Scala

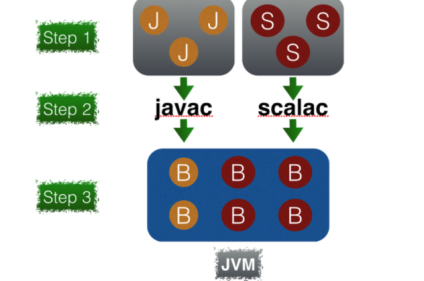
Scala is both Functional And Object Oriented Language that runs on the JVM .

It was developed By **Martin Odersky.**

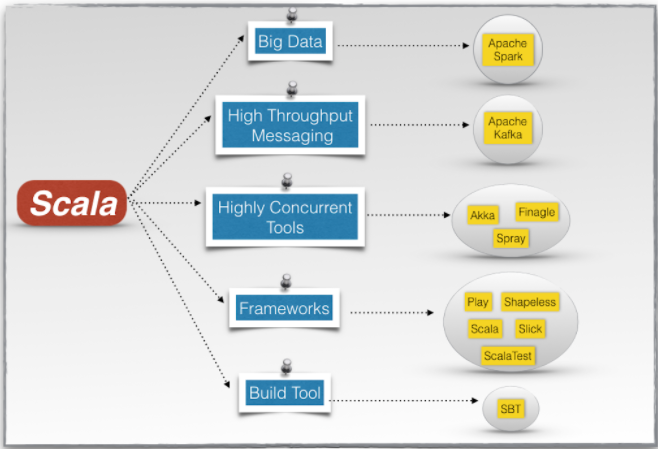
JDK – Contains set of libraries and utilities for developing the Java Application.It also comes with the JRE Java RunTime Environment. you will also hear the term Java Virtual Machine or JVM used to refer to the JRE.

 Scala, there is **NO** need to terminate the statement with a **semi-colon** .Scala can extend the Trait Called APP to run the code. Where the Trait “APP” have the Main Method. Consider Trait as an equivalent to the Interface in Java.

AS per the Below Diagram both Java And Scala Code will be converted into the ByteCode and will be running on top of JVM.



Scala EcoSystem



Setting Up the Development Environment

Installing the Scala Plugin In INTELLIJ

<http://allaboutscala.com/tutorials/chapter-1-getting-familiar-intellij-ide/scala-environment-setup-install-scala-plugin-intellij/>

2.1 Variables in Scala

2.2String interpolation

2.3

3.Functions

* Create a Simple Function named **favouritedonut** which has the return type as String
* Create a Function called leastFavouriteDonut without Paranthesis
* Create a Function “PrintDonutSalesReport” With No Return Type
* Create A Function “CalculateDonutCost” which has the donutName Quantity as Input and it return the total cost
* Modify the Above Function such that it accepts another parameter called CouponCode and it has some default value as “Nocode”.
* Modify the Function “CalculateDonutCost” with couponCode as Option Parameter
* Assign Default Value to the CouponCode Option Parameter
* Create a Function “CalculateCouponCode” which Returns Options[Strings] , if the return value is null display ur custiom code . call the CalculateCouponCode Function Using the Pattern Matching.
* Create A Function Called “ApplyDiscount” and make the Function as a Generic Parameter of Type T
* Lets modify the Above Function and we are returning a sequnce and make the sequence as general one
* Create A Function Called “PrintReport” and it accepts n number of Strings Array of Strings , Seq of Strings and Display it.
* Create A Function Named “Summation” and it should accepts n numbers and it should give back their sum to us.
* Create a Function called “totalCost” as Currying Function and Find the TotalCost for the Donuts

Typed Function :-

We have a Function 1 which calculates the Discount and it accepts String,

We have Another Function which calculates the Discount and it accepts Double.

Instead oF Two Functions we can use a Simple and Common Generic Functions To Handle the above two Scenarios,

we will create a **typed** function which will specify a **genericparameter** of type T as follows:

def applyDiscount[**T**](discount: **T**) {

* Sure you can rely on Scala's type inference to infer the type of your function parameter.
* But in general it's a good practice to explicitly specify your parameter types.

We can also add the retuen Type to the applyDicount Function

Variable Arguments Functions

def printReport(names: String\*) {

println(s"""Donut Report = ${names.mkString(", ")}""")

}

This Function Will Accepts a n number of Strings and Displays It .

We can also pass a list to this Function

Currying if we want to Calculate the students Grade based on the exam mark assignments quizzes then the variable argument Function is not suited Well

We can use Partial Functions

def add(a:Int,b:Int) = a+b;

def add\_curry(a:Int)(b:Int)=a+b;

val plus5 = add\_curry(5)

plus5(3) //8

we can also use the currly Bracessis

add\_curry{5}{3} // 8

add\_curry{

println(“hi”)

5}

{3} //8

Def examResults(marks:Int\*,assign:Int\*,quizzes:Int\*)={} while calling we don’t know whether how many marks assignments quizzes were there

So Better Option is Currying the same method can be re written as

Def examResults(marks:Int\*)(assign:Int\*)(quizzes:Int\*)={}

examResults(32,23,4,3)(23,23,23,2)(34,5,4)

Higher Order Functions

we will learn how to create **Higher Order Function** which is a function that takes another function as its parameter.

def totalCostWithDiscountFunctionParameter(donutType: String)(quantity: Int)(f: Double => Double): Double = {

println(s"Calculating total cost for $quantity $donutType")

val totalCost = 2.50 \* quantity

f(totalCost)

}

Creating the High Order Function With Another Function

println("\nStep 4: How to define and pass a function to a higher order function")

def applyDiscount(totalCost: Double): Double = {

val discount = 2 // assume you fetch discount from database

totalCost - discount

}

Higher Order Functions with Call-by Name

Notes:-

**Scala's typed functions provide greater flexibility with the use of variances which we will see in upcoming tutorial**

**Some Common Methods In Scala**

**Classes In Scala**

**Pattern Matching in scala**

**Working With Strings**

val name = "Alvin"

val fullName = name + " Alexander" // Alvin Alexander

val fullName = s"$name Alexander" // Alvin Alexander

name.length // 5

name.foreach(print) // Alvin

name.take(2) // Al

name.take(2).toLowerCase // al

val a = "foo"

val b = "foo"

val c = "bar"

a == b // true

a == c // false

val speech = """Four score and

|seven years ago""".stripMargin

val speech = """Four score and

#seven years ago""".stripMargin('#')

// use single- and double-quotes in a multiline string

val s = """This is known as a

"multiline" string

or 'heredoc' syntax"""

**String interpolation/substitution**

val name = "Joe"

val age = 42

val weight = 180.5

// prints "Hello, Joe"

println(s"Hello, $name")

// prints "Joe is 42 years old, and weighs 180.5 pounds."

println(f"$name is $age years old, and weighs $weight%.1f pounds.")

// 'raw' interpolator

println(raw"foo\nbar")

%c character

%d decimal (integer) number (base 10)

%e exponential floating-point number

%f floating-point number

%i integer (base 10)

%o octal number (base 8)

%s a string of characters

%u unsigned decimal (integer) number

%x number in hexadecimal (base 16)

%% print a percent sign

\% print a percent sign

val s = "four score"

s.substring(0,3) // "fou"

s.substring(0,4) // "four"

s.substring(1,5) // "our "

s.substring(1,6) // "our s"

s.substring(0, s.length-1) // "four scor"

s.substring(0, s.length) // "four score"

scala> for (c <- "hello") yield c.toUpper

res0: String = HELLO

"hello".getBytes.foreach(println)

val a = Array(1,2,3)

a.mkString // "123"

a.mkString(",") // "1,2,3"

a.mkString(" ") // "1 2 3"

a.mkString("(", ",", ")") // "(1,2,3)"

// the scala way

"hello"(0) // "h"

"hello"(1) // "e"

val s = "hello world"

s.toUpperCase // "HELLO WORLD"

s.toLowerCase // "hello world"

s.capitalize // "Hello world"

// distinct

"hello world".distinct // yields "helo wrd"

// intersect

val a = "Alvin"

val b = "Alexander"

a intersect b // yields "Aln"

b intersect a // yields "Aln"

// diff

val a = "Four score and six years ago"

val b = "Four score and seven years ago"

a diff b // "ix"

b diff a // "vene"

val a = "foo bar baz"

val foo = "foo"

A String can be treated as a collection of Char, and that’s what is shown in many of the following examples. I’ve skipped over a few of the collection-related methods because (a) they’re a little harder to show, and (b) I’m running out of time. Here are the String examples:

foo \* 3 // foofoofoo

a.capitalize // Foo bar baz

a.collect

a.compare

a.compareTo

a.count(\_ == 'a') // 2

a.diff("foo") // " bar baz"

a.distinct // fo barz

a.drop(4) // bar baz

a.dropRight(2) // foo bar b

a.dropWhile(\_ != ' ') // " bar baz"

a.endsWith("baz") // true

a.filter(\_ != 'a') // foo br bz

a.fold //

a.foreach(println(\_)) // prints one character per line

a.foreach(println) // prints one character per line

a.format //

a.getBytes.foreach(println) // prints the byte value of each character, one value per line

a.head // f

a.headOption // Some(f)

a.indexOf('a') // 5

a.intersect // shown above

a.isEmpty // false

a.lastIndexOf('o') // 2

a.length // 11

a.map(\_.toUpper) // FOO BAR BAZ

a.map(\_.byteValue) // Vector(102, 111, 111, 32, 98, 97, 114, 32, 98, 97, 122)

a.min // " "

a.mkString(",") // f,o,o, ,b,a,r, ,b,a,z

a.mkString("->", ",", "<-") // ->f,o,o, ,b,a,r, ,b,a,z<-

a.nonEmpty // true

a.par // a parallel array, ParArray(f, o, o, , b, a, r, , b, a, z)

a.partition(\_ > 'e') // (foorz, " ba ba") // a Tuple2

a.reduce //

a.replace('o', 'x') // fxx bar baz

a.replace("o", "x") // fxx bar baz

a.replaceAll("o", "x") // fxx bar baz

a.replaceFirst("o", "x") // fxo bar baz

a.reverse // zab rab oof

a.size // 11

a.slice(0,5) // foo b

a.slice(2,9) // o bar b

a.sortBy //

a.sortWith(\_ < \_) // " aabbfoorz"

a.sortWith(\_ > \_) // "zroofbbaa "

a.sorted // " aabbfoorz"

a.span //

a.split(" ") // Array(foo, bar, baz)

a.splitAt(3) // (foo," bar baz")

a.substring(4,9) // bar b

a.tail // oo bar baz

a.take(3) // foo

a.takeRight(3) // baz

a.takeWhile(\_ != 'r') // foo ba

a.toArray // Array(f, o, o, , b, a, r, , b, a, z)

a.toBuffer // ArrayBuffer(f, o, o, , b, a, r, , b, a, z)

a.toList // List(f, o, o, , b, a, r, , b, a, z)

a.toSet // Set(f, a, , b, r, o, z)

a.toStream //

a.toLowerCase // foo bar baz

a.toUpperCase // FOO BAR BAZ

a.toVector // Vector(f, o, o, , b, a, r, , b, a, z)

a.trim // "foo bar baz"

a.view //

a.zip(0 to 10) // Vector((f,10), (o,11), (o,12), ( ,13), (b,14), (a,15), (r,16), ( ,17), (b,18), (a,19), (z,20))

a. zipWithIndex // Vector((f,0), (o,1), (o,2), ( ,3),

**Chapter – 6 Collections**

Immutable Collections

List

1. Create a List

* Lisp Style
  + Val list = 1::2::3::4::NIil
* Java Style
  + **Val list = List(1,2,3,4)**
  + We don't need the "new" keyword before the List, and (b) you don't have to declare the type of elements in the List.
* Range Method
  + Val list = List.range(1 to 4)
* Fill Method
  + Val l = List.fill(3)(“esak”)
* Tabulate Method
  + Val l = List.tabulate(5)( n => n\*n)
  + x: List[Int] = List(0, 1, 4, 9, 16)

2. Accessing the List

println(s"Element at index 0 = ${list1(0)}")

3. How to append elements at the end of immutable List using :+

**if we want to append List @ the End use :+**

val list2: List[String] = list1 :+ "Vanilla Donut"

println(s"Append elements at the end using :+ = $list2")

If we Want to Prepend the List we use **+:**

If we Want to Merge two Different Lists into One we use ::**:**

**val c = a ::: b**

**c: List[Int] = List(1, 2, 3, 4, 5, 6)**

If we want to add two Immutable List we use **::**

val list4: List[Any] = list1 :: list2

println(s"Add two lists together using :: = $list4")

4. How to Initialize the Empty Immutable List

Val l = List.empty[Strings]

5. Iterating List With For Each Method

**val x = List(1,2,3)**

x.foreach(println)

**Another Example to Find the Sum of the List**

Var sum = 0

Val abc = List(1,2,3,4,5)

abc.foreach(sum += \_)

**Using For expression**

For( ele <- abc){ Println( “Element of the Lists is “ + ele) }

**Printing the Names if it starts With J For + Guard**

**val names = List("Bob", "Fred", "Joe", "Julia", "Kim")**

**for(name <- names if name.startsWith(‘J’) ) { println }**

**For/Yield Expression**

A common use case is to use a for loop with yield to create a new data structure from an existing data structure

**Val nameWithJ = for( name -< names if name.startWith(‘j’)) yield name.captalize**

**In Traditional Java Languages we Use Counters in scala if we want to do**

**for (i <- 0 until names.length) {**

**println(s"$i is ${names(i)}")**

**}**

**Much More scala way is to use zipWithIndex**

**For((name,count) <- names.zipWithIndex){**

**Println(s”$count is $name”)**

**zipWithIndex Starts From 0 if u want to Create Ur Own Counter Then Use Zip**

**for( (name,count) <- names.view.zip(Stream from 2)){**

**println(s,${count} is ${name})**

**}**

6. Filtering In Scala List

In Scala

**Filtering the List**

**val x = List(1,2,3,4,5,6,7,8,9,10)**

**x.filter( rec => rec % 2 == 0) // Only Even Records will be Available**

Filter the books which has pages greater than 120

val books = Seq(

Book("Future of Scala developers", 85),

Book("Parallel algorithms", 240),

Book("Object Oriented Programming", 130),

Book("Mobile Development", 495)

)

Books.filter{rec => rec.pages > 120 }

**TakeWhile**

**it** returns the element where the predicate is true

x.takewhile( x => x < 5)

**partition** - returns a pair of lists, one where the predicate is true,

the other where the predicate is false

val numbers = Seq(3, 7, 2, 9, 6, 5, 1, 4, 2)

numbers.partition(n => n % 2 == 0)

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

**find**  - returns the first element matching a predicate (as opposed to

returning all such elements)

**dropWhile,**

**span**

7. Map Function

**Map Function 🡺 "transforms each element of a collection based on a function."**

**val x = List(1,2,3)**

**x.map ( rec => rec \*3 ) // List(3,6,9)**

**x.map( \_ \*3) // List(3,6,9)**

**val names = List("Fred", "Joe", "Bob")**

**names.map ( \_.toLowerCase)**

**names.map(rec => <li>${rec}</li>)**

8. Sorting Function

9. List Functions

**length -** returns the length of a List

**head -** returns the first element of a List

**last -** returns the last element of a List

**init -** returns a List consisting of all elements except the last one

**tail -** returns every elements of a List except the first element

**isEmpty -** returns a Boolean indicating if the List is empty

**reverse -** returns a reversed version of the List

**flatten -** takes a list of lists and flattens it out to a single list

**mkString -** converts a List to a String

**iterator**

**toArray**

**foldLeft**

**reduceLeft**

**flatMap**

flatMap 🡺 Map + flatten

**forall**

when you need to ensure that all elements in a collection met some requirement. If at least one of the elements doesn’t correspond to the condition, you need to do something.

val numbers = Seq(3, 7, 2, 9, 6, 5, 1, 4, 2)

numbers.forall ( n => n < 10) //true

numbers.forall( n => n < 5) //false

**exists**

**Folding lists: /: and :\**

**sortWith**

**Max And Min**

case class Book(title: String, pages: Int)

val books = Seq(

Book("Future of Scala developers", 85),

Book("Parallel algorithms", 240),

Book("Object Oriented Programming", 130),

Book("Mobile Development", 495)

)

Find the Book Which has Max Pages

Books.maxBy( rec => rec.pages)

Books.minBy( rec => rec.pages )

**Fold foldLeft foldRight**

Fold takes data in one format and gives it back to you in another

The fold method for a List takes two arguments; the start value and a function

This function also takes two arguments; the accumulated value and the current item in the list.

val numbers = List(5, 4, 8, 6, 2)

**numbers.fold(0){ (tot,val) => tot+val }**

1 ) The function is then applied to its two arguments, in this case a simple addition, and returns the result.

2 ) Fold then gives the function the previous return value as its first argument and the next item in the list as its second argument, and applies it, returning the result.

3 ) This process repeats for each item of the list and returns the return value of the function once all items in the list have been iterated over**.**

**In all three folds the type of the start value must be the same as the return value)**

**Fold Left**

**We are Having the List of Foo objects now we want to display them based on the title Mr/Mrs**

**val fooList = Foo("Hugh Jass", 25, 'male) ::**

**Foo("Biggus Dickus", 43, 'male) ::**

**Foo("Incontinentia Buttocks", 37, 'female) ::**

**Nil**

**fooList.foldLeft(List[String]()) { (acc,cur) =>**

**val title = cur.sex match {**

**case ‘male’ => ‘MR.’**

**case ‘female’ => ‘Mrs.’**

**Case \_ => ‘Data Issue’**

**}**

**Acc :+ ${title} ${cur.name} ,${cur.age}**

**}**

**Difference Between Fold Foldleft FoldRight**

The primary difference is the order in which the fold operation iterates through the collection in question.

foldLeft starts on the left side—the first item—and iterates to the right;

foldRight starts on the right side—the last item—and iterates to the left.

fold goes in no particular order.

The first constraint is that the start value must be a supertype of the object you're folding. In our first example we were folding on a type List[Int] and had a start type of Int. Int is a supertype of List[Int].

The second constraint of fold is that the start value must be neutral, i.e. it must not change the result. For example, the neutral value for an addition operation would be 0, 1 for multiplication, Nil lists, etc.

**Reduce reduceLeft ReduceRight**

The foldLeft method works just like reduceLeft, but it lets you set a seed value to be used for the first element

val a = Array(12, 6, 15, 2, 20, 9)

a.reduceLeft { (x,y) => x + y }

a.reduceLeft( \_ + \_ )

a.reduceLeft (\_ \* \_)

a.reduceLeft (\_ max \_)

a.reduceLeft (\_ min \_)

Now we are going to write our own max function and supply it to reduceLeft Function

Def findMax (x:Int,y:Int) : Int = {

Val winner = x max y

Println(s,”Compared to ${x} and ${y} -- ${winner} was bigger”)

}

**Diff Union Intersect**

val num1 = Seq(1, 2, 3, 4, 5, 6)

val num2 = Seq(4, 5, 6, 7, 8, 9)

//List(1, 2, 3)

num1.diff(num2)

//List(4, 5, 6)

num1.intersect(num2)

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

num1.union(num2)

Notes

* **Using :: returns a new List(List(...), elements from the second list)**
* **Using ::: return a New List() , elements from both List a and List B will be Available**
* **If we want to use/Specify all the Elements in the List we have to use the \_ in the Map**

**Using Scala AS a Shell Script**

#!/bin/sh

exec scala "$0" "$@" $0 means File Name $@ means Input Parameters

!#

object HelloWorld {

def main(args: Array[String]) {

println("Hello, world! " + args.toList)

}

}

HelloWorld.main(args)

Calling the Script

./hello.sh Alvin

Another Example

#!/bin/sh

exec scala "$0" "$@"

!#

case class Person(name: String)

object HelloWorld {

def main(args: Array[String]) {

val al = Person("Al")

println(al)

}

}

HelloWorld.main(args)

**Read Files**

You want to open a plain-text file in Scala and process the lines in that file.

fromFile

import scala.io.Source

import java.io.{FileNotFoundException, IOException}

val filename = "no-such-file.scala"

try {

for (line <- Source.fromFile(filename).getLines) {

println(line)

}

} catch {

case e: FileNotFoundException => println("Couldn't find that file.")

case e: IOException => println("Got an IOException!")

}

def readFile(filename: String): Seq[String] = {

val bufferedSource = io.Source.fromFile(filename)

val lines = (for (line <- bufferedSource.getLines()) yield line).toList

bufferedSource.close

lines

}

**Exception Handling**

General Scala Exception Handling

try

{

// your scala code here

}

catch

{

case foo: FooException => handleFooException(foo)

case bar: BarException => handleBarException(bar)

case \_: Throwable => println("Got some other kind of exception")

}

finally

{

// your scala code here, such as to close a database connection

}

Scala try-catch-finally syntax is similar to the Java try-catch-finally syntax, except for the catch area, which uses Scala's pattern matching capabilities to handle the different exceptions you might run into

**Using Scala to Connect to Database**

package jdbc

import java.sql.DriverManager

import java.sql.Connection

/\*\*

\* A Scala JDBC connection example by Alvin Alexander,

\* http://alvinalexander.com

\*/

object ScalaJdbcConnectSelect {

def main(args: Array[String]) {

// connect to the database named "mysql" on the localhost

val driver = "com.mysql.jdbc.Driver"

val url = "jdbc:mysql://localhost/mysql"

val username = "root"

val password = "root"

// there's probably a better way to do this

var connection:Connection = null

try {

// make the connection

Class.forName(driver)

connection = DriverManager.getConnection(url, username, password)

// create the statement, and run the select query

val statement = connection.createStatement()

val resultSet = statement.executeQuery("SELECT host, user FROM user")

while ( resultSet.next() ) {

val host = resultSet.getString("host")

val user = resultSet.getString("user")

println("host, user = " + host + ", " + user)

}

} catch {

case e => e.printStackTrace

}

connection.close()

}

}

**Working With SBT**

ListSet

ListMap

Map

HashMap

TreeMap

Queue

Sequence

Set

HashSet

TreeSet

SortedSet

BitSet

Stack

Stream

Vector

Read the Content from the File and Display it as Strings

println("Read the Content From the File….")

def readLines(file:String):String ={

Source.fromInputStream(getClass.getResourceAsStream(file).getLines.mkString(‘,’)

}