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:**