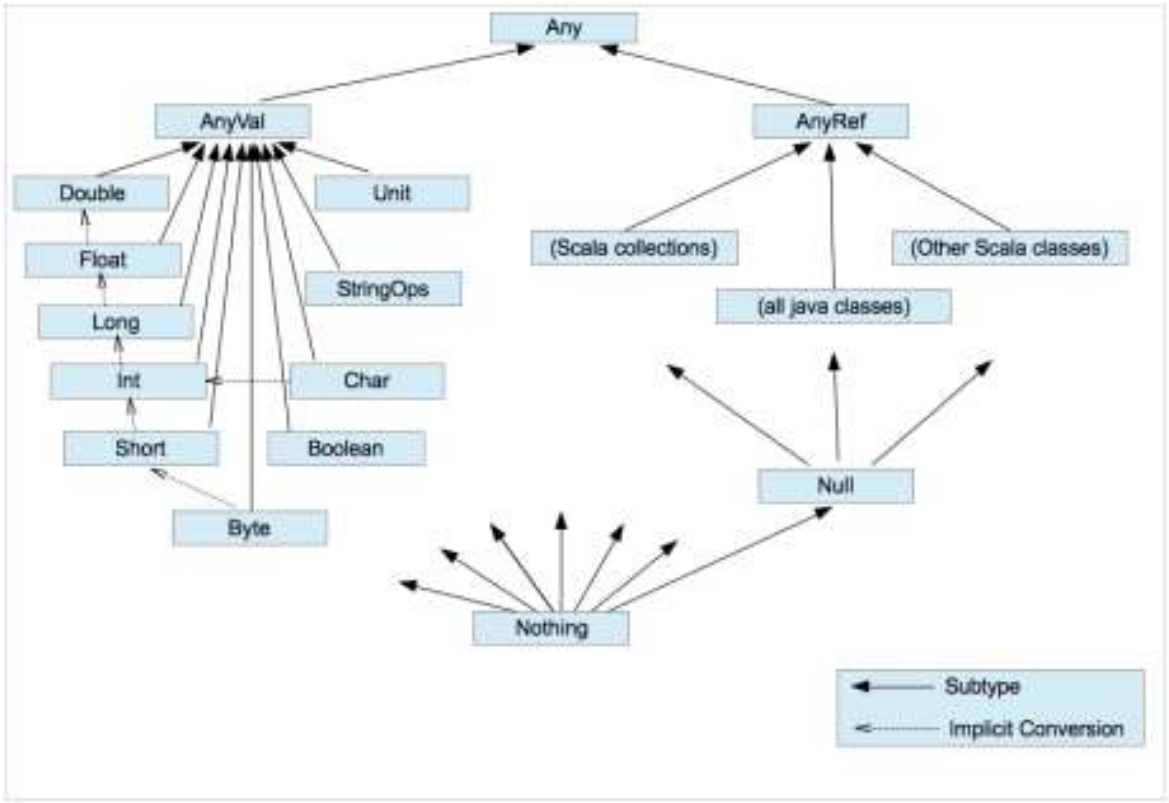
Scala Coding - II

Principles of Programming Languages

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.

Basic Types


Any
AnyVal
AnyRef
Double
Unit
Float
StringOps
Long
Int
Char
Short
Boolean
Byte
(Scala collections)
(Other Scala classes)
(all java classes)
Null
Nothing
Subtype
Implicit Conversion

Singleton Objects

- 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.

```
object Main {

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

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

Discussion on - Hello World Program

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

Save this file as HelloWorld.scala

To compile and run, use the below commands:

\> scalac HelloWorld.scala

\> scala HelloWorld

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.

Discussion on Simple if-else

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

```
true
Process finished with exit code 0
```

```
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")

    }

  }
}
```

```
Age is not grater than 10
Process finished with exit code 0
```

Discussion on Simple Loops

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

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

  }
}
```

```
0123456789
Process finished with exit code 0
```

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

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

  }
}
```

```
0123456789
Process finished with exit code 0
```

Discussion on Simple Loops

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

    // for with range
    for( a <- 1 to 20){
      println( "Value : " + a );
    }

    // print all elements of a list
    val list1:List[String]=List("Apple","banana")
    for(ele <- list1){
      println( "ele : " + ele );
    }


  }
}
```

```
Value : 1
Value : 2
Value : 3
Value : 4
Value : 5
Value : 6
Value : 7
Value : 8
Value : 9
Value : 10
Value : 11
Value : 12
Value : 13
Value : 14
Value : 15
Value : 16
Value : 17
Value : 18
Value : 19
Value : 20
ele : Apple
ele : banana
```

Discussion on Pattern matching

- It is a generalization of C or Java's switch statement. This matching method is used instead of a switch statement.
- It is defined in Scala's root class Any and therefore is available for all objects.
- The match method takes several cases as an argument.
- Each alternative takes a pattern and one or more expressions that will be performed if the pattern matches.
- A symbol => is used to separate the pattern from the expressions.

Discussion on Pattern matching

```
object Intellipaat {
def main(args: Array[String]) {
println(matchValue(2))
}
def matchValue(i: Int): String = i match {
case 1 => "one"
case 2 => "two"
case 3=> "three"
case _=> "unknown"
}
}
```

Case Classes

- These are the special types of classes that are used for pattern matching with case expressions. By adding a case keyword there is the number of advantages which are:
  - The compiler automatically changes the constructor arguments into immutable fields.
  - The compiler automatically includes equals, hashCode and toString methods to the class

Case Classes

```
object Intellipaat {
def main(args: Array[String]) {
val a = new employee(1, “abc”)
val b  = new employee(2, “xyz”)
val c  = new employee(3, “pgr”)
for (employee <- List(a, b, c)) {
employee match {
case employee(1, “abc”) => println("Hello abc")
case employee(2, “xyz”) => println(“Hello xyz”)
case employee(id, employee_name) => println("ID: " +id + ", Employee:" + employee_name
}
}
}
case class employee(id: Int, employee_name: String) // case class
}
```

Parameterize arrays with types

- Using **new** to instantiate objects, or class instances
- Parameterization means “configuring” an instance when you create it.

```
val big = new java.math.BigInteger("12345")
```

- Parameterizing an array with a type

```
val greetStrings = new Array[String](3)
greetStrings(0) = "Hello"
greetStrings(1) = ", "
greetStrings(2) = "world!\n"
for (i <- 0 to 2)
  print(greetStrings(i))
```

You can’t change the length of an array after it is instantiated, you can change its element values. Thus, arrays are mutable objects

- Specified the type of *greetStrings* explicitly like this:

```
val greetStrings: Array[String] = new Array[String](3)
```

- Creating and initializing an array

```
val numNames = Array("zero", "one", "two")
```

Discussion on Lists

- In Scala, we can create a list in two ways

```
valvariable_name: List[data_type] = List(element1, element2 element3, element4)
```

```
valvariable_name = List(element1, element2 element3, element4)
```

```
val list1: List[Int] = List(100, 200, 300, 400, 500)
```

```
val list2 = List("hello", "hello 2", "hello3 ", "so on..")
```

Discussion on Lists

- Scala's List class: immutable sequence of objects that share the same type
- Scala's List is designed to enable a functional style of programming.
- Creating and initializing a list. `val oneTwoThree = List(1, 2, 3)`
- Method ':::' - for list concatenation.

```
val oneTwo = List(1, 2)
val threeFour = List(3, 4)
val oneTwoThreeFour = oneTwo ::: threeFour
```

- Operator '::' Cons operator

```
val twoThree = List(2, 3)
val oneTwoThree = 1 :: twoThree
```

- Nil : Empty List

```
val oneTwoThree = 1 :: 2 :: 3 :: Nil
println(oneTwoThree)
```

Discussion on Lists

- It is immutable and it is defined under scala.collection.immutable package.
- The Scala list is based on a linked list data structure.
- It provides us various methods to deal with the list.
- Data present inside Scala list should be of the same type only.

Discussion on Lists

```
# to, until
(1 to 5).toList                  # List(1, 2, 3, 4, 5)
(1 until 5).toList               # List(1, 2, 3, 4)

(1 to 10 by 2).toList            # List(1, 3, 5, 7, 9)
(1 until 10 by 2).toList         # List(1, 3, 5, 7, 9)
(1 to 10).by(2).toList           # List(1, 3, 5, 7, 9)

('d' to 'h').toList              # List(d, e, f, g, h)
('d' until 'h').toList           # List(d, e, f, g)

('a' to 'f').by(2).toList        # List(a, c, e)
```

```
# range method
List.range(1, 3)                 # List(1, 2)
List.range(1, 6, 2)              # List(1, 3, 5)
```

```
List.fill(3)("foo")              # List(foo, foo, foo)
List.tabulate(3)(n => n * n)     # List(0, 1, 4)
List.tabulate(4)(n => n * n)     # List(0, 1, 4, 9)
```

Discussion on Lists

Method	Description	Example
:+	append 1 item	oldList :+ e
++	append N items	oldList ++ newList
+:	prepend 1 item	e +: oldList
++:	prepend N items	newList ++: oldList

```
val v1 = List(4,5,6)          # List(4, 5, 6)
val v2 = v1 :+ 7              # List(4, 5, 6, 7)
val v3 = v2 ++ List(8,9)      # List(4, 5, 6, 7, 8, 9)

val v4 = 3 +: v3              # List(3, 4, 5, 6, 7, 8, 9)
val v5 = List(1,2) ++: v4     # List(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

List methods and Usage

head This method returns the first element of a list.
tail This method returns a list consisting of all elements except the first.
isEmpty This method returns true if the list is empty otherwise false.

List methods and Usage

```
object Demo {
   def main(args: Array[String]) {
      val fruit = "apples" :: ("oranges" :: ("pears" :: Nil))
      val nums = Nil

      println( "Head of fruit : " + fruit.head )
      println( "Tail of fruit : " + fruit.tail )
      println( "Check if fruit is empty : " + fruit.isEmpty )
      println( "Check if nums is empty : " + nums.isEmpty )
   }
}
```

```
Head of fruit : apples
Tail of fruit : List(oranges, pears)
Check if fruit is empty : false
Check if nums is empty : true
```

List methods and Usage - Concatenating

```
object Demo {
   def main(args: Array[String]) {
      val fruit1 = "apples" :: ("oranges" :: ("pears" :: Nil))
      val fruit2 = "mangoes" :: ("banana" :: Nil)

      // use two or more lists with ::: operator
      var fruit = fruit1 ::: fruit2
      println( "fruit1 ::: fruit2 : " + fruit )

      // use two lists with Set.:::() method
      fruit = fruit1.:::(fruit2)
      println( "fruit1.:::(fruit2) : " + fruit )

      // pass two or more lists as arguments
      fruit = List.concat(fruit1, fruit2)
      println( "List.concat(fruit1, fruit2) : " + fruit  )
   }
}
```

```
fruit1 ::: fruit2 : List(apples, oranges, pears, mangoes, banana)
fruit1.:::(fruit2) : List(mangoes, banana, apples, oranges, pears)
List.concat(fruit1, fruit2) : List(apples, oranges, pears, mangoes, banana)
```

List methods and Usage

```
object Demo {
   def main(args: Array[String]) {
      val fruit = List.fill(3)("apples") // Repeats apples three times.
      println( "fruit : " + fruit  )

      val num = List.fill(10)(2)         // Repeats 2, 10 times.
      println( "num : " + num  )
   }
}
```

```
fruit : List(apples, apples, apples)
num : List(2, 2, 2, 2, 2, 2, 2, 2, 2, 2)
```

Discussion on Lists

Method	Description
`distinct`	Return a new sequence with no duplicate elements
`drop(n)`	Return all elements after the first `n` elements
`dropRight(n)`	Return all elements except the last `n` elements
`dropWhile(p)`	Drop the first sequence of elements that matches the predicate `p`
`filter(p)`	Return all elements that match the predicate `p`
`filterNot(p)`	Return all elements that do not match the predicate `p`
`find(p)`	Return the first element that matches the predicate `p`
`head`	Return the first element; can throw an exception if the `List` is empty
`headOption`	Returns the first element as an `Option`
`init`	All elements except the last one
`intersect(s)`	Return the intersection of the list and another sequence `s`
`last`	The last element; can throw an exception if the `List` is empty
`lastOption`	The last element as an `Option`
`slice(f,u)`	A sequence of elements from index `f` (from) to index `u` (until)
`tail`	All elements after the first element
`take(n)`	The first `n` elements
`takeRight(n)`	The last `n` elements
`takeWhile(p)`	The first subset of elements that matches the predicate `p`

```
val a = List(10, 20, 30, 40, 10)        # List(10, 20, 30, 40, 10)
a.distinct                              # List(10, 20, 30, 40)
a.drop(2)                               # List(30, 40, 10)
a.dropRight(2)                          # List(10, 20, 30)
a.dropWhile(_ < 25)                     # List(30, 40, 10)
a.filter(_ < 25)                        # List(10, 20, 10)
a.filter(_ > 100)                       # List()
a.filterNot(_ < 25)                     # List(30, 40)
a.find(_ > 20)                          # Some(30)
a.head                                  # 10
a.headOption                            # Some(10)
a.init                                  # List(10, 20, 30, 40)
a.intersect(List(19,20,21))             # List(20)
a.last                                  # 10
a.lastOption                            # Some(10)
a.slice(2,4)                            # List(30, 40)
a.tail                                  # List(20, 30, 40, 10)
a.take(3)                               # List(10, 20, 30)
a.takeRight(2)                          # List(40, 10)
a.takeWhile(_ < 30)                     # List(10, 20)
```

Discussion on Lists

Method	Returns
`collect(pf)`	A new collection by applying the partial function `pf` to all elements of the list, returning elements for which the function is defined
`distinct`	A new sequence with no duplicate elements
`flatten`	Transforms a list of lists into a single list
`flatMap(f)`	When working with sequences, it works like `map` followed by `flatten`
`map(f)`	Return a new sequence by applying the function `f` to each element in the `List`
`updated(i,v)`	A new list with the element at index `i` replaced with the new value `v`
`union(s)`	A new list that contains elements from the current list and the sequence `s`

```
val x = List(Some(1), None, Some(3), None)

x.collect{case Some(i) => i}          # List(1, 3)

val x = List(1,2,1,2)
x.distinct                            # List(1, 2)
x.map(_ * 2)                          # List(2, 4, 2, 4)
x.updated(0,100)                      # List(100, 2, 1, 2)

val a = List(List(1,2), List(3,4))
a.flatten                             # List(1, 2, 3, 4)

val fruits = List("apple", "pear")
fruits.map(_.toUpperCase)             # List(APPLE, PEAR)
fruits.flatMap(_.toUpperCase)         # List(A, P, P, L, E, P, E, A, R)

List(2,4).union(List(1,3))            # List(2, 4, 1, 3)
```

Discussion on Lists

```
# diff
val oneToFive = (1 to 5).toList       # List(1, 2, 3, 4, 5)
val threeToSeven = (3 to 7).toList    # List(3, 4, 5, 6, 7)
oneToFive.diff(threeToSeven)          # List(1, 2)
threeToSeven.diff(oneToFive)          # List(6, 7)
```

```
# map, flatMap
val fruits = List("apple", "pear")
fruits.map(_.toUpperCase)             # List(APPLE, PEAR)
fruits.flatMap(_.toUpperCase)         # List(A, P, P, L, E, P, E, A, R)
```

```
List(1,2,1,2).distinct                # List(1, 2)

val a = List(List(1,2), List(3,4))
a.flatten                             # List(1, 2, 3, 4)
```

```
List(1,2,3).reverse                   # List(3, 2, 1)

val nums = List(10, 5, 8, 1, 7)
nums.sorted                           # List(1, 5, 7, 8, 10)
nums.sortWith(_ < _)                  # List(1, 5, 7, 8, 10)
nums.sortWith(_ > _)                  # List(10, 8, 7, 5, 1)
```

```
List(1,2,3).updated(0,10)             # List(10, 2, 3)
List(2,4).union(List(1,3))            # List(2, 4, 1, 3)
```

```
# zip
val women = List("Wilma", "Betty")    # List(Wilma, Betty)
val men = List("Fred", "Barney")      # List(Fred, Barney)
val couples = women.zip(men)          # List((Wilma,Fred), (Betty,Barney))

val a = List.range('a', 'e')          # List(a, b, c, d)
a.zipWithIndex                        # List((a,0), (b,1), (c,2), (d,3))
```

- Consider some sample Lists

```
val evens = List(2, 4, 6)
val odds = List(1, 3, 5)
val fbb = "foo bar baz"
val firstTen = (1 to 10).toList
val fiveToFifteen = (5 to 15).toList
val empty = List[Int]()
val letters = ('a' to 'f').toList
```

```
evens.contains(2)                          # true
firstTen.containsSlice(List(3,4,5))        # true
firstTen.count(_ % 2 == 0)                 # 5
firstTen.endsWith(List(9,10))              # true
firstTen.exists(_ > 10)                    # false
firstTen.find(_ > 2)                       # Some(3)
firstTen.forall(_ < 20)                    # true
firstTen.hasDefiniteSize                   # true
empty.hasDefiniteSize                      # true
letters.indexOf('b')                       # 1 (zero-based)
letters.indexOf('d', 2)                    # 3
letters.indexOf('d', 3)                    # 3
letters.indexOf('d', 4)                    # -1
letters.indexOfSlice(List('c','d'))        # 2
letters.indexOfSlice(List('c','d'),2)      # 2
letters.indexOfSlice(List('c','d'),3)      # -1
firstTen.indexWhere(_ == 3)                # 2
firstTen.indexWhere(_ == 3, 2)             # 2
firstTen.indexWhere(_ == 3, 5)             # -1
letters.isDefinedAt(1)                     # true
letters.isDefinedAt(20)                    # false
letters.isEmpty                            # false
empty.isEmpty                              # true
```

```
# lastIndex...
val fbb = "foo bar baz"
fbb.indexOf('a')                    # 5
fbb.lastIndexOf('a')                # 9
fbb.lastIndexOf('a', 10)            # 9
fbb.lastIndexOf('a', 9)             # 9
fbb.lastIndexOf('a', 6)             # 5
fbb.lastIndexOf('a', 5)             # 5
fbb.lastIndexOf('a', 4)             # -1
```

```
fbb.lastIndexOfSlice("ar")          # 5
fbb.lastIndexOfSlice(List('a','r'))     # 5
fbb.lastIndexOfSlice(List('a','r'), 4)  # -1
fbb.lastIndexOfSlice(List('a','r'), 5)  # 5
fbb.lastIndexOfSlice(List('a','r'), 6)  # 5
```

```
fbb.lastIndexWhere(_ == 'a')        # 9
fbb.lastIndexWhere(_ == 'a', 4)     # -1
fbb.lastIndexWhere(_ == 'a', 5)     # 5
fbb.lastIndexWhere(_ == 'a', 6)     # 5
fbb.lastIndexWhere(_ == 'a', 8)     # 5
fbb.lastIndexWhere(_ == 'a', 9)     # 9
```

```
val x = List(1,2,9,1,1,1,1,4)
x.segmentLength(_ < 4, 0)           # 2
x.segmentLength(_ < 4, 2)           # 0
x.segmentLength(_ < 4, 3)           # 4
x.segmentLength(_ < 4, 4)           # 3
```

```
firstTen.startsWith(List(1,2))      # true
firstTen.startsWith(List(1,2), 0)   # true
firstTen.startsWith(List(1,2), 1)   # false
firstTen.sum                        # 55
```

```
firstTen.fold(100)(_ + _)           # 155
firstTen.foldLeft(100)(_ + _)       # 155
firstTen.foldRight(100)(_ + _)      # 155
firstTen.reduce(_ + _)              # 55
firstTen.reduceLeft(_ + _)          # 55
firstTen.reduceRight(_ + _)         # 55
```

```
firstTen.fold(100)(_ - _)           # 45
firstTen.foldLeft(100)(_ - _)       # 45
firstTen.foldRight(100)(_ - _)      # 95
firstTen.reduce(_ - _)              # -53
firstTen.reduceLeft(_ - _)          # -53
firstTen.reduceRight(_ - _)         # -5
```

```
val oneToFive = List(1, 2, 3, 4, 5)   # List[Int] = List(1, 2, 3, 4, 5)
```

```
for (i <- oneToFive) yield i          # List[Int] = List(1, 2, 3, 4, 5)
for (i <- oneToFive) yield i * 2      # List[Int] = List(2, 4, 6, 8, 10)
for (i <- oneToFive) yield i % 2      # List[Int] = List(1, 0, 1, 0, 1)
```

```
for {                                 # List[Int] = List(3, 4, 5)
    i <- oneToFive
    if i > 2
} yield i
```

```
val oneToThree = List(1, 2, 3)
oneToThree.foreach(print)             # 123
for (i <- oneToThree) print(i)        # 123
```

```
for {                                 # List[Int] = List(6, 8, 10)
    i <- oneToFive
    if i > 2
} yield {
    # could be multiple lines here
    i * 2
}
```

Tuple

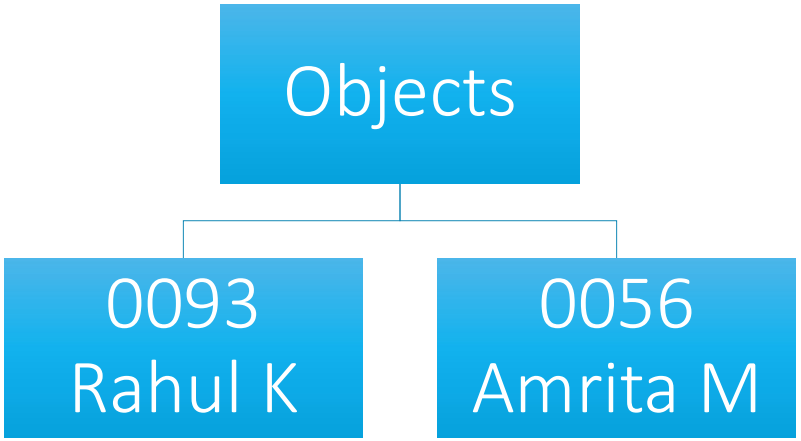
- Tuple is a collection of elements.
- Tuples are heterogeneous data structures, i.e., is they can store elements of different data types.
- A tuple is immutable, unlike an array in scala which is mutable.

```
// this is tuple of type Tuple3[ Int, String, Boolean ]
val name = (15, "Chandan", true)
```

```
val tuple3: (String, Int, Boolean) = ("Joe", 34, true)
val tuple4: (String, Int, Boolean, Char) = ("Joe", 34, true, 'A')
```

- Tuple elements can be accessed using an underscore syntax, method tup._i

Class and Object



Class and Object

```
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())

  }
}
```

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

- 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.

Example of Inheritance

```
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
   user2.action()

 }
}
```