Scala

* In scala class is object and extends App which provide main class and no ; req and println used to print value.
* Scala is a pure object-oriented language in the sense that every value is an object. Types and behavior of objects are described by classes and traits
* Scala is also a functional language in the sense that every function is a value and every value is an object so ultimately every function is an object.

**Scala different from java:**

* All types are objects
* Type inference
* Nested Functions
* Functions are objects
* Domain specific language (DSL) support
* Traits
* Closures
* Concurrency support inspired by Erlang

Scala key terms:

* **Object,class,method ,field,closure and traits**
* Closure: is a function, whose return value depends on the value of one or more variables declared outside this function.
* Traits : A trait encapsulates method and field definitions, which can then be reused by mixing them into classes. Traits are used to define object types by specifying the signature of the supported methods.

Basics Syntax:

* You can import more than one class or object from a single package, for example, TreeMap and TreeSet from the scala.collection.immutable package –

import scala.collection.immutable.{TreeMap, TreeSet}

* All the data types listed above are objects. There are no primitive types like in Java. This means that you can call methods on an Int, Long, etc.

Variable:

* Var: var myVar : String = "Foo" (muttable)
* Val: val myVal : String = "Foo" (immutable)

**Variable Syntax:**

val or val VariableName : DataType = [Initial Value]

* By default : var myVar :Int;

val myVal :String;

* Mutiple assignment: val (myVar1, myVar2) = Pair(40, "Foo")

**Varaible scope:**

* Field: Fields are variables that belong to an object can be var or val
* Method Parameter: Method parameters are variables, which are used to pass the value inside a method, when the method is called. Can be always val.
* Local: Local variables are variables declared inside a method. Can be val or var.

**Class variables are called, fields of the class and methods are called class methods.**

**The class name works as a class constructor which can take a number of parameters. The above code defines two constructor arguments, xc and yc; they are both visible in the whole body of the class.**

**Syntax:** class Point(xc: Int, yc: Int) {

var x: Int = xc

var y: Int = yc

def move(dx: Int, dy: Int) {

x = x + dx

y = y + dy

println ("Point x location : " + x);

println ("Point y location : " + y);

}

}

object Demo {

def main(args: Array[String]) {

val pt = new Point(10, 20);

// Move to a new location

pt.move(10, 10);

}

}

**Extending a class:**

* Same as java but there are two restrictions: method overriding requires the **override** keyword, and only the **primary** constructor can pass parameters to the base constructor
* Extending a class and inheriting all the features of a parent class is called inheritance but Scala allows the inheritance from just one class only.

**Implicit class:**

**Singleton class:**

* Scala is more object-oriented than Java because in Scala, we cannot have static members
* You create singleton using the keyword **object** instead of class keyword

**Access Modifier**

* By default, public. Same as java
* Access modifiers in Scala can be augmented with qualifiers. A modifier of the form private[X] or protected[X] means that access is private or protected "up to" X, where X designates some enclosing package, class or singleton object.

package society {

package professional {

class Executive {

private[professional] var workDetails = null

private[society] var friends = null

private[this] var secrets = null

def help(another : Executive) {

println(another.workDetails)

println(another.secrets) //ERROR

}

}

}

}

**Function and method:**

* Scala method is a part of a class which has a name, a signature, optionally some annotations, and some bytecode whereas a function in Scala is a complete object which can be assigned to a variable. In other words, a function, which is defined as a member of some object, is called a method.
* Scala permits nested function definitions, that is, function definitions inside other function definitions.
* Scala function's name can have characters like +, ++, ~, &,-, --, \, /, :, etc.
* A function, that does not return anything can return a **Unit** that is equivalent to **void** in Java and indicates that function does not return anything. The functions which do not return anything in Scala, they are called procedures.

**Declaration:** def functionName (list of parameters) : return type = {

function body

return [expr] }

**Different Scala Function :**

* **Function call by name:** if we need to write a function that accepts as a parameter an expression that we don't want evaluated until it's called within our function.

object Demo {

def main(args: Array[String]) {

delayed(time());

}

def time() = {

println("Getting time in nano seconds")

System.nanoTime

}

def delayed( t: => Long ) = {

println("In delayed method")

println("Param: " + t)

}

}

* **Function with variable argument:**  The last parameter to a function may be repeated. This allows clients to pass variable length argument lists to the function. String\* is array of String

object Demo {

def main(args: Array[String]) {

printStrings("Hello", "Scala", "Python");

}

def printStrings( args:String\* ) = {

var i : Int = 0;

for( arg <- args ){

println("Arg value[" + i + "] = " + arg );

i = i + 1;

}

}

}

* **Default Parameter value:** you specify default values for function parameters. The argument for such a parameter can optionally be omitted from a function call, in which case the corresponding argument will be filled in with the default. If you specify one of the parameters, then first argument will be passed using that parameter and second will be taken from default value but not vice versa.

object Demo {

def main(args: Array[String]) {

println( "Returned Value : " + addInt() );

}

def addInt( a:Int = 5, b:Int = 7 ) : Int = {

var sum:Int = 0

sum = a + b

return sum

}

}

**Nested Function :** Scala allows you to define functions inside a function and functions defined inside other functions are called **local functions.**

object Demo {

def main(args: Array[String]) {

println( factorial(0) )

println( factorial(1) )

println( factorial(2) )

println( factorial(3) )

}

def factorial(i: Int): Int = {

def fact(i: Int, accumulator: Int): Int = {

if (i <= 1)

accumulator

else

fact(i - 1, i \* accumulator)

}

fact(i, 1)

}

}

**Partially applied function:** If you pass all the expected arguments, you have fully applied it. If you send only a few arguments, then you get back a partially applied function. By passing \_ in argument.

import java.util.Date

object Demo {

def main(args: Array[String]) {

val date = new Date

val logWithDateBound = log(date, \_ : String)

logWithDateBound("message1" )

Thread.sleep(1000)

logWithDateBound("message2" )

Thread.sleep(1000)

logWithDateBound("message3" )

}

def log(date: Date, message: String) = {

println(date + "----" + message)

}

}

**Function with Named argument:**  the arguments in the call are matched one by one in the order of the parameters of the called function. Named arguments allow you to pass arguments to a function in a different order.

object Demo {

def main(args: Array[String]) {

printInt(b = 5, a = 7);

}

def printInt( a:Int, b:Int ) = {

println("Value of a : " + a );

println("Value of b : " + b );

}

}

**Recursion Function:** Recursion means a function can call itself repeatedly.

**High order function :** Scala allows the definition of **higher-order functions**. These are functions that take other functions as parameters, or whose result is a function. ????

object Demo {

def main(args: Array[String]) {

println( apply( layout, 10) )

}

def apply(f: Int => String, v: Int) = f(v)

def layout[A](x: A) = "[" + x.toString() + "]" }}

**Anonymous Function:** Anonymous functions in source code are called **function literals** and at run time, function literals are instantiated into objects called **function values**.

var inc = (x:Int) => x+1

**Currying Function:** Currying transforms a function that takes multiple parameters into a chain of functions, each taking a single parameter.

def strcat(s1: String)(s2: String) = s1 + s2

object Demo {

def main(args: Array[String]) {

val str1:String = "Hello, "

val str2:String = "Scala!"

println( "str1 + str2 = " + strcat(str1)(str2) )

}

def strcat(s1: String)(s2: String) = {

s1 + s2

}

}

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

object Demo {

def main(args: Array[String]) {

println( "multiplier(1) value = " + multiplier(1) )

println( "multiplier(2) value = " + multiplier(2) )

}

var factor = 3

val multiplier = (i:Int) => i \* factor

}

**String : Same as java .**

* **String interpolation:** the new way to create Strings .Three ways to do that.

1. **with s string interpolator**

object Demo {

def main(args: Array[String]) {

val name = "James"

println(s"Hello, $name")

println(s"1 + 1 = ${1 + 1}")

}

}

1. **The f interpolator:** like c printf statement

val height = 1.9d

val name = "James"

println(f"$name%s is $height%2.2f meters tall") //James is 1.90 meters tall

‘ f ’ interpolator makes use of the String format utilities (format specifiers) available in Java. By default, means, there is no % character after variable reference. It will assume as %s (String).

1. **Raw interpolator :** The ‘raw’ interpolator is similar to ‘s’ interpolator except that it performs no escaping of literals within a string.

object Demo {

def main(args: Array[String]) {

println(raw"Result = \n a \n b")

} }

Output :Result = \n a \n b

**Array:**

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

or

var z = new Array[String](3)

object Demo {

def main(args: Array[String]) {

var myList = Array(1.9, 2.9, 3.4, 3.5)

// Print all the array elements

for ( x <- myList ) {

println( x )

}

// Summing all elements

var total = 0.0;

for ( i <- 0 to (myList.length - 1)) {

total += myList(i);

}

println("Total is " + total);

// Finding the largest element

var max = myList(0);

for ( i <- 1 to (myList.length - 1) ) {

if (myList(i) > max) max = myList(i);

}

println("Max is " + max);}}

**Expression:**

Unit is void in other languages.

Basic expression:

val x=1+2

val even =x%2==0

val odd=!even

val x= if(x>5)5 else 3

scala everything is expression(return something) not instruction (Do something)

val x {

multiple code lines

}

**Function:**

**Syntax:** def functionName(varname:varType , varname:varType ) :returntype ={}

Parameter less function can be called as without ();

Syntax: println(functionName)

Instead of loops use recursive function in scala