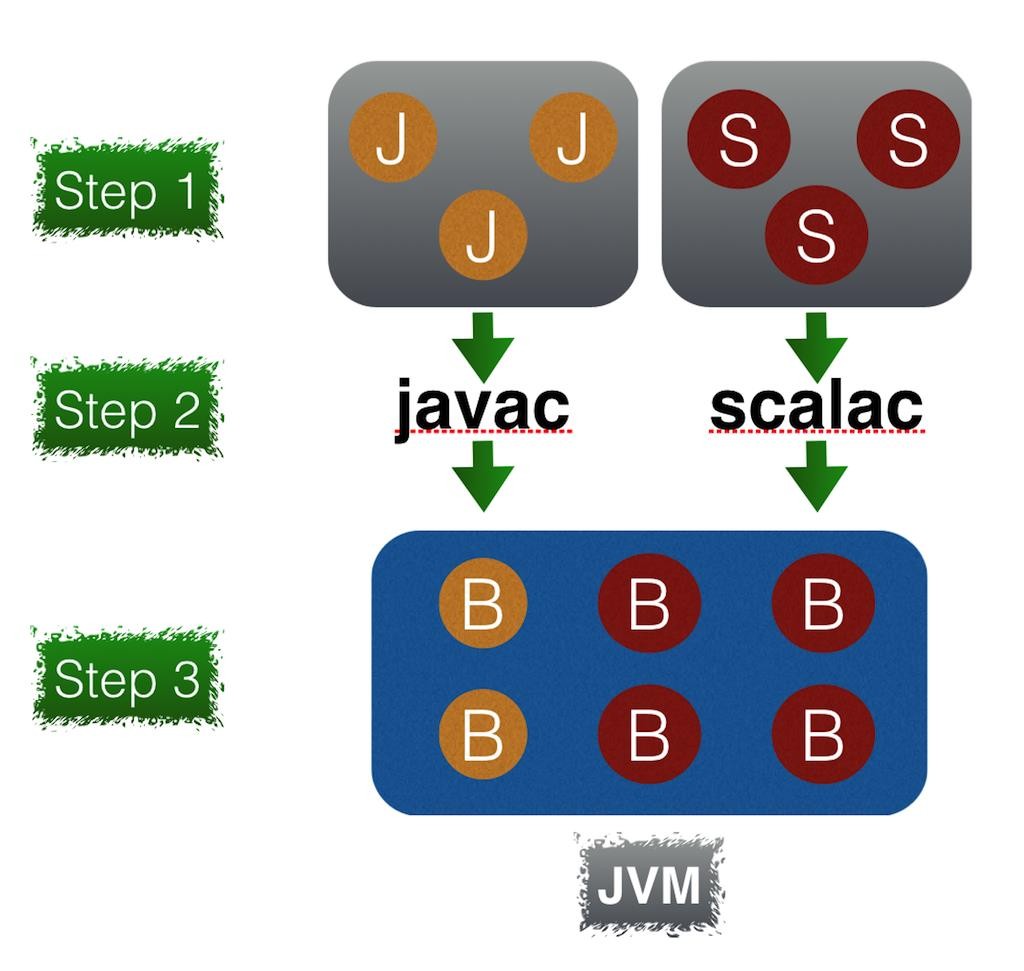
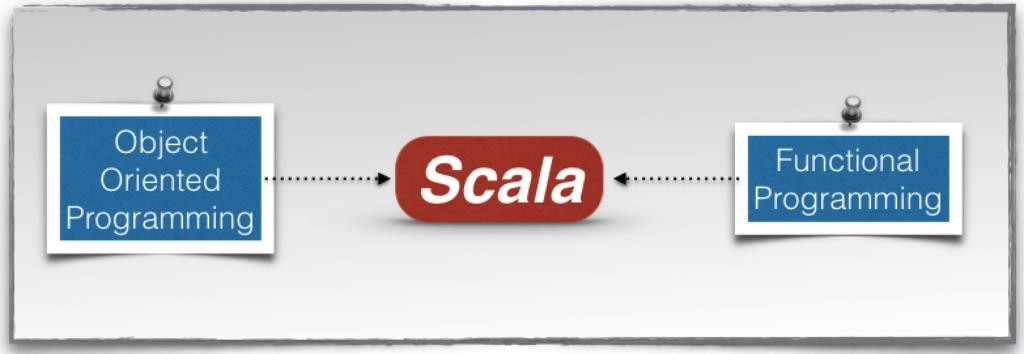
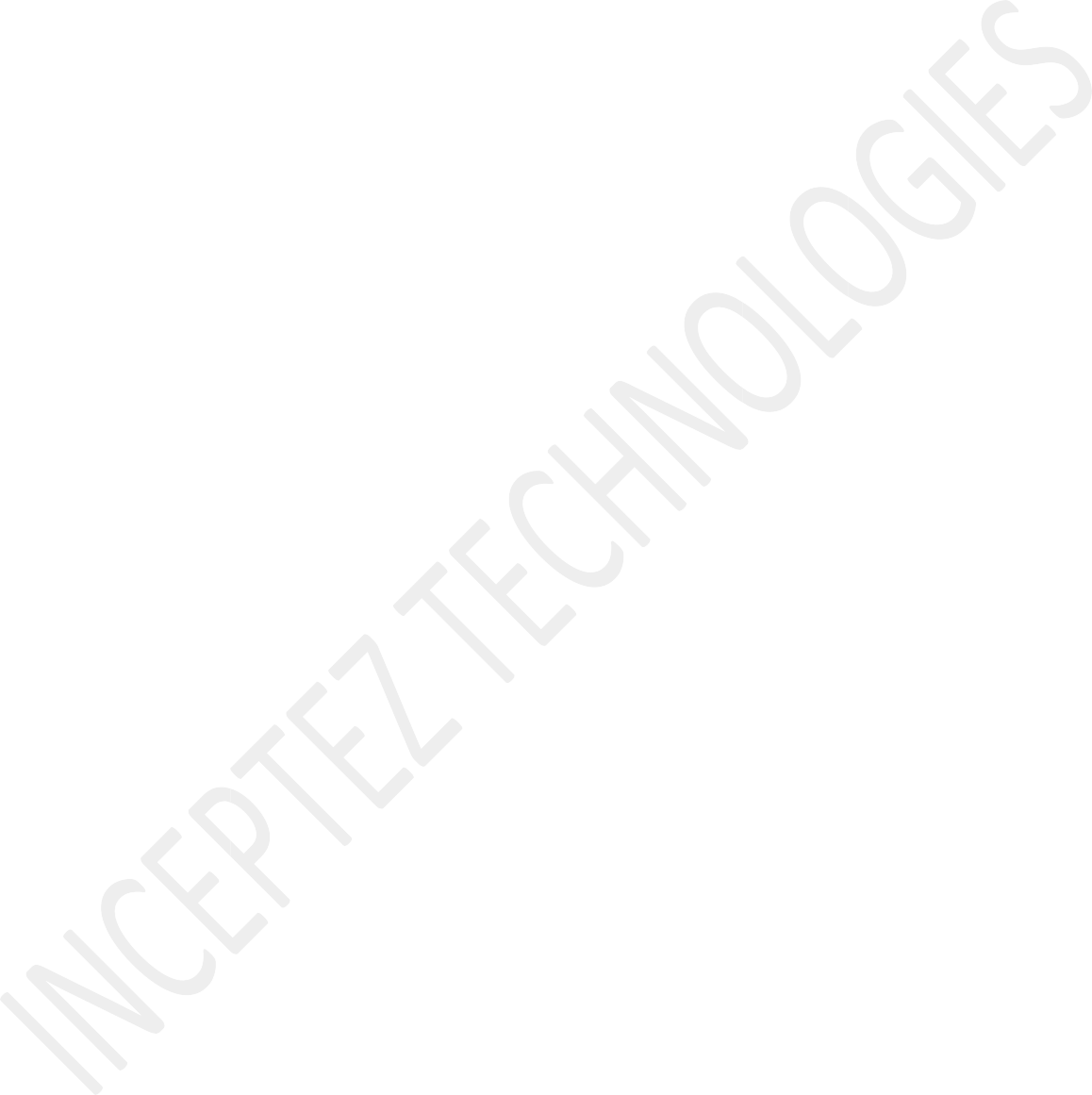
SCALA Introduction and Hands on



# Why Scala?

* First, a developer can achieve a **significant productivity** jump by using Scala.
* Second, it helps developers write **robust code** with reduced bugs.
* Third, Spark is written in Scala, so Scala is a natural fit for **developing Spark applications**.

**Why do you need the JDK for writing Scala applications?**

**Functional Programming**

**What is Functional Programming?**

* Functional programming makes it easier to write **concurrent or multithreaded** applications.
* Functional programming languages make it easier to write elegant code, which is **easy to read, understand, and reason** about.

# Functions – Characteristics First Class

**FP treats functions as first-class citizens. A function has the same status as a variable or value**. It allows a function to be used just like a variable. While in case of any imperative language such as C it treats function and variable differently.

# Compostable

Functions in functional programming are compostable. Function composition is a mathematical and computer science concept of **combining simple functions to create a complex one**.

# No Side Effects

A function in functional programming does not have side effects. The result returned by a function depends only on the input arguments to the function**. The behavior of a function does not change with time. It returns the same output every time for a given input, no matter how many times it is called**. In other words, a function does not have a state. It does not depend on or update any global variable.

# Simple

Functions in functional programming are **simple**. A function consists of a few lines of code and it does only one thing. A simple function is easy to reason about. Thus, it enables robustness and high quality.

# Variables

* Statically Defined and Dynamically Inferred
* **Static inference** means, when we just assing a value to a variable, it automatically detects the type of the value and assigns the value to it.
* For example

scala> val a = 1

a: Int = 1

scala> var nbkid\_dynm = "zkzxh4c"

nbkid\_dynm: String = zkzxh4c

* **Dynamic inference** means, while defining a variable, its data type will be explicitly defined.
* For example

scala> val b:String = "kannan"

b: String = kannan

scala> var empid:Int = 21674304

empid: Int = 21674304

* Mutable – **Var**

It means once the variable is assigned with values, it can be modified later.

For eg:

scala> var a = 10

a: Int = 10

scala> a = 20

a: Int = 20

* Immutable – **Val** Lazy Values

It means once the variable is assigned with values, it can be modified later.

For eg:

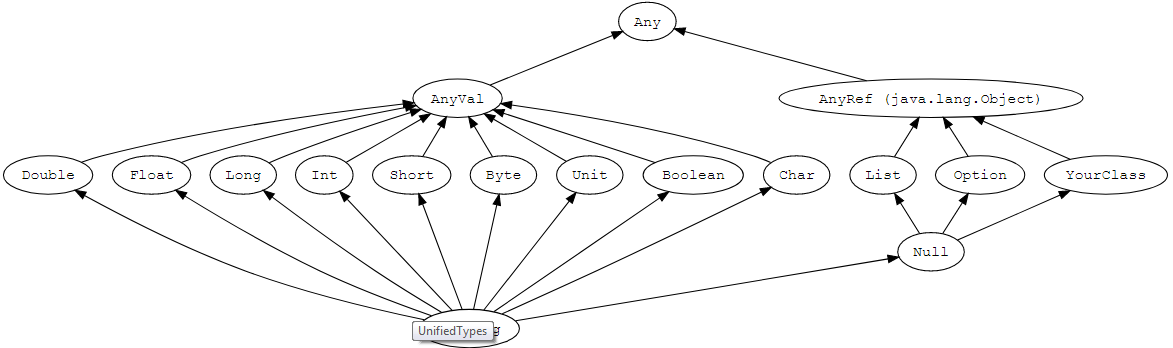
scala> val a = 10

a: Int = 10

scala> a = 20

a: Int = 20

**Scala DataTypes:**



Nothing

* 1. **Value Type**
  2. **Reference**

**AnyVal** is the value type and **AnyRef** is the reference type. **Any** is the super type which accepts both.

* **Any** is the super type of all types, also called the top type.
* It defines certain universal methods such as equals, hashCode, and toString.
* Any has two direct subclasses: AnyVal and AnyRef.

//Any: Super type of any value or reference type

val atype1:Any = "Inceptez" 🡪here is string

val atype2:Any = 100🡪here is int

val atype3:Any = 200.50🡪here is double

val atype4:Any = 'G'🡪here is char

val atype5:Any = null🡪here is null

val atype6:Any = new Student()🡪here is object reference

* **AnyVal** represents value types.
* There are nine predefined value types and they are non-nullable:
  + Boolean
    - true or false
  + Byte
    - 8-bit signed two's complement integer (-2^7 to 2^7-1, inclusive)
    - -128 to 127
  + Short
    - 16-bit signed two's complement integer (-2^15 to 2^15-1, inclusive)
    - 32,768 to 32,767
  + Int
    - 32-bit two's complement integer (-2^31 to 2^31-1, inclusive)
    - 2,147,483,648 to 2,147,483,647
  + Long
    - 64-bit two's complement integer (-2^63 to 2^63-1, inclusive)
    - -9,223,372,036,854,775,808 to +9,223,372,036,854,775,807
  + Float
    - 32-bit IEEE 754 single-precision float
    - 1.40129846432481707e-45 to 3.40282346638528860e+38 (positive or negative)
  + Double
    - 64-bit IEEE 754 double-precision float
    - 4.94065645841246544e-324d to 1.79769313486231570e+308d (positive or negative)
  + Char
    - 16-bit unsigned Unicode character (0 to 2^16-1, inclusive)
    - 0 to 65,535
  + String
    - a sequence of Chars
  + Unit
    - It is a value type which carries no meaningful information.
    - There is exactly one instance of Unit which can be declared literally like so: ().
    - All functions must return something so sometimes Unit is a useful return type.
* **AnyVal**: is the root class of all value types

val av1:AnyVal = 10 🡪here is Int

val av2:AnyVal = 10.0 🡪here is double

val av3:AnyVal = 'Dam' 🡪here is string

val av4:AnyVal = ()🡪here is unit

* If we need to have a variable in a method as input which will access any type of values , then we can declare the input types as AnyVal.. it can accept any values types inputs and can process it inside the method as per the logic written.

* **AnyRef** represents reference types.
* All non-value types are defined as reference types.
* Every user-defined type in Scala is a subtype of AnyRef.
* If Scala is used in the context of a Java runtime environment, AnyRef corresponds to java.lang.Object.

* String: A sequence of characters enclosed with double quotes.
* All custom objects are reference type

val str2:AnyRef = "xybbdxx"

val str22:AnyVal= 10

scala> class student{

| val name:String="kannan"

| val id:Int=21674304

| }

defined class student

scala> val studobj:AnyRef = new student()

studobj: AnyRef = **student@539fc5d1**

* The string which is bolded above is the reference of the class for which instance is created.
* It have the reference(address) of the object studobj.

//Null: corresponds to the null value or empty reference,**only for reference type**

val sobj:Student = null

val str3:String = null

//Anyref: is a super class of all reference types

val aref1:AnyRef = "Inceptez"

println(aref1);

val aref2:AnyRef = new Student()

println(aref2);

The easiest way to get started with Scala is by using the Scala interpreter, which provides an interactive shell for writing Scala code. It is a **REPL** (read, evaluate, print, loop) tool

One key objective of functional programming is to use immutable objects as often as possible.

Try to use operations that transform immutable objects into a new immutable object.

**Control Structures:**

**If condition:**

|  |
| --- |
| **In REPL:**  Just type:  val x = 10  x: Int = 10  val y = 20  y: Int = 20  scala> if (x>y)  | {  | println("x is greater") } else if  | (y>x)  | { println("y is greater") } else  | { println("both are equal")}  y is greater  **In REPL:**  Go to paste mode and paste the code here  scala> :paste  // Entering paste mode (ctrl-D to finish)  if (x > y ) println("x is greater") else if (y>x)  println("y is greater") else  println("x equals y")  // Exiting paste mode, now interpreting.  y is greater  **In Scala IDE:**  Open Eclipse in Scala prespective:  package org.inceptez.scalaprograms  object objconditional3 {  def main(args:Array[String]):Unit =  {  */\* if statement consists of Boolean expression followed by one or more statements.*  *Syntax:*  *if(boolean\_expression) {*  *statements*  *}*  *\* \*/*    var studmarks = 75  if( studmarks > 65)  {  println("Student passed the exam with distinction")  }  //Else  var marks1 = 80  if( marks1 > 65)  {  println("Student passed the exam with distinction")  }  else  {  println("Student mark is less than 65")  }  //Else If  val x =10;  val attendance="p";  var marks = 0;  //mark 70 and 40 - c, mark 71 and 79 - b, mark 71 and 79 - b  if ((marks == 0 | attendance != "p") & x==10)  println("Either Student didnt attended exam or he scored 0");  else  {  if ( attendance == "p")  {  if ( marks < 70 & marks > 40)  {  println("Grade is C")  }  else if(marks == 70)  {  println("Grade is B")  }  else if(marks == 80)  {  println("Grade is A")  }  else if(marks == 90)  {  println("Grade is A+")  }  else  {  println("Student is not performing well hence no grade is awarded")  }  }  }  }  } |

**For loops:**

|  |
| --- |
| package org.inceptez.scalaprograms  object objlooping4 {  def main(args:Array[String]):Unit =  {  //print numbers from 1 to 10  val n=10;  for(i <- 1 to n)  {  println("For Loop : " + i)  }  //################ Output: no 1 to 10 will be printed in ascending  //Decrement from 10 to 1  for(i <- 10 to 1 by -1)  {  println("Negative For Loop : " + i)  }  //################ Output: no 10 to 1 will be printed in descending  //using until which will not include the final number  for(i <- 1 until 10)  {    println("For loop until : " + i)  }  //################ Output: no 1 to 9 will be printed excluding 10. |

**While loops:**

|  |
| --- |
| //using while loop  var i = 0;  while (i <= 10)  {  println("while loop : " + i)  i = i + 1  }    //using do while loop    var j = 20;  do  {    println("Do while : " + j)  j = j + 1    }  while(j <= 10)      // multi level looping  for(i <- 1 to 10; j <- 1 to 3)  {  println("i value:" + i + " j value:" + j)  }  }  } |

# Expressions :

// "Returns" the final value in a block automatically

val x = 10;

x + 20

println ({ val x = 10; x + 20})

**Methods**

**How to Define Method in Scala?**

A method in Scala is defined with the keyword **def**. A method definition starts with the name, which is followed by the comma-separated input parameters in parentheses along with their types. The closing parenthesis is followed by a colon, method output type, equal sign, and the method body in optional curly braces.

**Points to note:**

* The input arguments inside the parenthesis should be comma separate.
* Should be defined with data type.
* If the method is supposed to return any value as output, then after semi-colon we need to declare the output data type along with equal to sign.
* Methods can
  + Return a value
  + Return an expression
  + Not return any value or expression.
* Return data type is optional.

|  |
| --- |
| package org.inceptez.scalaprograms  object methodfuncs5 {  def main(args:Array[String]):Unit =  {  // Methods, accept/returns arguments or not (Unit), overloading , currying , default params  // Functions -> Ananymous, lambda, literals, value functions - quick functionalities  // without considering reuability accross , cant have return keyword  /\* Methods in Scala  Syntax:  def functionName(parameters : typeofparameters) : returntypeoffunction =  {  statements to be executed  }  \* \*/  // FYI, the method add1/2/3…. Are declare below.  //function called with arguments inside println function.  println(add1(100,200));  println(add2(100,300));  println(add3(100,400));  println(add4(20,3));  println(add5(200,500))  println("add5 curly braces is optional if there is no more than 1 expression")  println(add6(100));  println(add6("Inceptez"));  add8();  //Function call with named arguments  add8(b = 20,a = 10);  add8;  add9(15)(30);    //Anonymous function or Lambda function or Function literal  println(add10(25,40));    println(sub(20,50))  println(add11(50,10))    }  // no return values  def add1(a:Int,b:Int):Unit =  {  println("add1 No return values")  val c = a + b;  println(c)  //return c;  }  // return an expression  def add2 (a:Int,b:Int):Int =  {  println("add2 Return an expression")  return {  val x=a  a + b  };  }  //return keyword optional  def add3(a:Int,b:Int):Int =  {  println("add3 Return keyword in optional")  a + b;  }  //return datatype is optional  def add4(a:Int,b:Int)=  {  println("add4 Return datatype is optional")  val x = a.toFloat/b.toFloat;  println(x)  //return x  //x  }  //for single statement curly braces are optional  def add5(a:Int,b:Int) = a + b |

# Method over loading:

# Method overloading is,

# Same method names with different arguments.

# Consider when we need to create a method in a class which will be used by different users.

# Different kind of users should be able to access the same method for different functionalities

# But the name should be same.

# For example,

# Lets consider we have a requirement to show the project synopsis to different team.

# We create a method “projInfo” with an argument high\_level\_det:string to give it to non tech team . def projInfo(highdet:string)

# For a tech team we create a method with same name “projInfo” with different aruguments such as high\_level\_det , tech\_det. Def projInfo(highdet:String, techdet:String)

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

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

|  |
| --- |
| // Overloading  def add6 (a:Int):Int =  {  println("add6 Function overloading")  return a + 10  }  def add6 (a:Any):Any =  {  println("add6 Function overloading with any")  return a + a.toString()  }  // Overloading  def add6(a:String):String =  {  println("add6 Function Type overloading")  return "Hello " + a;  }  //Parameter example with default value  def add8(a:Int=30,b:Int=10) =  {  println("add7 Example for Default Arguments")  println(a + b)  }  // Method with no arguments  def add8 =  {  println("add8 with no arguments")  println(100 + 50)  }  //add6(20,30);  //currying  def add9(a:Int) (b:Int):(Int,Int) =  {  println("Currying method with multiple input and output arguments")  val c = a + b;  println(c)  return (c,b);  }  //add9(10)(20);  //store function in a variable  // Anonymous function or Lambda function or Function literal  val add10 = (a:Int,b:Int) =>  {  println("Anonymous add10 function call, cant have return, only last expression returns")  a - b  a + b  }  //Also we can define the same function in the short form  var sub = (a:Int,b:Int) => a - b  //Also we can define the same function in the short form  var add11 = (\_:Float) + (\_:Float) |

Scala allows a concise version of the same function, as shown next.

def add(firstInput: Int, secondInput: Int) = firstInput + secondInput

add(5,10)

The second version does the exact same thing as the first version. The type of the returned data is omitted since the compiler can infer it from the code. However, it is recommended not to omit the return type of a function.

The curly braces are also omitted in this version. They are required only if a function body consists of more than one statement.

**Pattern matching:**

Pattern matching is a mechanism for checking a value against a pattern or a value.

A successful match can also deconstruct a value into its constituent parts. It is a more powerful version of the **switch** statement in Java or **evaluate** statement in Cobol and it can likewise be used in place of a series of if/else statements.

Lets see an example to understand how patten matching works.. also we can see eg for try/catch/finally.

|  |
| --- |
| package org.inceptez.scalaprograms  object patternmatch6 {    /\*Pattern matching is a way of checking the given sequence of tokens for the presence of the  \* specific pattern.  \* It is a technique for checking a value against a pattern.  \*/    def main(args:Array[String]):Unit =  {    /\*Exception handling is a mechanism which is used to handle abnormal conditions.  \* Avoid termination of your program unexpectedly.  \*/    //here is the example for try/catch/finally.  //among try/catch/finaly, the logic inside try will be executed by default .  //if any exception needs to be handled, that logic will be placed inside catch.. the logic inside catch will be //executed only when there is any corresponding exceptional errors occurred in try block logic.  //the logic finally will be executed at last by default..  try  {  val y = 10/1  println("first statement executed and result is : " + y);  val z=Array(10,20,30);  //z(2);    //val x = 10/0  println("All statements executed");  println(y)      }  catch  {  case ex: java.lang.Exception =>  {  println("Some exception occured")  println("I will be executed when error occured");  println("calling test method with param as 2")  println(testmatch(2));  }  case ex: java.lang.ArrayIndexOutOfBoundsException =>  {  println("Array index should be given properly")  println("I will be executed when error occured");  println("calling test method with param as 1")  println(testmatch(1));  }      }  finally  {  println("I will be executed at any cost");  println("calling test method with param as 1")  println(testmatch(1));  //println(testcaseconditional(x=11))  }      }    // method containing match keyword  // its similar to SWITCH statement in java or EVALUATE statement in cobol.  //if the input value matches the value specified in the case statement, corresponding logic written inside //the case statement will be executed.  //In the below example, if the input is passed as 0 thru x, then “Hello, Techies” will be printed.  def testmatch(x:Int) = x match  {  // if value of x is 0,  // this case will be executed  case 0 => "Hello, Techies"    // if value of x is 1,  // this case will be executed  case 1 => "Are you learning Scala?"    // if x doesnt match any sequence,  // then this case will be executed  case \_ => "Good Luck!!"  }    //similarly the matching variable can be as, one of the input.. here op is the matching variable and its one //of the input for the method.  //here the condition can be as OR condition.  def calcuator(a:Int,b:Int,op:String):Any =  op match  {  case "add" | "addition" =>  {  println("Add Numbers")  a + b  }  case "sub" | "subtract" =>  {  println("Sub Numbers")  a - b  }  case "mul" | "multiply" =>  {  println("Multiply Numbers")  a \* b  }  case \_ =>  {  println("Operation Not matched")  "No Match"  }  }    val a = 100  val b = 100    // method containing match keyword  // here based on the input match, we determine a condition.  def testcaseconditional(x:Int) = x match  {    // if value of x is 0,  // this case will be executed  case i if (x == 0) => {if (a == b) {println(a)}}    // if value of x is 1,  // this case will be executed  case j if x > 0 & x < 10 => {if (a != b) {println(b)}}    // if x doesnt match any sequence,  // then this case will be executed  case \_ => "Good Luck!!"  }    } |

**Collection types:**

Collections are the container of things which contains random number of elements. All collection classes are found in the package scala.collection.

Collections are of two types –

Mutable Collections

Immutable Collections

* Mutable Collection – This type of collection is changed after it is created. All [Mutable collection classes](https://intellipaat.com/tutorial/spark-tutorial/programming-with-rdds/) are found in the package scala.collection.mutable.
* Immutable Collection – This type of collection will never change after it is created. All Immutable collection classes are found in the package scala.collection.immutable.

**Types of collections:**

* **Sequence**
* **Array**

It’s a collection of similar data items which are mutable.

* **List**

Kind of linked list. Similar to Array. But its immutable.

* **Map**

A Map is a collection of key/value pairs. Any value can be retrieved based on its key.

* **Set**

A set is a collection of pairwise different elements of the same type.

* **Tuples**

Unlike an array or list, a tuple can hold objects with different types.

* **Options**

Option[T] provides a container for zero or one element of a given type.

* **Iterators**

An iterator is not a collection, but rather a way to access the elements of a collection one by one.

**Let’s see some examples for each collection items.**

List: ########################

scala> **var seqvar = Seq(1,2,3,4,5)**

seqvar: Seq[Int] = List(1, 2, 3, 4, 5)

* Sequence gives List by default.

scala**> seqvar(3)**

res55: Int = 4

* while referring the data back, we need to fetch with offset value which starts from zero.

scala> var seqvar = List[Int](1,"two",3)

<console>:23: error: type mismatch;

found : String("two")

required: Int

var seqvar = List[Int](1,"two",3)

^

* In the above code we try to add a new item which is of different data type. Since the list was initially created as INT, its throwing error.
* So we create the list as the type ANY which can accept any data type.

scala> var seqvar = List(1,"two",3)

seqvar: List[Any] = List(1, two, 3)

scala> seqvar(1) = 50

<console>:26: error: value update is not a member of Seq[Int]

seqvar(1) = 50

^

* Also LIST is immutable.

ARRAY:########################

scala> var arrvar = Array(1,2,3,4,5)

arrvar: Array[Int] = Array(1, 2, 3, 4, 5)

scala> arrvar(2) = 100

scala> arrvar

res58: Array[Int] = Array(1, 2, 100, 4, 5)

* Array is mutable.. so as show above, we have added a new item to the array.

scala> arrvar(5) = 200

java.lang.ArrayIndexOutOfBoundsException: 5

... 48 elided

* But array cant be extended.. the existing values can be update/removed. But cant accommodate a new item after creation of an array.

MAP:##########################

scala> var map1 = Map("kannan" -> 14 , "vandhana" -> 24, "jvtk" -> 100)

map1: scala.collection.immutable.Map[String,Int] = Map(kannan -> 14, vandhana -> 24, jvtk -> 100)

scala> map1("kannan")

res60: Int = 14

* map is a key,value pair kind of collection type.
* To access the map, we need to give the key and get its value.

scala> map1 = ("selvi" -> 50)

<console>:25: error: type mismatch;

found : (String, Int)

required: scala.collection.immutable.Map[String,Int]

map1 = ("selvi" -> 50)

* Maps are immutable by default. To make is mutable we need to define it by using scala.collection.mutable.Map class.

SET:########################

scala> var setvar = Set("one","one","two")

setvar: scala.collection.immutable.Set[String] = Set(one, two)

* in the above example we can see that Set wont allow duplicates.

scala> var setvarmut = scala.collection.mutable.Set("one","two","three")

setvarmut: scala.collection.mutable.Set[String] = Set(three, two, one)

scala> setvarmut.add = "four"

<console>:27: error: missing argument list for method add in trait SetLike

Unapplied methods are only converted to functions when a function type is expected.

You can make this conversion explicit by writing `add \_` or `add(\_)` instead of `add`.

val $ires10 = setvarmut.add

^

<console>:25: error: reassignment to val

* Setvarmut.add is not the right way to add a new item to this collection item.
* The correct method of item addition is show below.

scala> setvarmut.add("four")

res62: Boolean = true

scala> setvarmut

res63: scala.collection.mutable.Set[String] = Set(four, three, two, one)

TUPLE:#############################

scala> var tup = (91,"two",3)

tup: (Int, String, Int) = (91,two,3)

scala> tup.\_2

res61: String = two

setvarmut.add = "four"

^

All the above discussed collection items are shown in a program format in the below box.

|  |
| --- |
| package org.inceptez.scalaprograms  //to work with array and array buffer we need to import below classes.  //both array and array buffer are mutable. Array is of **fixed** **length**. Array Buffer is **resizable**  import scala.collection.mutable.ArrayBuffer  import scala.collection.mutable.Seq  import Array.\_  object collections7 {  /\*  **\* Scala has rich set of collection library**  **\* Collection are the containers that holds elements**    **\* 1. Seq - linear collection of element may be indexed(array) or linear list(list)**  **\* 2. Map - contains a collection of key value pairs**  **\* 3. Set - collection that contains no duplicate values**    \*/  def main(args:Array[String]):Unit =  {  /\*  \* Seq -  \* - mutable Array  \* - immutable List  \*  \* \*/    //LIST  // Also we can define as  var s2 = Seq(10,20,30,40,50)  // or var s2 = List(10,20,30,40,50)  //if we declare a variable as Seq, by default it will be created as list.  // while accessing the data, it should be based on offset value.. starts from zero.  //s2(4) =50  println("This shows Seq is mutable " + s2(4))    //ARRAY  //An array is sequential and is of a fixed size.  //Array - Array in scala is sequential, fixed in size and mutable    //Declare Array with 1 element with type of Int    //val ar = Array(Array(5,"hi"),Array(6,"Hello"))  val ar = Array[Float](5);  println("result of Array[Int](5) is : " + ar(0))    //or    val ar1 = Array(1,2,3,4,5)    println("Fourth element of Array(1,2,3,4,5) is : " + ar1(3))    //Creating array with range  //var ar2 = range(start\_value, end\_value,incremental\_value)  val ar2= range(2,15,2)  println("Second element of range(2,15,2) is : " + ar2(1))  val ar3 = range(15,2,-2)  println("Second element of range(15,2,-2) " + ar3(1))  println(ar2(1));  println(ar1.length)    ar1.sorted.take(3).foreach(println)  ar1.tail  println(ar1.length)  println(ar1.isEmpty)  ar1.sorted  ar1(2)=10;    ar1(3)=100;    /\*  \* A list of Scala Collections is much like a Scala array.  \* Except that, it is immutable and represents a linked list.  \* An Array, on the other hand, is flat and mutable.  \*/    val lst = List(10,20,30,40)    val fruits = List("apples", "oranges", "mangoes")    val g = fruits(0)    var list = List(1,8,5,6,9,58,23,15)  //var list11 = List(List(1,"a",100000),List(2,"b",200000))    //Access value from the list  //list(2) = 10  val lstval = list(0)    //Merging 2 list  var list3 = list ++ lst  //or  var list4 = list ::: lst    println(list3)  println(list4)  println(list.contains(2))    println(list4.head)  list4.tail.foreach(println)  println(list4.length)  println(list4.isEmpty)  list4.sorted.foreach(println)  list4.reverse.foreach(println)      /\*  Tuples are immutable, contains fixed length of different type elements  \*/    val emp = (101,"Karthik",200.00)  val empid = emp.\_1  val empname = emp.\_2  val empsal = emp.\_3    val emp1 = (101,"Karthik",200.00,("New Street","Chennai","TN"))    val empcity = emp1.\_4.\_2      //// ARRAY PROGRAMS    val ar222=Array[Int]();  if (ar222.isEmpty)  {  println("Array is empty");  }    //1. write a program to create an Int array with 5 different value and sum all the values      /\*  \* A Map in Scala is a collection of key-value pairs, and is also called a hash table.  \* We can use a key to access a value.  \* Keys are unique and values may be in common  \* Map is by default immutable  \* \*/    var m = Map("Mani" -> 10000,"Karthik" -> 20000)    //Add an element  m += ("Raj" -> 30000)    //Remove an element  m -= ("Mani")    //println(m)  var m2 = Map.empty[String,String]  //Add multiple Map elements    m2 += ("1"->"A","2"->"B")  println(m2)    //Immutable map, doesn't allow to modify but allows to add or remove by recreating    var immutablemap = scala.collection.immutable.Map(1 -> "Alto",2 -> "Swift")    immutablemap -= (2)    //Below update is not possible  //immutablemap(1)="Alto k10"    var mutablemap = scala.collection.mutable.Map(1 -> "Alto",2 -> "Swift")  mutablemap(1)="Alto k10"    m.keys  m.values  m.isEmpty    val lst1 = m.toList  val arr = m.toArray    /\* A Scala Set is a collection that won’t take duplicates.  \* By default, Scala uses immutable sets.  \* \*/    var mutablegames = scala.collection.mutable.Set("Cricket","VollyBall","BaseBall","Hockey")    println("Due to mutable, I am able to modify")  mutablegames.add("Chess");  mutablegames.add("Hockey");  mutablegames.remove("Cricket");  println(mutablegames)    var games = scala.collection.immutable.Set("Cricket","Football","Hockey","Golf","Cricket","Football")  println(games)    println("Due to Immutable, I am not able to modify, uncomment the below code and check")  //games.add("Tennis");  //games.remove("Cricket");    println("Either mutable or immutable, I can reassign if its of type var");  //val games = scala.collection.immutable.Set("Cricket","Football","Hockey","Golf","Cricket","Football")    games += "Tennis"  games -= "Cricket"    println(games);  println(games.head) // Returns first element present in the set  println(games.tail) // Returns all elements except first element.  println(games.isEmpty) // Returns either true or false    println(games.max)  println(games.min)  println(games.contains("Tennis"))  println(games ++ mutablegames);  println(games.intersect(mutablegames))  println(games.union(mutablegames))  println("Set difference");  println(games.diff(mutablegames))    }  } |

**Higher-Order Methods**

A method that takes a function as an input parameter is called a higher-order method. Similarly, a high order function is a function that takes another function as input. Higher-order methods and functions help reduce code duplication. In addition, they help you write concise code. The following example shows a simple higher-order function.

val salaries = Seq(20000, 70000, 40000)

def bonus(a:Int):Double =

{ (a\*1.5)

}

val normalmethod = salaries.map (a=>(a\*1.5) )

val higherordermethod = salaries.map (bonus)

in the above code, the val **normalmethod** is a typical way of accessing a function.

But the val **higherordermethod** is the concept of higher order function. It passes a function as argument to another function. Map is a function and bonus is another function

Example with sample program.

|  |
| --- |
| package org.inceptez.scalaprograms  object highordcaseclassclosure8 {  case class emp(id: Int, name: String, address: String)  def main(args:Array[String])  {  /\* Higher-Order Methods - scala treats functions/methods as a variable, hence it can be passed  \* as a param for another function/method  A method that takes a function as an input parameter is called a higher-order method. Similarly, a  highorder function is a function that takes another function as input. Higher-order methods and  functions help reduce code duplication. In addition, they help you write concise code. The following  example shows a simple higher-order function.\*/    def bonus(a:Int):Double = ((a\*1.5))  val salaries = Array(20000, 70000, 40000)  val normalmethod = salaries.map (a=>(a\*1.5))  val higherordermethod = salaries.map(bonus)  println(higherordermethod(0));    /\*Case Classes  A case class is a class with a case modifier. An example is shown next.  All input parameters defined implicitly treated as Val, Useful for immutable objects and pattern  matching , Creates Factory method automatically\*/    val request = emp(1, "Sam", "1, castle point blvd, nj")  println(request.name)  //request.address |

**Closures**

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

var bonuspercent = .10

def bonus = (i:Int) => {i+(i \* bonuspercent)}

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

in the above code the value which is printed is depended on the value bonuspercent which is declared outside the method.

|  |
| --- |
| /\*Closures  A closure is a function, whose return value depends on the value of one or  more variables declared outside this function.\*/  var bonuspercent = .10  def bonus1(i:Int) =  {i+(i \* bonuspercent)}  println( "Bonus value is = " + bonus1(100) )  } } |

# Few other SCALA methods:

|  |
| --- |
| package org.inceptez.scalaprograms  object fewscalamethods9 {  def main (args:Array[String])  {  //Map - Evaluates a function over each element in the list, returning a list with the same number of elements.  val lst = List(1,2,3,4,5,6,7,8,9,10)  lst.map( x => x\*2 )  println(lst)    //Filter - Removes any elements where the function you pass in evaluates to false.  lst.filter((i: Int) => i % 2 == 0)    //write a program to find prime numbers in the list    //foreach is like map but returns nothing.  lst.foreach(x => println(x))    val Icecreamprice = List(25,13,45,34,25)  val total = Icecreamprice.reduce((a,b) => (a + b))    val donutPrices: List[Double] = List(1.5, 2.0, 2.5)  val sum: Double = donutPrices.reduce(\_ + \_)    val prices: Seq[Double] = Seq(1.5, 2.0, 2.5)  println(s"Icecream prices = $prices")  println("\nHow to sum all the icecream prices using fold function")  val sum1 = prices.fold(0.0)(\_ + \_)  println(s"Sum = $sum1")    //flattern  val lst1 = List(List(1,2,3,4),List(5,6),List(7,8)).flatten  lst1.foreach(println)  }  } |

# OOPS

# Packages:

# It’s a collection of classes, functions, methods etc……

# Classes

* A class is a template or blueprint which can consist of methods/functions/objects/variables etc..
* A class definition starts with the class name, followed by comma-separated class parameters in parentheses, and then fields and methods enclosed in curly braces. An example is shown next.

# Objects

* An object is an instance of a class.
* A class is defined in source code, whereas an object exists at runtime. A class is defined using the keyword class.
* An instance of a class is created using the keyword new.
* It consists of two types.
  + Single ton object
    - Its similar to a class.
    - But the only difference is, it doesn’t need an object to create an instance.
    - Its gets instantiated automatically when the program is executed.
    - Single ton object will get instantiated only once and can be instantiated only once in program. Where as a regular class can be instantiated number of times with different values as per our requirement.
    - Generally the class which has main method will be defined as single ton object, so that it gets instantiated automatically when the program is executed. Because anyways main method is the place where the execution starts.
  + Regular object
    - Its an instance of the referred class.
    - The logic behind this is,
      * When a program is executed, its loaded in the memory.
      * So if it refers a method from another class, then that class should also loaded in the memory while execution.
      * To serve that purpose, we construct a memory area using the key work “new”(which is the **constructor**).
      * Now a memory is created and the class along with its properties like variables, methods are loaded into that memory.
      * Now the memory needs a name to be referred by other class.
      * That name is called as object.
      * For eg: val sc = new sparkContext()
      * In the above example a memory area is constructed for the class “sparkContext()” using “new” and its named as “sc”
      * So “sc” is the instantiated object for the class sparkContext().
      * Now with the name “sc” we can access all the methods written inside the class sparkContext()
      * Eg: val filedata = sc.textFile(“file:/home/hduser/data.txt”)

|  |
| --- |
| class salary(val p1: Int, val p2: Int, val p3:Int) {  var hike: Int = p1  var penality: Int = p2 var incentive:Int = p3  def hike(id:String,sal: Int)  {  val netsal = hike + sal -penality  println ("nett salary for empid " +id +" is "+ netsal);  }  def grosssalary(id:String,sal:Int)  {  val grosssal = hike + sal +incentive-penality  println ("gross salary for empid " +id +" is "+ grosssal);  }  }  //every class should have a main method.. the execution starts from the main method.  object Salcalc {  def main(args: Array[String]) {  val sal = new salary(10000,3000,args(0).toInt);  sal.hike("a1",30000);  sal.hike("a2",40000);  sal.grosssalary("a1",25000);  sal.grosssalary("a2",35000);  }  }  ////////////////////////////////////////////////////////  Salcalc.main(Array("5000"))  /\* if we concentrate on the highlighted part of the above code, salary is a class.  In another class salcalc which has main method, the class salary is invoked  In the line “val sal = new salary(10000,3000,args(0).toInt);”   * A memory is allotted for the class salary by using the constructer key work ‘new’ * Then the memory is named as ‘sal’ and its called as object. * Now the class salary and its methos/functions/etc will be in memory with name ‘sal’ which is object. * We can access any of the properties of class salary using its object. |

|  |
| --- |
| Concept:   * Class * is a template which will have a collection of methods, functions, objects, variables etc… * it can be considered as a big page in which we write all out functionalities.. * some other developer can access any content from this page referring to the class name. * objects * its an instance for a class. * Consider we write a program and we need to access some method from another class. * When our code is getting executed it will be loaded into memory. * When some other method from some other class is referred in our program, that class along with all its methods also needs to be loaded into memory. * So we need to create a memory location and need to load the other class into the memory. * The name for that memory is object for that class. |

Main method and custom method.

# Case Classes

* A Scala Case Class is like a regular class, except it is good for modeling immutable data.
* It also serves useful in pattern matching, such a class has a default apply() method which handles object construction.
* A scala case class also has all vals, which means they are immutable.
* Creates Factory method automatically

1. scala> case class **Song**(title:String,artist:String,track:Int)

now lets instantiate the above case class using an object “stay” and then call the invoke the class using that object.

1. scala> val stay=**Song**("Stay","Inna",4)
2. stay: Song = **Song**(Stay,Inna,4)

now we can access the argument of the case class with its name. since the arguments of case case is “val” (immutable) we cant modify it.

1. scala> stay.title
2. res1: String = Stay
3. And now, let’s try modifying it.
4. scala> stay.title="Me Gusta"
5. <console>:12: error: reassignment to val
6. stay.title="Me Gusta"

^

|  |  |
| --- | --- |
| **Normal class** | **Case class** |
| * Creating a constructor for normal class requires the key work ‘new’ * The values passed to the normal class can be modified later inside the class * By default pattern matching is not supported. * Normal class can extend another class | * Creating a constructor for case class doesn’t require the key work ‘new’ * Values passed to a case class is immutable. (val) * By default pattern matching is supported. * Case class cannot extend other class. |

# Abstraction:

Abstraction is the concept of hiding the logic behind the functionality to the user.

For eg:

* sparkContext() is a abstract class created by apache spark.
* While involving in spark programing, we use the class “sparkContext” to make use of its method.
* But we don’t bother of knowing the logic written inside the method from sparkContext..
* That’s called , those logic inside sparkContext is abstracted(hidden) from its user.

# Traits

A *trait* represents an **interface** supported by a hierarchy of related classes. It is an **abstraction**

mechanism that helps development of **modular, reusable, and extensible code**

A trait looks similar to an **abstract class**. Both can contain fields and methods. The key difference is that a class can inherit from only one class, but it can inherit from any number of traits. An example of a trait is shown next.

|  |  |
| --- | --- |
| **Abstract class** | **traits** |
| * Parameter/arguments are supported by this class. We can pass values while invoking this abstract class. * A class can only inherit from only one class. | * Parameters/arguments are not applicable here. * A class can inherit multiple traits. |

|  |
| --- |
| trait Shape {  def area(): Int ;  def plintharea(plintharea1:Int,reduced: Int): Int =  {  val reducedarea = plintharea1 - reduced;  println ("Final plinth area calculated is " +reducedarea);  return reducedarea;  }  }  trait landarea {  def totarea(length:Int,breadth: Int): Int = { val total = length\*breadth;  return total;  }  }  class PlinthClass(length: Int,breadth:Int) extends Shape with landarea{  def area = length \* length ;  val reducedarea = area/4;  def plinth = plintharea(area, reducedarea);  println("Total area value is " +area);  println("Reduced area value is " +reducedarea);  println("Plinth area value is " +plinth);  def ta = totarea(length,breadth);  println("Total area returned is : " + ta);  }  val plinthinstance = new PlinthClass(20,40)  val pli = plinthinstance.area  val pli = plinthinstance.ta |

**Below example show the concept of OOPS.**

|  |
| --- |
| package org.inceptez.scalaprograms  **/\*Singleton object is an object which is declared by using object keyword instead by class.**  **No object is required to call methods declared inside singleton object.**  **In scala, there is no static concept.**  **So scala creates a singleton object to provide entry point for your program execution. \* \*/**  object oops8 {  // **Companion Object Class**  // An object with the same name as a class is called a companion object. Conversely, the class is the object's //companion class. A companion class or object can access the private members of its companion.  // lets assume we have a private variable declared in a parent class.  //when another class extends the parent class, it cant access the private variable form its inherited parent.  //In order to access that private variable, this class should be declare as a companion class.  //to declare the child class as companion class, the name of the child class should be as same as parent.  //so child becomes companion of parent and it can access all properties of parent including parents private.  def main(args:Array[String])  {    def polymorph(a:Any):Any=  {  return a    }    println(polymorph("hello"))  println(polymorph(10))  val obj1=new oops8();  println(obj1.x);    }    }  class oops8 {  private val x= "private value";  };  trait trait1{  def tr1(b:Int);  }  trait trait2{  def tr2(b:Int);  }  class classtrait extends trait1 with trait2  {  val p1=10;  val p2=20;  def tr1(b:Int):Unit= b+p1;  def tr2(b:Int):Unit= b-p2;  } |

**Below bank class example will be having all concepts of OOPS..**

|  |
| --- |
| package org.inceptez.scalaprograms  /\* OOPS  \* SPARK DEPENDENT  which must be known for Spark Develpment  \* PACKAGE  its collection of methods/functions/objects/variables etc .  \* CLASS  Its template which has set of methods/functions/variable which is of particular functionality  \* OBJECT  **singleton obj/class** – its similar to class. Only difference is its naturally instantiated. No need to create instance separately.  **Normal obj** – it’s the name which is used to point out the instance memory created for its corresponding class.  \* main or custom METHODs/FUNCTIONs  **Main method** – it’s the method inside a class which is the start point of execution. Only one main method per class is allowed.  **Customized method –** is the method created inside a class other than main method. Many methods can be created in a class based on different logics for each method.  \* CONSTRUCTOR  **Primary constructor**- when an instance is create by constructor for a class, we pass arguments if the class supports input arguments.  **Auxiliary constructor** – when the instance is create by contructor for a class, if we don’t want to pass arguments even if the class supports input arguments, then that class should have an inbuilt input value defined using the **“this”** keyword . so that the value defined in the method using “this” keyword will be used inside that class.  This concept will be explained in the below example.  \* CASE CLASS  It’s a template which can accept constant values as input and it will remain immutable.  \* HIGHER ORDER METHOD/FUNC  If a function/method is passed as an argument to another function/method, then its called higher order function/method.  For eg: **sc.foreach(println)**  Foreach is a function and println is another function.  \* CLOSURE  \* if the output of a method is dependent of a value declared outside that method, its closure.  \* OOPS Concepts made simple with Scala program implementation:  \*  \* All these are Classes -> Class, singleton object (instantiated class),case class, abstract class, traits  \*  \* Polymorphism - Polymorphism means that a function type comes "in many forms".  \* the type can have instances of many types.  \* Example - Method with different type of arguments  Eg: var a = 100  Var b = 50  Var c = “sub”  def operations(a:Int,b:Int,c:String) :Int =  { a match {  case “add” -> return (a+b)  case “sub” -> return (a-b)  case “div” -> return (a/b)  case \_ -> return (a\*b)  }  }  The output of the method “operation” is dependent on the variables a,b and c declared outside that method.  \*  \* Overloading - Scala Method overloading is when one class has more than one method with the  \* same name but different signature.  \* This means that they may differ in the number of parameters, data types, or both.  \* Example : Method with different number of arguments  \*  \* Abstraction - Abstraction is the process to hide the internal details  \* and showing only the functionality.  \* abstraction is achieved by using an abstract class.  \* Example : Abstract class and Traits  \*  \* Inheritance -  \* Inheritance is the process of inheriting the feature of the parent class  \* Multiple Inheritance: In Multiple inheritance ,one class can have more than one superclass and inherit features from all parent classes.  \* Scala does not support multiple inheritance with classes, but it can be achieved by traits.  \* Example - Abstract Class and Traits  \*  \* Overriding - Scala overriding method provides your own implementation of it. When a class inherits  \* from another, it may want to modify the definition for a method of the superclass or  \* provide a new version of it.  \* This is the concept of Scala method overriding and we use the 'override' modifier to implement this.  \* Example : Method or vals override with different implementations  \*  \* Encapsulation - Specify access specified/modifier for providing access control to the objects or  \* values  \* Example : private var a=100;  \*  \* Companion Object - It is again a singleton object called as companion if we create the object in the same name of the instantiating class  \* using companion object we can able to access the encapsulated members of the class.  \*/  /\*  Class -  A class is a user defined blueprint or prototype from which objects are created.  It represents the set of properties or methods  \* \*/  //Primary Constructor  class bankclass(actype:String,intpct:Double) {    //Auxilary Constructor  def this(actype:String)  {  this(actype,0.0);  }    def cust(amt:Double):Double=  {  return {  if (actype == "SB")  amt+(amt\*intpct)  else  amt  }  }  }  /\*  Singleton object is an object which is declared by using object keyword instead by class.  No object is required to call methods declared inside singleton object.  In scala, there is no static concept.  So scala creates a singleton object to provide entry point for your program execution. \* \*/  object singletonbankobj{  val welcome= "Singleton Object Initialized with his members"  }  /\*Object -  An object is an instance of a class. Objects have states and behaviors.\*/  //object can't have params  object bankclassobj  {    def main(args:Array[String])  {  println(singletonbankobj.welcome);    println("Primary constructor will be initialized");  //Primary constructor will be initialized  val bankclassobjinstance1= new bankclass("SB",8.5);    println("Auxilary constructor will be initialized");  //Auxilary constructor will be initialized  val bankclassobjinstance2= new bankclass("CU");      }  }  //Abstraction, Inheritance  //Trait can't have params  trait cardtrait  {  def cardtype(ctype:String,withdrawlimit:Long):Int;  }  trait banktypetrait  {  def banktype(btype:String):Int= btype match  {  case "Investment" => 10  case "Retail" => 20  case \_ => 30  }    class bankclassinherit extends cardtrait with banktypetrait  {  def cardtype (ctype:String,withdrawlimit:Long):Int= ctype match  {  case "Credit" => 100000  case "Debit" => 20000  case \_ => 10000  }    val btype=banktype("Investment");  }    //Can pass arguments, only 1 abstract class can be extended, can have implementation or not.  abstract class cardtrait1(inputargispossible:Int)  {  val argassign=inputargispossible;  def cardtype(ctype:String,withdrawlimit:Long):Int;  }  abstract class banktypetrait1  {  def banktype(btype:String):Int= btype match  {  case "Investment" => 10  case "Retail" => 20  case \_ => 30  }  }    class bankclassinherit1 extends cardtrait1(100) //with banktypetrait1  {  def cardtype (ctype:String,withdrawlimit:Long):Int= ctype match  {  case "Credit" => 100000  case "Debit" => 20000  case \_ => 10000  }    val btype=banktype("Investment");  }    /// Overriding, Companion Object, Encapsulation  class bankclassobj  {  private def pvtmethod=  {  println("I am a private method, only inline or companion can access me");  }    private val pvtval=1;  val FDMaturity = 5+pvtval;  }    //Companion object has same name of the class, which can access private variable  object bankclassobj extends bankclassobj  {  val objbankclassobj=new bankclassobj;  override val FDMaturity=7;  println(objbankclassobj.pvtval);  objbankclassobj.pvtmethod;  } |