Scala: Scala is JVM programming Language, multi-paradigm programming language (Integrates object oriented and functional programming smoothly), object-oriented language. Scala runs on top of the java virtual machine. So that java compiler compiles our scala code to the same bytecode that java runs. We need to install scala plugin in IntelliJ IDE.

Created sample file in scala Playground.scala

package playground  
object Playground {  
 def main(args: Array[String]): Unit = {  
 *println*("learn scala")  
 }  
}

extends App is equivalent to creating a main function which makes this application runnable.

object Playground extends App {  
}

**Values Variables and Types:**

val *x*: Int =42  
*println*(*x*)

x=2 //getting error

Declared with the val keyword cannot be reassigned. val are immutable similar to final in java.

* Types of the val is optional.

val x=42  
 println(x)

* var *empName* : String ="lalitha"  
  *println*(*empName*.take(4)) o/p: lali

Initializing variables to default values: The variable can be initialized to default values using “\_”. “\_” can be used only with var and not val.

**Types in Scala:**

Int,String,Boolean,Char,Short,Long,Float,Double

* Declared with the var keyword is mutable.

var y : Int=32   
y=33  
println(y)

* Compiler automatically infers types when omitted
* Everything in Scala is an expression but not instruction.

**If Expression:**

var *aCondition*=true  
var *aConditionValue* = if(*aCondition*) 5 else 3  
*println*(*aConditionValue*)

o/p:5

**Unit:**  Unit is a special type in scala, which is equivalent to void in other languages.

var *x*=3  
var *aValue* = (*x*=3)  
*println*(*aValue*);

o/p: ()

Side effects: println(),whiles,reassigning (these return unit)

**Code Blocks:**

The value of the block is the value of its last expression.

val codeBlock = {  
 val y=2  
 val z=y+1  
 if(z>2) "hello" else "goodbye"  
 if(z==3) 42 else 5  
}  
println(codeBlock)

o/p: 42

Difference between “hello world” and println(“hello world”)

“hello world” is literal and the type is string but println(“hello world”) type is unit.

val *codeBlock* = {  
 val y=2  
 val z=y+1  
 if(z>2) "hello" else "goodbye"  
 if(z==3) 42 else 5  
 10  
}  
*println*(*codeBlock*)

o/p: 10

**Functions:** Every function is an object.

* def aFunction(a:String, b:Int) :String={  
   a+" "+b  
  }  
  *println*(*aFunction*("hello",3)) o/p: hello 3
* def parameterLessFunction : Int =42  
  *println*(*parameterLessFunction*) o/p:42

Concatenating the string n times:

(when we will need loops we will use recursion)

def repeatedFunction(aString :String, n:Int) :String={  
 If(n==1) aString  
 else aString+repeatedFunction(aString,n-1)  
 }  
 println(repeatedFunction("hello",3))

if we delete the return type of the recursive function compiler will complain.

def biggerFunction(n: Int): Int={  
 def smallerFunction(a:Int, b:Int) : Int=a+b  
 smallerFunction(n,n-1)  
 }  
 println(biggerFunction(3))

**Stack and Tail Recursion:**

Factorial of number:

def factorial(n:Int):Int={

if(n<=1) 1

else n\*factorial(n-1)

}

println(factorial(5)) o/p:120

println(factorial(5000)) o/p: stack overflow error

@tailrec

def anotherFactorial(n:Int,accumulator:BigInt) :BigInt={  
 if(n<=1) accumulator  
 else *anotherFactorial*(n-1,n\*accumulator)  
}  
*println*(*anotherFactorial*(5000,1))

Fibonacci of a number:

def fibonacci(n:Int): Int={  
 def findFib(i:Int,last:Int,nextToLast :Int):Int={  
 if(i>=n) last  
 else findFib(i+1,last+nextToLast,last)  
 }  
 if(n<=2) 1  
 else findFib(2,1,1)  
}  
*println*(*fibonacci*(8))

**Lazy Values:** initializatiom of these variables that is deferred until they are first accessed

lazy val empMessage =” hi”

**Variable Scopes:**

Scala provides three variable scopes fields,method parameters, local variables.

**Input:** readLine(),readInt(),readFloat()……

**String Interpolators:**

S-interpolators

val *name*="david"  
val *age*=25  
val *greeting*= s"hello my name is **$***name* and I am **$**{*age*+1} years old"  
*println*(*greeting*)

F-interpolators

val *speed*=1.2f  
val *myth*=f"**$***name* can eat **$***speed*%2.2f burgers per min"  
*println*(*myth*)

raw-interpolators

*println*(raw"This is a \n newline")  
val *escaped* = "This is a \n newline"  
*println*(raw"**$***escaped*")

**Call-by-Name and Call-by-Value**

def calledByValue(x: Long) :Unit={  
 *println*("by value: "+x)  
 *println*("by value: "+x)  
}  
  
def calledByName(x: =>Long) :Unit={  
 *println*("by name: " + x)  
 *println*("by name: " + x)  
}  
*calledByValue*(System.*nanoTime*())  
*calledByName*(System.*nanoTime*())

o/p: by value: 90754975571300

by value: 90754975571300

by name: 90755068138400

by name: 90755069418500

In call by name , by name parameter delays the evaluation of the expression passed here until it’s used.

def infinite(): Int =1+*infinite*()  
def printFirst( x: Int, y: =>Int) =*println*(x)  
//printFirst(infinite(),34) stack-overflow error  
*printFirst*(34,*infinite*())

In call by value, value is computed before call. Same value used everywhere.

In call by name expression is passed literally, expression is evaluated at every use within the definition.

**Default and Named arguments:**

def trFact(n:Int, acc:Int=1) :Int={  
 if(n<=1) acc  
 else *trFact*(n-1,n\*acc)  
}  
*println*(*trFact*(4))  
  
def savePicture(format:String="jpg",width:Int=1920,height:Int=1000):Unit=*println*("saving picture")  
*savePicture*(800) //error  
*savePicture*(height=600) =>Named Argument

**Object Oriented Programming in Scala:**

object OOBasics extends App {  
 val *person*=new Person  
 *println*(*person*)  
}  
class Person

Pass Parameters to the Class:

Class parameters are not the fields.

object OOBasics extends App{  
 val *person*=new Person("john",26)  
 *println*(*person*.*age*)  
}  
class Person(name:String,val *age*:Int) //constructor

To convert parameters to fields is to add the keyword val or var to the class parameters.

object OOBasics extends App{  
 val *person*=new Person("john",26)  
 *println*(*person*.*x*)  
}  
class Person(name:String,val *age*:Int) //constructor  
{  
 val *x*=2  
 *println*(1+3)  
}

o/p: 4

2

Every instantiation of the class person this whole block of code will be evaluated, every single expression and side effects will be evaluated.

object OOBasics extends App{  
 val *person*=new Person("john",26)  
 *println*(*person*.*x*)  
 *person*.greet("lalitha")  
}  
class Person(name:String,val *age*:Int) //constructor  
{  
 val *x*=2  
 *println*(1+3)  
  
 //method  
 def greet(name:String): Unit=*println*(s"**$**{this.name} says Hi, **$**name")  
}

o/p: 4

2

john says Hi, Lalitha

Multiple Constructors:

object OOBasics extends App {  
 val *person* = new Person("john", 26)  
 val *person1*= new Person("ram")  
 val *person2*= new Person()  
 *println*(*person*.*age*)  
 *println*(*person1*.*name*)  
 *println*(*person2*.*name*)  
}  
class Person(val *name*:String,val *age*:Int) {  
 def this(name:String)=this(name,0)  
 def this()=this("")  
}

or we can have the default parameters

object OOBasics extends App {  
 val *person* = new Person("john", 26)  
 val *person1*= new Person("ram")  
 val *person2*= new Person()  
 *println*(*person*.*age*)  
 *println*(*person1*.*name*)  
 *println*(*person2*.*name*)  
}  
class Person(val *name*:String="",val *age*:Int=0) {  
   
}

**Syntactic Sugar (Method annotations)**

object MethodNotations extends App{  
 class Person(val *name*:String, favoriteMovie :String)  
 {  
 def likes(movie:String): Boolean=movie==favoriteMovie  
 def +(person: Person): String =s"**$**{this.*name*}"+" "+s"**$**{person.*name*}"  
 }  
 val *mary*=new Person("Mary","Inception")  
 *println*(*mary*.likes("Inception"))  
 *println*(*mary* likes "Inception") //infix notation or operator notation  
 val *person*=new Person("john","Inception")  
 *println*(*mary* + *person*)  
}

prefix notation:

val x=-1

val y=1.unary\_-

unary\_ prefix only works with - + ~ !

class Person(val *name*:String, favoriteMovie :String)  
{  
 def unary\_! : String =s"hi! **$***name* This is prefix notation"  
}  
val *mary*=new Person("Mary","Inception")  
*println*(!*mary*)

postfix notation:

import scala.language.*postfixOps*object MethodNotations extends App{  
 class Person(val *name*:String, favoriteMovie :String)  
 {  
 def postFix : String = "This is postfix notation"  
 }  
 val *mary*=new Person("Mary","Inception")  
 *println*(*mary* postFix)

apply method:

class Person(val *name*:String, favoriteMovie :String)  
{  
 def apply() : String ="This is apply method"  
}  
val *mary*=new Person("Mary","Inception")  
*println*(*mary*())

**Scala Objects:**

Scala doesn’t have class -level functionality (“static”)

Objects can be defined in similar way that classes can with the exception that objects do not receive parameters. Scala object is the singleton instance.

object Person {  
 val *name*="john"  
}  
val *person1*= Person  
val *person2*=Person  
*println*(*person1*==*person2*)

o/p: true

**Inheritance:**

object InheritanceAndTraits extends App{  
 class Animal {  
 protected def eat= *println*("animal---")  
 }  
 class Cat extends Animal{  
 def crunch={  
 eat  
 *println*("cat child class for animal")  
 }  
 }  
 val *cat* = new Cat  
 *cat*.crunch

for constructors:

class Human(name:String,age:Int)  
class Adult(name:String,age:Int,idCard:String) extends Human(name,age)

**overriding:**

In java we can’t override the variables but we can override the variables along with the methods in scala.

class Animal {  
 val *creatureType*="Animal"  
 protected def eat= *println*("animal---")  
}  
class Dog extends Animal{  
 override val *creatureType*="Dog"  
 override def eat=*println*("crunch---")  
}  
val *d*=new Dog  
*println*(*d*.*creatureType*)  
*d*.eat

(or)

class Animal {  
 val *creatureType*="Animal"  
 protected def eat= *println*("animal---")  
}  
class Dog(override val *creatureType*: String) extends Animal{  
 override def eat=*println*("crunch---")  
}  
val *d*=new Dog("Dog")  
*println*(*d*.*creatureType*)  
*d*.eat

prevent the overriding

1. use final on member.
2. Use final on the entire class
3. Seal the class(sealed)- extend classes in this file, prevent extension in other files.

**Abstract Class:**

abstract class Animal {  
 val creatureType: String  
 def eat: Unit  
}  
class Dog extends Animal {  
 override val creatureType : String="Dog"  
 def eat : Unit= println("override keyword is optional for the implementation of abstract class")  
}  
val dog = new Dog  
dog.eat

**Trait:**

trait Carnivore {  
 def eat(animal:Animal) : Unit  
}  
class Crocodile extends Animal with Carnivore{  
 override val *creatureType*="croc"  
 def eat : Unit=*println*("nomnom")  
 def eat(animal:Animal) : Unit=*println*(s"**$**{animal.*creatureType*}")  
}  
val *croc* = new Crocodile  
val *dog* = new Dog  
*croc* eat *dog*

we can implement multiple inheritance in scala using the traits.

**Generics:**

class MyList[A]  
{  
  
}  
val *listOfIntegers* = new MyList[Int]  
val *listOFStrings* = new MyList[String]

covariance:

class Animal {  
 def print : Unit = *println*("In Animal Parent class")  
 }  
 class Dog extends Animal  
 {  
 override def print : Unit = *println*("In Dog Child class")  
 }  
 class Cat extends Animal{  
 override def print : Unit = *println*("In Cat Child class")  
 }  
  
class Animals[+A]  
{  
 def animal :Animal= new Dog  
}  
val *dog* = new Dog  
val *Animals*: Animals[Dog] = new Animals[Dog]  
val *animalAnimals* : Animals[Animal] = *Animals*

contravariance:

class Animals[-A]  
{  
 def animal :Animal= new Dog  
}  
val *dog* = new Dog  
val *Animals*: Animals[Animal] = new Animals[Animal]  
val *animalAnimals* : Animals[Dog] = *Animals*

**Bounded Types:**

class Animals[A <: Animal](animal: A) => Upper Bound  
{  
 def printing: Unit = animal.print  
}  
class Car  
val *animals*= new Animals(new Dog) // in this line we can’t pass car object  
*animals*.printing

**Anonymous Classes:**

abstract class Animal {  
 def eat: Unit= *println*("Animal is eating")  
 }  
 val *anonymousAnimal* = new Animal{  
 override def eat : Unit = *println*("eating and creating anonymous classes")  
 }  
*anonymousAnimal*.eat  
class Person(val *name*: String)  
{  
 def print: Unit= *println*(s"**$**{*name*}")  
}  
val *person1* = new Person("jim"){  
 override def print: Unit = *println*(s"hiii **$**{*name*}")  
}  
*person1*.print

o/p:

eating and creating anonymous classes

hiii jim

**Companion Object:**

In Scala, a companion object is an object that has the same name as a class and is defined in the same source file. A companion object is used to define methods and values that are shared across all instances of the class, similar to static methods and fields in Java. However, Scala does not have static members; instead, it uses companion objects to achieve similar functionality.

class Person(val name: String, val age: Int) {

def greet(): Unit = println(s"Hello, my name is $name.")

}

object Person {

def apply(name: String, age: Int): Person = new Person(name, age)

def printInfo(): Unit = println("This is a Person companion object.")

}

**Case classes:**

case class Person(*name*:String,*age*:Int)  
  
//1.class parameters are fields  
val *jim*=new Person("jim",34)  
*println*(*jim*.*name*)  
  
//2.toString() default implementation  
*println*(*jim*)  
  
//3. equals and hashcode implementation  
val *jim2*= new Person("jim",34)  
*println*(*jim*==*jim2*)  
  
//4.having copy method  
val *jim3* = *jim*.copy(age=45)  
*println*(*jim3*)  
  
//5. having the companion objects  
val *thePerson*= Person  
val *mary* = Person("Mary",25)  
*println*(*mary*)  
  
//6. These are serializable and can be used in pattern object  
  
case object United{  
 def name: String="Case objects"  
}

**Enums:**

Enums are the data type that can define like a class.

enum Permissions {  
 case *READ*, *WRITE*, *EXECUTE*, *NONE* //add fields/methods  
 def openDocument():Unit=  
 if(this==*READ*) *println*("opening document")  
 else *println*("reading not allowed")  
}  
val *somePermission* = Permissions.*READ  
somePermission*.openDocument()

Enums can also take the constructor arguments

enum PermissionsWithBits(val *bits*: Int)  
{  
 case *READ* extends PermissionsWithBits(4)  
 case *WRITE* extends PermissionsWithBits(2)  
 case *EXECUTE* extends PermissionsWithBits(1)  
 case *NONE* extends PermissionsWithBits(0)  
}  
val *permissionBits* = PermissionsWithBits.*READ*.*bits  
println*(*permissionBits*)

we also can have the comapanion object for the enums.

Standard API in enums:

* val *somePermission* = Permissions.*READ*

val *index*= *somePermission*.ordinal

*println*(*index*) o/p: 0

* val *allPermissions*= PermissionsWithBits.*values* //array of all possible values of the enum  
  *println*(*allPermissions*.mkString(","))
* val *readPermission*= Permissions.*valueOf*("READ")  
  *println*(*readPermission*) o/p: READ

**Exceptions:**

val *x*: String= null  
// val wierdValue = throw new NullPointerException  
 def getInt(withException : Boolean) : Int ={  
 if(withException) throw new RuntimeException  
 else 42  
 }  
 val *potentialFail* = try{  
 *getInt*(true)  
 }  
 catch{  
 case e: RuntimeException => 43  
 }  
 finally{  
 *println*("finally")  
 }  
 *println*(*potentialFail*)

How to define own Exception

class MyException extends Exception  
val *exception* = new MyException

OutOfMemoryError:

val array = Array.ofDim(Int.MaxValue)

StackOverFlowError

def infinite : Int = 1+infinite

val noLimit = infinte

**Packaging & Imports:**

Package members are accessible by their simple name, if we are not in the proper package, we need to import the package.

Package Object:

When creating the package object it takes the name of package. We can define the methods or constants and use them by their simple name throughout the entire package.

Suppose we have to import two classnames classname1, classname2 from the package playground.

import playground.{classname1, classname2}

or import playground.\_

import java.util.Date

import java.sql.Date

* val date = new Date
* val sqlDate = new java.sql.Date(2018,5,4)

another way is aliasing

import java.util.Date

import java.sql.{Date=> SqlDate}

* val date = new Date
* val sqldate = new SqlDate(2018,5,4)

**Functional Programming in Scala:**

The purpose of the functional programming is to use and work with functions as first class elements.

All scala functions are the objects.

val *doubler* = new MyFunction[Int,Int]{  
 override def apply(element: Int): Int = element \*2  
}  
*println*(*doubler*(2))

trait MyFunction[A,B] {  
 def apply(element: A): B  
}

//function types = Function1[A,B]  
val *stringToIntConverter* = new Function1[String,Int]{  
 override def apply(string: String) :Int = string.toInt  
}  
*println*(*stringToIntConverter*("2"))

=>

val *adder* : (Int,Int) => Int = new Function2[Int,Int,Int]{  
 override def apply(a:Int, b:Int) :Int = a+b  
}  
*println*(*adder*(2,4))

Exercise:

Define a function which takes an Int and returns another function which takes an int and returns an int.

val *superAdder* = new Function1[Int, Function1[Int,Int]]{  
 override def apply(x:Int) : Function1[Int,Int]= new Function1[Int,Int]{  
 override def apply(y:Int): Int= x+y  
 }  
}  
*println*(*superAdder*(4)(5)) //curried functions

**Anonymous Functions:(lambda)**

val doubler :Int=>Int = x=> x\*2

or

val doubler = (x:Int) = x\*2

If we want to call lambda we have to call with the paranthesis for the no parameters lambda.

**Moar Syntactic Sugar:**

val niceIncrementer : Int => Int = \_+1 //equivalent to x=>x+1

val niceAdder : (Int,Int) => Int = \_+\_ //equivalent to (a,b)=> a+b

val superadd = (x:Int) => (y : Int) => x+y

**Higher Order Functions and Curried Functions:**

Higher order functions that either takes function as parameters or returns functions as a result is called higher order function

//function that applies a function n times over a value x  
 // nTimes(f,3,x) = f(f(f(x)))  
 def nTimes(f : Int => Int, n:Int , x:Int) :Int =  
 {  
 if(n<=0) x  
 else *nTimes*(f,n-1,f(x))  
 }  
val *plusOne* = (x: Int) => x+1  
*println*(*nTimes*(*plusOne*, 10, 1))  
def nTimesBetter(f:Int => Int, n: Int) :(Int => Int) =  
 {  
 if(n<=0) (x:Int)=> x  
 else (x:Int) => *nTimesBetter*(f,n-1)(f(x))  
 }  
val *plus* = *nTimesBetter*(*plusOne*,10)  
*println*(*plus*(1))

o/p: 11

11

Curried Function:

A curried function in Scala is a function that takes multiple argument lists, rather than a single list of all arguments. Curried functions enable partial application, which means you can call a function with some of its arguments and get back another function that takes the remaining arguments.

def curriedFunction(a: Int)(b: Int): Int = a + b  
*println*(*curriedFunction*(2)(3)) // Output: 5  
val *addTwo* = *curriedFunction*(2) // Partial application  
*println*(*addTwo*(3)) // Output: 5

**Map, flatMap, filter and for:**

val *list* = List(1,2,3)  
*println*(*list*.head)  
*println*(*list*.tail)  
*println*(*list*.map(\_+1))  
*println*(*list*.map(\_+" is a number"))  
*println*(*list*.filter(\_%2==0))  
val *toPair* = (x:Int) => List(x,x+1)  
*println*(*list*.flatMap(*toPair*))

o/p:

1

List(2, 3)

List(2, 3, 4)

List(1 is a number, 2 is a number, 3 is a number)

List(2)

List(1, 2, 2, 3, 3, 4)

* Print all the combinations of the two list

val *numbers* = List(1,2,3,4)  
 val *chars* = List('a','b','c','d')  
 //List("a1","a2"-----,"d4")  
  
 val *combinations* = *numbers*.flatMap(n => *chars*.map(c => ""+c+n))  
 *println*(*combinations*)

val *colors* = List("black","white")  
val *combination* = *numbers*.flatMap(n=> *chars*.flatMap(c => *colors*.map(color => ""+c+n+"-"+color)))  
*println*(*combination*)  
  
*list*.foreach(*println*) //printing the list numbers in the different lines

val *forCombinations* = for{  
 n <- *numbers* c <- *chars* color <- *colors*} yield ""+c+n+"-"+color  
*println*(*forCombinations*)

for{

n <- numbers

} println(n)

Syntax Overload:

list. Map { x =>

x\*2

}

**Scala Collections:**

Scala offers both mutable and immutable collections

Immutable collections are found in scala.collections.immutable package

Traversable

HashMap

HashSet

Iterable

Map

Set

Seq

SortedMap

SortedSet

IndexedSeq

LinearSeq

Queue

Stack

Stream

List

Range

String

Vector

Mutable collections are found in scala.collections.mutable package

In set HashSet and LinkedHashSet, In map HashMap, MultiMap.

In seq IndexedSeq => StringBuilder and ArrayBuffer

In seq Buffer => ArrayBuffer and ListBuffer

In seq LinearSeq => LinkedList and MutableList

val *aSequence* = Seq(1,4,3,2)  
*println*(*aSequence*)  
*println*(*aSequence*.reverse)  
*println*(*aSequence*(2))  
*println*(*aSequence* ++ Seq(7,5,6))  
*println*(*aSequence*.sorted)  
  
val *aRange* : Seq[Int] = 1 until 10  
//aRange.foreach(println)  
  
val *aList* = List(1,2,3)  
val *prepended* = 42 +: *aList* :+ 89  
*println*(*prepended*)  
  
val *apples5* = List.fill(5)("apple")  
*println*(*apples5*)  
*println*(*apples5*.mkString("-"))  
  
val *numbers* = Array(1,2,3,4)  
val *threeElements* = Array.*ofDim*[String](3)  
*threeElements*.foreach(*println*)  
  
*numbers*(2)=0  
*println*(*numbers*.mkString(" "))  
  
//array can be converted to sequence implicitly  
  
val *capacity*=10  
val *numberList* = (1 to *capacity*).toList  
*println*(*numberList*)

o/p: List(1, 4, 3, 2)

List(2, 3, 4, 1)

3

List(1, 4, 3, 2, 7, 5, 6)

List(1, 2, 3, 4)

List(42, 1, 2, 3, 89)

List(apple, apple, apple, apple, apple)

apple-apple-apple-apple-apple

null

null

null

1 2 0 4

List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

**Tuples and Maps:**

val *aTuple* = (2,"hello-world")  
*println*(*aTuple*.*\_1*)  
*println*(*aTuple*.copy(\_2="good bye"))  
*println*(*aTuple*.swap)  
val *aMap* : Map[String,Int]= Map()  
val *phonebook* = Map(("jim",555),"John" -> 666).withDefaultValue(-1)  
*println*(*phonebook*)  
*println*(*phonebook*.contains("Jim"))  
*println*(*phonebook*("abc")) //exception if we don't use withDefaultValue  
*println*(*phonebook*.map(pair => pair.*\_1*.toLowerCase -> pair.*\_2*))  
*println*(*phonebook*.view.filterKeys(x => x.startsWith("J")).toMap)  
*println*(*phonebook*.view.mapValues(number => number\*10).toMap)  
  
val *names*= List("Bob","Bobby","Jim","Angela","John")  
*println*(*names*.groupBy(name => name.charAt(0)))

o/p:

2

(2,good bye)

(hello-world,2)

Map(jim -> 555, John -> 666)

false

-1

Map(jim -> 555, john -> 666)

Map(John -> 666)

Map(jim -> 5550, John -> 6660)

HashMap(J -> List(Jim, John), A -> List(Angela), B -> List(Bob, Bobby))

**Options:**

Option is a wrapper for a value that might be present or not

val *myFirstOption* : Option[Int] = Some(4)  
val *noOption* : Option[Int] = None  
*println*(*myFirstOption*)  
def unSafeMethod() : String = null  
val *result* = Option(*unSafeMethod*())  
*println*(*result*)  
  
//chained methods  
def backUpMethod() : String ="A valid result"  
val *chainedResult* = Option(*unSafeMethod*()).orElse(Option(*backUpMethod*()))  
*println*(*chainedResult*)  
val *chainresult* = *myFirstOption* orElse *noOption  
println*(*chainresult*)  
*println*(*myFirstOption*.isEmpty)

o/p:

Some(4)

None

Some(A valid result)

Some(4)

False

**Handling Failure:**

val aSuccess = Success(3)

val aFailure = Failure(new RunTimeException(“Super Failure”))

println(aSuccess) => Success(3)

println(aFailure) => Failure(java.lang.RuntimeException : Super Failure)

def unSafeMethod() : String = throw new RuntimeException(“No String”)

val potentialFailure = Try(unsafeMethod())

println(potentialFailure)

Here program didn’t crash although we called unsafemethod because try to catch exception wrap up in the Failure.

Println(potentialFailure.isSuccess) => false

def backupMethod () : String =”A Valid Result”

val fallbackTry = Try(unSafeMethod()).orElse(Try(backupMethod()))

println(fallbackTry) => Success(A Valid Result)

def betterUnSafeMethod() : Try[String] = Failure(new RuntimeException)

def betterBackupMethod() : Try[String] = Success(“A valid result”)

val betterFallBack = betterUnSafeMethod() orElse betterBackupMethod()

println(betterFallBack) => Success(A valid result)

**Pattern Matching:**

val random = new Random()

val x = random.nextInt(10)

val desc = x match {

case 1 => “The One”

case 2 => “ double or nothing”

case 3 => “third time”

case \_ => “some thing else”

}

Decompose the values:

case class Person(name:String,age:Int)

val bob = Person(“Bob”,20)

val greeting = bob match {

case Person(n,a) => s”Hi , my name is $n and I am $a years old”

case \_ => “Some thing”

}

cases are matched in order.

Pattern matching works well with the case classes.