Working with Scala - 1

Introduction to Scala

→ A general-purpose programming language

→ Aimed to implement common programming patterns in a concise, elegant, and type-safe way

→ Supports both object-oriented and functional programming styles, thus helping programmers to be more productive

→ Publicly released in January 2004 on the JVM platform and a few months later on the .NET platform

Martin Odersky and his team started developing Scala in 2001

Introduction to Scala(Contd.)

→ Scala is Statically Typed

- » Statically typed language binds the type to a variable for its entire scope
- » Dynamically typed languages bind the type to the actual value referenced by a variable

→ Mixed Paradigm - OOP

- » Fully supports Object Oriented Programming
- » Everything is an object in Scala
- » Unlike Java, Scala does not have primitives
- » Supports "static" class members through Singleton Object Concept
- » Improved support for OOP through Traits, similar to Ruby Modules

Introduction to Scala(Contd.)

→ Mixed Paradigm - Functional Programming

- » Scala supports Functional Programming (FP)
- » "Pure" functional languages don't allow any mutable state, thereby avoiding the need for synchronization on shared access to mutable state
- » Scala supports this model with its Actors library, but it allows for both mutable and immutable variables
- » Functions are "first-class" citizens in FP, means they can be assigned to variables, passed to other functions, etc., just like other values
- » In Scala everything is an object, functions are themselves objects in Scala
- » Scala also offers closures, similar to Python and Ruby

Frameworks in Scala

Play - For Web Development 

Play is a high-productivity Java and Scala web application framework that integrates the components and APIs you need for modern web application development

Scalding - For Map/Reduce 

Scalding is a Scala library that makes it easy to specify Hadoop MapReduce jobs. Scalding is built on top of Cascading, a Java library that abstracts away low-level Hadoop details

Akka - Actors Based Framework 

Akka is a toolkit and runtime for building highly concurrent, distributed, and fault tolerant applications on the JVM. Akka is written in Scala

Spark - In - memory Processing 

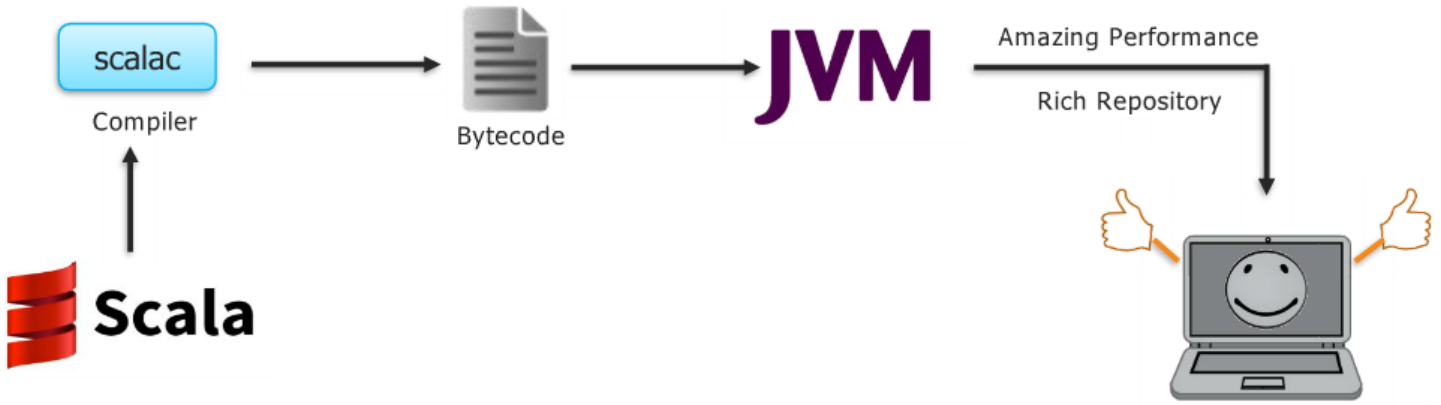
Apache Spark is a general-purpose cluster in-memory computing system. It is used for fast data analytics and it abstracts APIs in Java, Scala and Python, and provides an optimized engine that supports general execution graphs

Why Scala?

→ Developers want more flexible languages to improve their productivity

→ This resulted in evolution of scripting languages like Python, Ruby, Groovy, Clojure etc.

→ The optimizations performed by today's JVM are extraordinary, allowing byte code to outperform natively compiled code in many cases



Why Scala?

James Gosling

The father of the Java programming language

"If I were to pick a language to use today other than Java, it would be Scala"

Scala REPL

→ REPL: Read - Evaluate - Print - Loop

→ Easiest way to get started with Scala, acts as an interactive shell interpreter

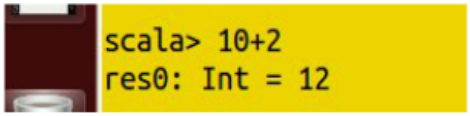
→ Even though it appears as interpreter, all typed code is converted to Bytecode and executed

→ Invoked by typing Scala as shown below

```
$ scala
Welcome to Scala version 2.9.3
Type in expressions to have them evaluated.
Type :help for more information.
scala>
```

Scala REPL Explained

→ After you type an expression, such as 10 + 2, and hit enter:
  » scala> 10 + 2
→ The interpreter will print:
  » res0: Int = 12



→ This line includes:
  » An automatically generated or user-defined name to refer to the computed value (res0, which means result 0),
  » A colon (:), followed by the type of the expression (Int),
  » An equals sign (=),
  » The value resulting from evaluating the expression (12)

Scala : Variable Types

Scala : Variable Types (Contd.)

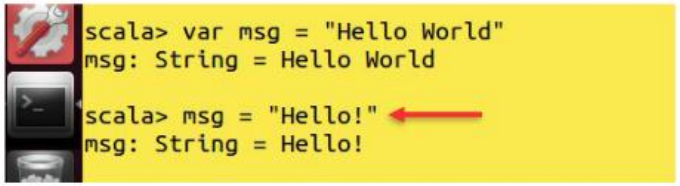
→ Scala allows one to decide whether a variable is immutable or mutable

→ Immutable - "val" (Read only)
- » Similar to Java Final Variables
- » Once initialized, Vals can't be reassigned

```
scala> val msg = "Hello World"
msg: String = Hello World

scala> msg = "Hello!"
<console>:8: error: reassignment to val
       msg = "Hello!"
```

→ Mutable - "var" (Read-write)
- » Similar to non-final variables in Java

```
scala> var msg = "Hello World"
msg: String = Hello World
scala> msg = "Hello!"
msg: String = Hello!
```



Scala: Type Inference

→ Even though we never declared variable type, Scala inferred it!

→ This is called as Type Inference in Scala

```
scala> msg = 3
<console>:8: error: type mismatch;
 found   : Int(3)
 required: String
      msg = 3
            ^
```

```
scala> msg = 3
<console>:12: error: type mismatch;
 found   : Int(3)
 required: String
       msg = 3
             ^
```

→ Once a type is assigned to a variable, it remains same for entire scope

→ Thus, Scala is Statically Typed language

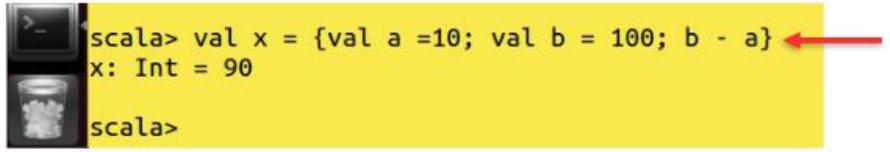
Assigning Block Expression

→ In Java or C++ a code block is a list of statements in curly braces { }

→ In Scala, a { } block is a list of expressions, and result is also an expression

→ The Value of a block is the value of the last expression of it

```
scala> val x = {val a =10; val b = 100; b - a}
x: Int = 90
```



Note: You can assign an anonymous function result to a variable/value in Scala

Lazy Values

→ You can define a value as Lazy in Scala

→ Lazy value initialization is deferred till it's accessed for first time

For example : If you want to read a file vikas, if the file is not existing or present, you will get FileNotFoundException exception.
But if you initialize the value as Lazy, you won't get this error, because it will delay the initialization till it accesses the file vikas

```
scala> val file = scala.io.Source.fromFile("vikas").mkString
java.io.FileNotFoundException: vikas (The system cannot find the file specified

    » at java.io.FileInputStream.open(Native Method)
    » at java.io.FileInputStream.<init>(FileInputStream.java:138)
    » at scala.io.Source$.fromFile(Source.scala:90)
    » at scala.io.Source$.fromFile(Source.scala:75)
    » at scala.io.Source$.fromFile(Source.scala:53)
    » at .<init>(<console>:7) ...
```

```
scala> val file = scala.io.Source.fromFile("vikas").mkString
java.io.FileNotFoundException: vikas (No such file or directory)
        at java.io.FileInputStream.open(Native Method)
        at java.io.FileInputStream.<init>(FileInputStream.java:146)
        at scala.io.Source$.fromFile(Source.scala:91)
        at scala.io.Source$.fromFile(Source.scala:76)
        at scala.io.Source$.fromFile(Source.scala:54)
        at .<init>(<console>:7)
```

Lazy Values

→ Lazy values are very useful for delaying costly initialization instructions

→ Lazy values don't give error on initialization, whereas no lazy value do give error

```
scala> lazy val file = scala.io.Source.fromFile("vikas").mkString
file: String = <lazy>
```

```
scala> lazy val file = scala.io.Source.fromFile("vikas").mkString
file: String = <lazy>
```

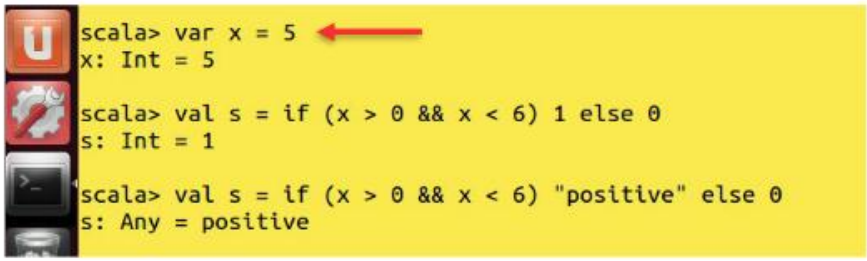
Control Structures in Scala

→ If-else syntax in Scala is same as Java or C++

→ In Scala, if-else has a value, of the expression following it

→ Semicolons are optional in Scala

```
scala> var x = 5
x: Int = 5
scala> val s = if (x > 0 && x < 6) 1 else 0
s: Int = 1
scala> val s = if (x > 0 && x < 6) "positive" else 0
s: Any = positive
```



→ Every expression in Scala has a type

→ First If statement has a type Int

→ Second statement has a type Any. Type of a mixed expression is supertype of both branches

Scala : While Loop

→ In Scala while and do-while loops are same as Java

```
var args = "500"
var i = 0
while (i < args.length)
{
println(args(i))
i += 1
}
```

```
scala> var args = "500"
args: java.lang.String = 500

scala> var i = 0
i: Int = 0

scala> while (i < args.length)
     | {
     | println(args(i))
     | i += 1
     | }
5
0
0
```

→ The ++i, or i++ operators don't work in Scala

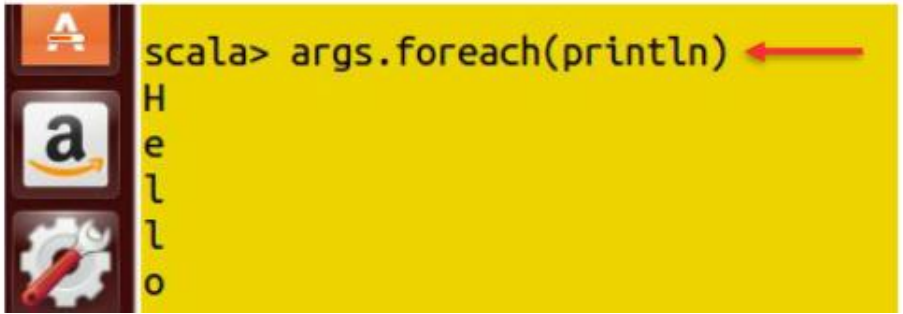
→ You'll have to use i+=1 or i=i+1 expressions instead

```
scala> while (i < args.length)
     | {
     | println(args(i))
     | i++
     | }
<console>:13: error: value ++ is not a member of Int
              i++
               ^
```

Scala : Foreach Loop

→ Looping with foreach:

```
var args = "Hello"

args.foreach(arg => println(arg))

args.foreach(println)
```



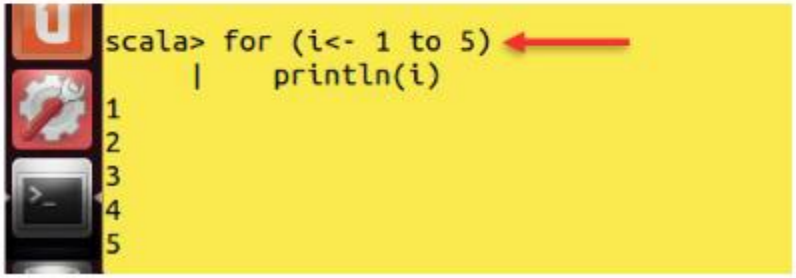
Scala : For Loop

→ For Loop:

» Scala doesn't have for (initialize; test; update) syntax

» Either you'll use a while loop or a statement like below

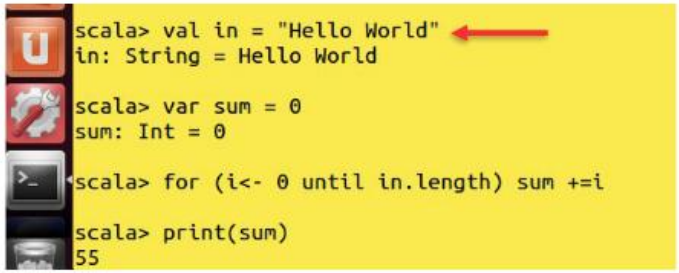
```
for (i<- 1 to 5)
  println(i)
```



Scala : For Loop

→ While traversing an array, following could be applied:

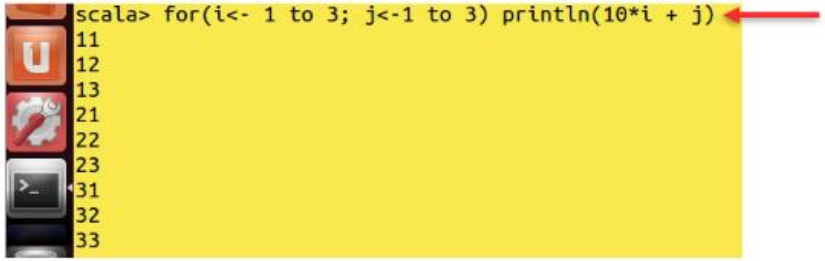
```
val in = "Hello World"
var sum = 0
for (i<- 0 until in.length) sum +=i
print(sum)
```



→ Advanced For Loop:

» We can have multiple generators in for loop

```
for(i<- 1 to 3; j<-1 to 3) println(10*i + j)
```



Scala : For Loop

» We can put conditions in multi generators for loop

```
for(i<- 1 to 3; j<-1 to 3 if i ==j) println(10*i + j)
```

```
scala> for(i<- 1 to 3; j<-1 to 3 if i ==j) println(10*i + j)
11
22
33
```

» We can introduce variables in loop!

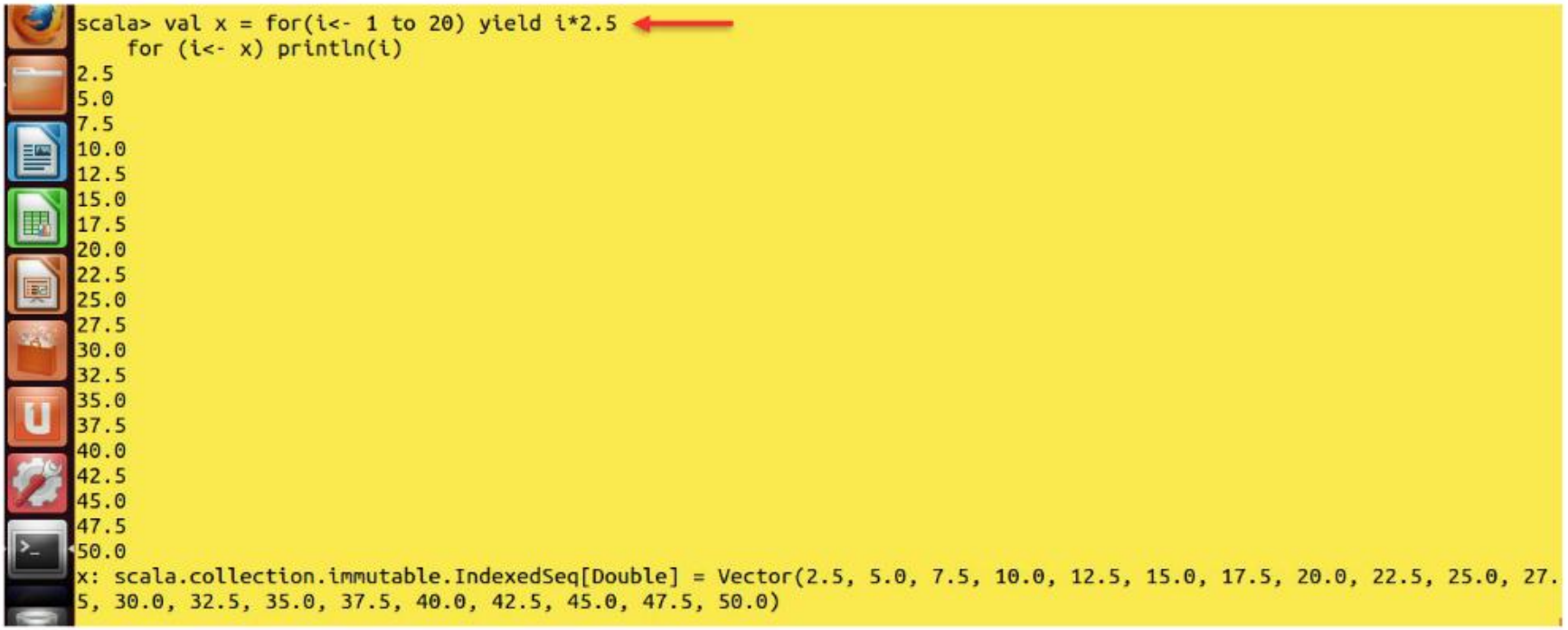
```
for(i<- 1 to 3;  x = 4-i; j<- x to 3) println(10*i + j)
```

```
scala> for(i<- 1 to 3;  x = 4-i; j<- x to 3) println(10*i + j)
13
22
23
31
32
33
```

Scala : For Loop

» If the body of for loop starts with yield, it returns a collection of values

```
val x = for(i<- 1 to 20) yield i*2.5
   for (i<- x) println(i)
```



Scala: Functions

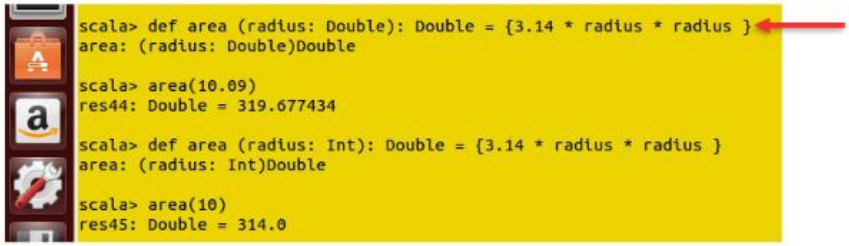
→ In addition to methods, Scala has the concept of functions

→ Methods are always invoked on objects, but functions are NOT

→ In Java, this concept is very close to a static method

→ Example:

```
def area (radius: Int): Double = {3.14 * radius * radius }
```

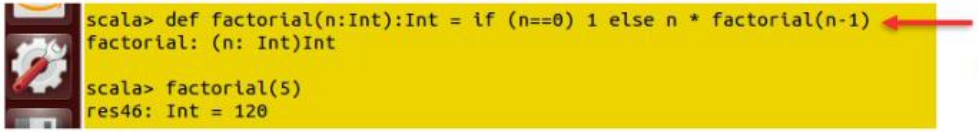


→ There is no need of a Return Statement in Scala functions

→ We need to specify the datatype for a recursive function:

→ Example:

```
def factorial(n:Int):Int = if (n==0) 1 else n * factorial(n-1)
```



Arguments to Functions

→ Named and Default Arguments

» We can provide defaults to function arguments, which will be used in case no value is provided in function calls

```
def concatStr(arg1:String, arg2:String = "Vishal", arg3:String = " Kumar")
                    concatStr("Hi! ")
```

» We can specify argument names in function calls
» In named invocations the order of arguments is not necessary
» We can mix unnamed and named arguments, if the unnamed argument is the first one

→ Variable Arguments

» Scala supports variable number of arguments to a function

Scala: Procedures

→ Scala has special functions which don't return any value

→ If there is a scala function without a preceding "=" symbol, then the return type of the function is Unit

→ Such functions are called Procedures

→ Procedures do not return any value in Scala

→ Example:

```
def rect_area(length:Float, breadth:Float) { val area = length* breadth; print(area)}
```

```
scala> def rect_area(length:Float, breadth:Float) { val area = length* breadth; print(area)}
rect_area: (length: Float, breadth: Float)Unit
```

→ Same rules of default and named arguments apply on Procedures as well

Scala : Collections

→ Scala has a rich library of Collections, they are:

- Array
- ArrayBuffers
- Maps
- Tuples
- Lists

Scala Collections : Array

→ Fixed Length Arrays:

→ Examples:

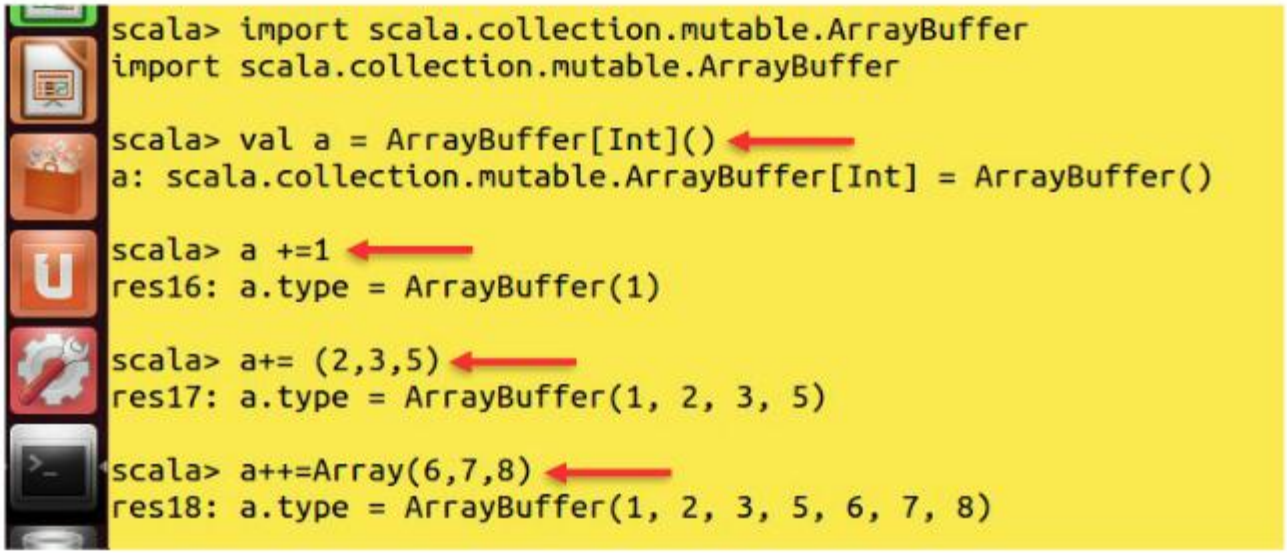
```
val n = new Array[Int](10)
val s = new Array[String](10)
val st = Array("Hello", "World")
```

```
scala> val n = new Array[Int](10)
n: Array[Int] = Array(0, 0, 0, 0, 0, 0, 0, 0, 0, 0)

scala> val s = new Array[String](10)
s: Array[String] = Array(null, null, null, null, null, null, null, null, null, null)

scala> val st = Array("Hello", "World")
st: Array[String] = Array(Hello, World)
```

Scala Collections : ArrayBuffer

→ Variable Length Arrays (Array Buffers)
→ Similar to Java ArrayLists

```
import scala.collection.mutable.ArrayBuffer
val a = ArrayBuffer[Int]()
a +=1
a+= (2,3,5)
a++=Array(6,7,8)
```



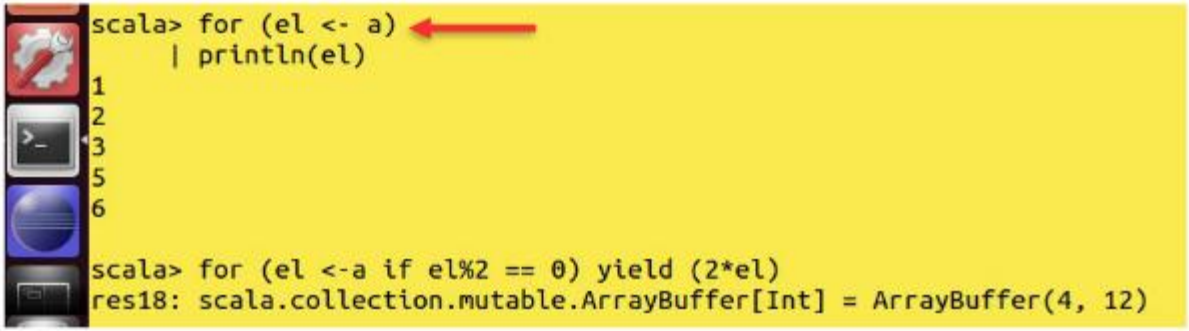
Scala Collections : Array and ArrayBuffer

→ Common Operations:

```
a.trimEnd(2)   //Removes last 2 elements
a.insert(2, 9)  // Adds element at 2nd index
a.insert (2,10,11,12)  //Adds a list
a.remove(2)   //Removes an element
a.remove(2,3)  //Removes three elements from index 2
```

→ Traversing and Transformation:

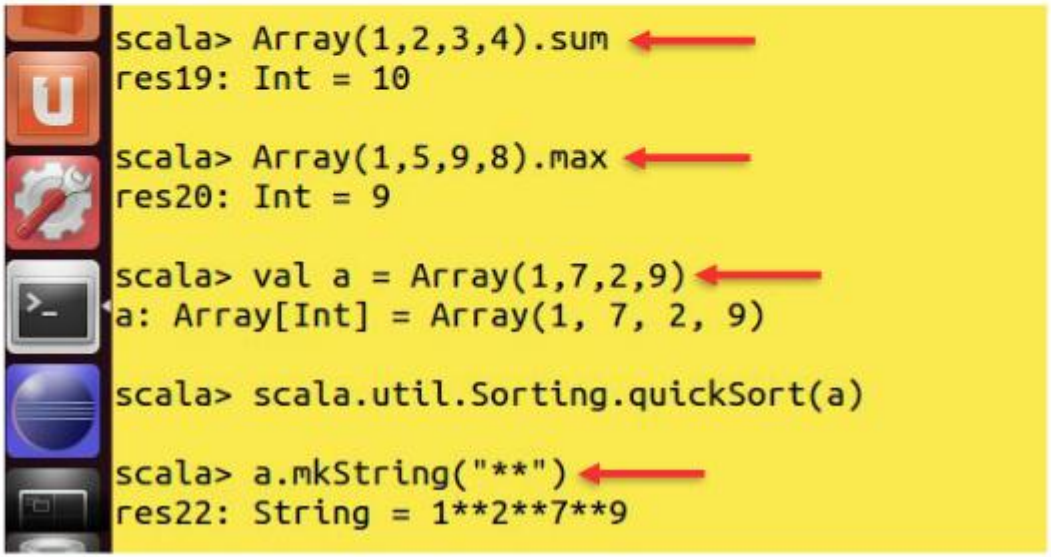
```
for (el <- a)
        println(el)

        for (el <-a if el%2 == 0) yield (2*el)
```



Scala Collections : Array and ArrayBuffer

→ Common Operations:

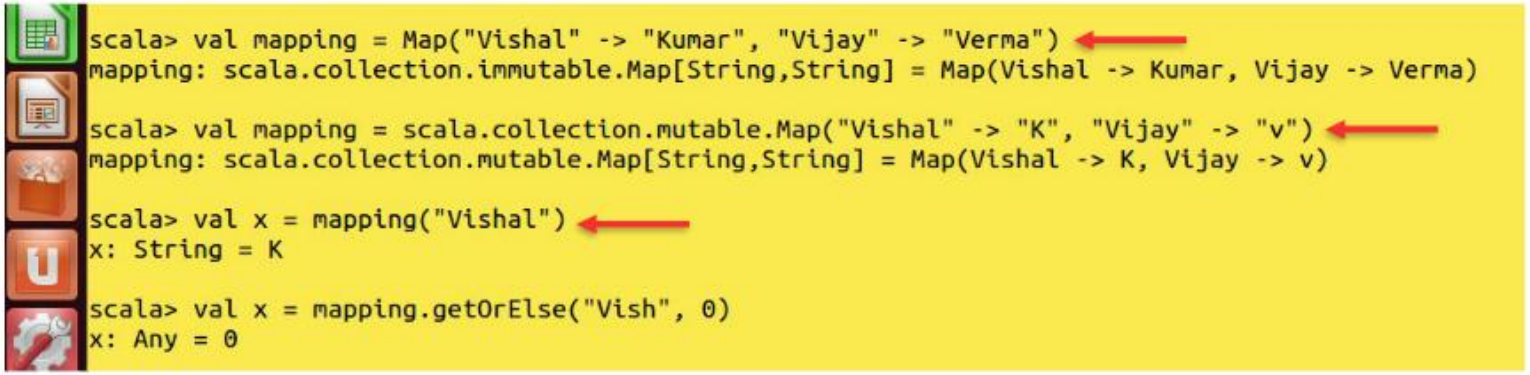
```
Array(1,2,3,4).sum
Array(1,5,9,8).max
val a = Array(1,7,2,9)
scala.util.Sorting.quickSort(a)
a.mkString( " ** ")
```



Scala Collections : Maps

→ In Scala, a map is a collection of Pair
→ A pair is a group of two values (Not necessarily of same type)

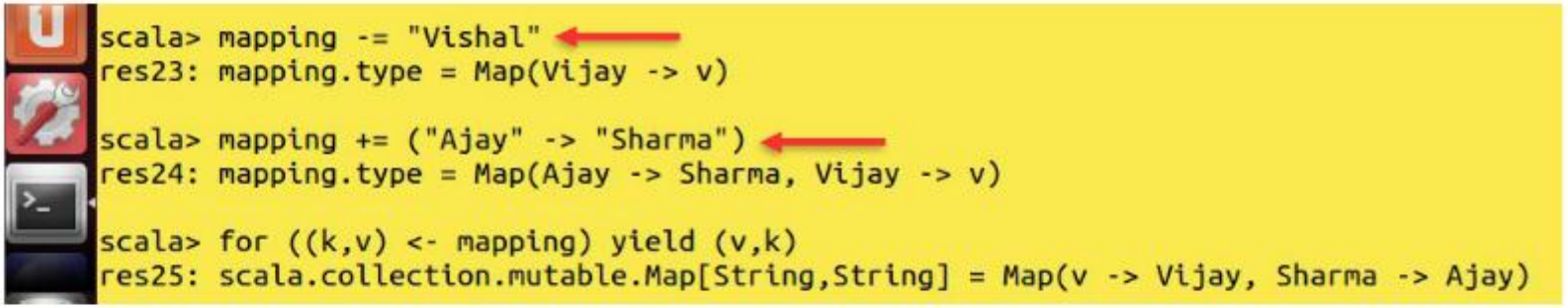
```
val mapping = Map("Vishal" -> "Kumar", "Vijay" -> "Verma")
val mapping = scala.collection.mutable.Map("Vishal" -> "K", "Vijay" -> "V")
```



Scala Collections : Maps (Contd.)

→ Accessing Maps:

```
val x = mapping("Vishal")
val x = mapping.getOrElse("Vish", 0)
mapping -= "Vishal"
mapping += ("Ajay" -> "Sharma")
```



→ Iterating Maps:

```
for ((k,v) <- mapping) yield (v,k)
```

Scala Collections: Tuples

→ Tuple is more generalized form of pair

→ Tuple has more than two values of potentially different types

```
val a = (1,4, "Bob", "Jack")
```

```
scala> val a = (1,4, "Bob", "Jack")
a: (Int, Int, String, String) = (1,4,Bob,Jack)

scala> a._2
res26: Int = 4

scala> a _2
warning: there were 1 feature warning(s); re-run with -feature for details
res27: Int = 4
```

→ Accessing the tuple elements:

```
a._2  or a _2//Returns 4
```

→ In tuples the offset starts with 1 and NOT from 0

→ Tuples are typically used for the functions which return more than one value:

```
"New Delhi India".partition(_.isUpper)
```

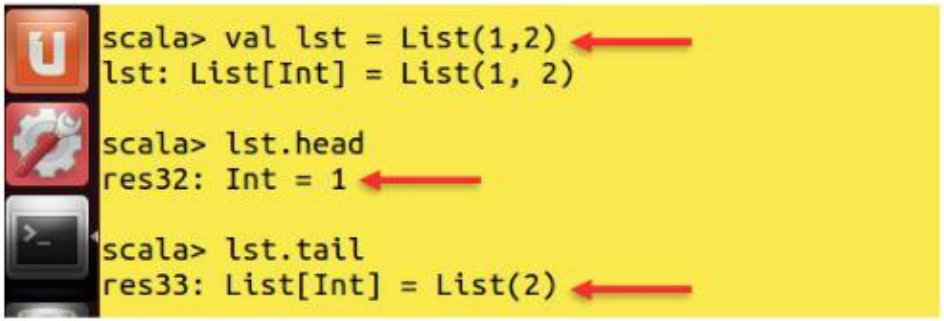
```
scala> "New Delhi India".partition(_.isUpper)
res28: (String, String) = (NDI,ew elhi ndia)
```

Scala Collections: Lists

→ List is either Nil or a combination of head and tail elements where tail is again a List

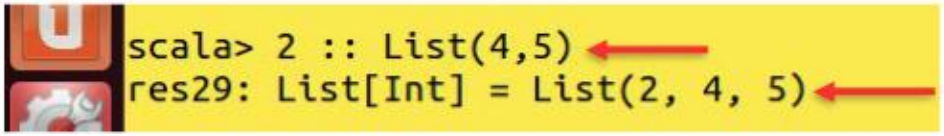
→ Example:

```
val lst = List(1,2)
lst.head = 1
lst.tail = List(2)
```



→ :: operator adds a new List from given head and tail

```
2 :: List(4,5)
List[Int] = List(2, 4, 5)
```

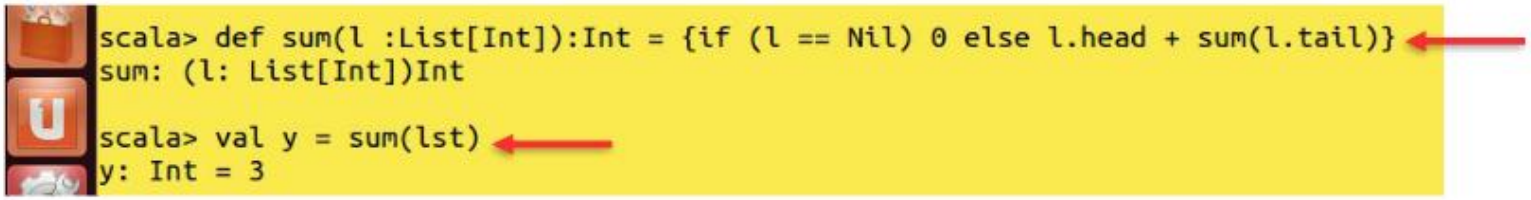


Scala Collections: Lists (Contd.)

→ We can use iterator to iterate over a list, but recursion is a preferred practice in Scala

→ Example:

```
def sum(l :List[Int]):Int = {if (l == Nil) 0 else l.head + sum(l.tail)}
val y = sum(lst)
```



Question

The ArrayBuffer has a fixed number of elements

- True
- False

Answer

- False

Question

The result of a block is a:

- Var
- Val
- Expression
- None

Answer

- Expression

Question

Val in Scala is like instance variable in Java

- True
- False

Answer

- False

Question

Named and Unnamed Arguments can be mixed:

- No, they can't be
- Yes, in any order
- Yes, only if the first argument is unnamed

Answer

- Yes, only if the first argument is unnamed

Question

What is the outcome of following:

```
val in = Array(1,2,3,4,5)
val sum = 0
for (i<- 0 until in.length) sum +=i
print(sum)
```

- 10
- 12345
- Error

Answer

- Error

Question

What is the output of following code:

```
val a = Map("a" ->1, "b" ->1)
a("c") =2
```

- A = (“a”->1, “b” ->1, “c” ->2)
- A = (“a”->1, “b” ->2, “c” ->2)
- Error

Answer

- Error

Thank You