Scala basics

Login to VM with training as user and change the directory.

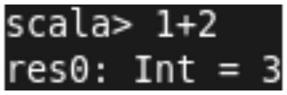
```
cd /home/training/spark_data
spark-shell
```

Once inside the spark-shell, we should be able to see Scala terminal. Also a web UI would be started to monitor spark.

Execute below commands to understand basics of Scala

Performing simple arithmetic operations in Scala

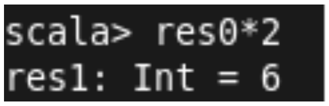
```
1+2
```



```
scala> 1+2
res0: Int = 3
```

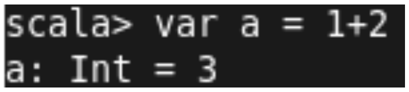
Note: Scala has assigned a default name to the result set. Also identified the datatype by itself

```
res0*2
```



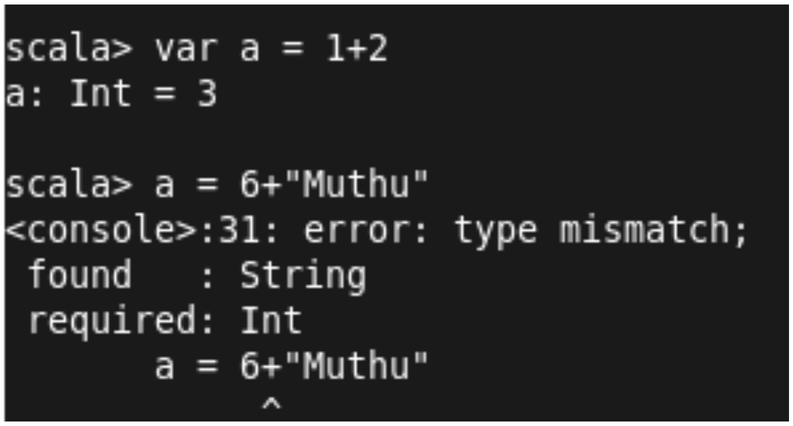
```
scala> res0*2
res1: Int = 6
```

```
var a = 1+2
```



```
scala> var a = 1+2
a: Int = 3
```

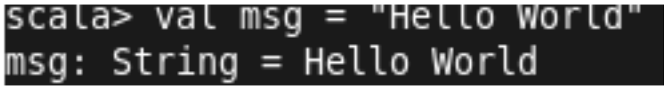
```
a = 6+"Muthu"
```



```
scala> var a = 1+2
a: Int = 3

scala> a = 6+"Muthu"
<console>:31: error: type mismatch;
 found   : String
 required: Int
       a = 6+"Muthu"
            ^
```

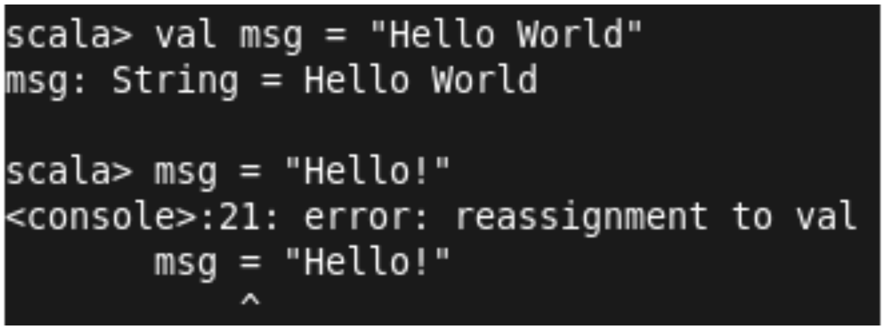
Note: The error occurred because of datatype mismatch. It itself identifies that operation on a has to be of type integer.

```
val msg = "Hello World"
```



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

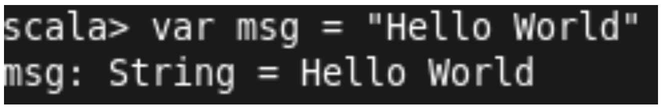
```
msg = "Hello!"
```



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

scala> msg = "Hello!"
<console>:21: error: reassignment to val
       msg = "Hello!"
           ^
```

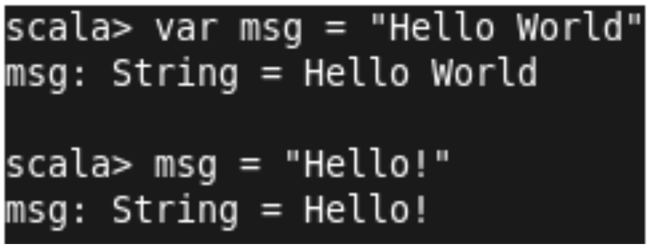
Note: Error occurred because ***val*** is immutable. So, the value cannot be reassigned

```
var msg = "Hello World"
```



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

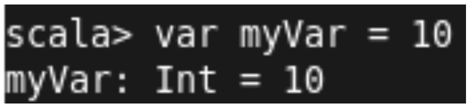
```
msg = "Hello!"
```



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

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

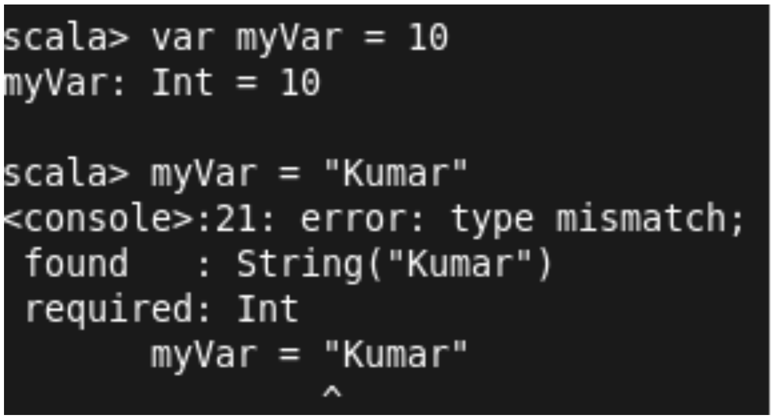
Note: No error occurred while reassignment because ***var*** is mutable.

```
var myVar = 10;
```



```
scala> var myVar = 10
myVar: Int = 10
```

```
myVar = "Kumar"
```



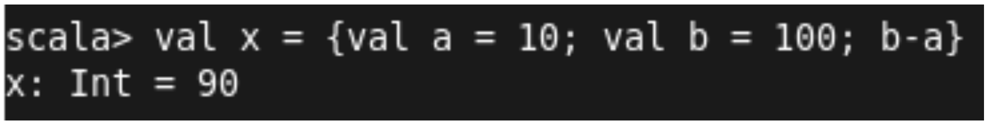
```
scala> var myVar = 10
myVar: Int = 10

scala> myVar = "Kumar"
<console>:21: error: type mismatch;
 found   : String("Kumar")
 required: Int
       myVar = "Kumar"
               ^
```

Note: Error occurred because of datatype mismatch. ***Var*** is assigned value of Integer at the first time it is assigned.

Expressions in Scala

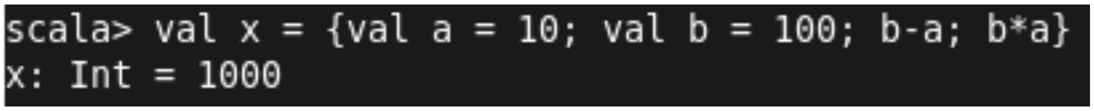
In Scala, a {} block is a list of expressions, and result is also an expression.The value of the block is the value of last expression of it.

```
val x = {val a = 10; val b = 100; b-a}
```



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

```
val x = {val a = 10; val b = 100; b-a; b*a}
```



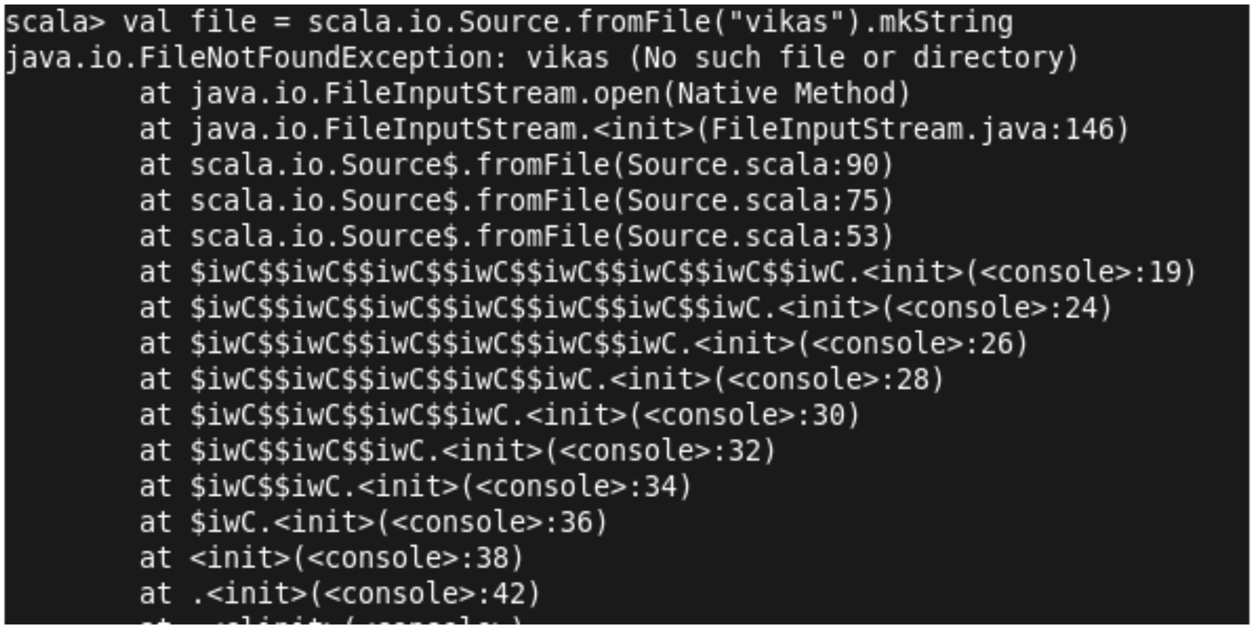
```
scala> val x = {val a = 10; val b = 100; b-a; b*a}
x: Int = 1000
```

Lazy Value concept in Scala

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

Eg : 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 access the file vikas

```
val file =
scala.io.Source.fromFile("/home/training/spark_data/vikas").mkString
```



```
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$.fromFile(Source.scala:90)
        at scala.io.Source$.fromFile(Source.scala:75)
        at scala.io.Source$.fromFile(Source.scala:53)
        at $iwC$$iwC$$iwC$$iwC$$iwC$$iwC$$iwC$$iwC.<init>(<console>:19)
        at $iwC$$iwC$$iwC$$iwC$$iwC$$iwC$$iwC.<init>(<console>:24)
        at $iwC$$iwC$$iwC$$iwC$$iwC$$iwC.<init>(<console>:26)
        at $iwC$$iwC$$iwC$$iwC$$iwC.<init>(<console>:28)
        at $iwC$$iwC$$iwC$$iwC.<init>(<console>:30)
        at $iwC$$iwC$$iwC.<init>(<console>:32)
        at $iwC$$iwC.<init>(<console>:34)
        at $iwC.<init>(<console>:36)
        at <init>(<console>:38)
        at .<init>(<console>:42)
```

```
val file = scala.io.Source.fromFile("/home/training/spark_data/readFile").mkString
```

o/p

file: String = Welcome to spark and scala training.

```
lazy val file1 = scala.io.Source.fromFile("/home/training/spark_data/readFileLazyNotExists").mkString
```

o/p

file1: String <lazy>

```
println(file1)
```

o/p

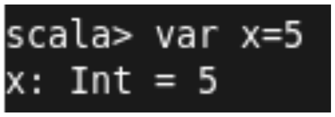
Throws error message

Lazy values are very useful for delaying costly initialization instructions

Control structures in Scala

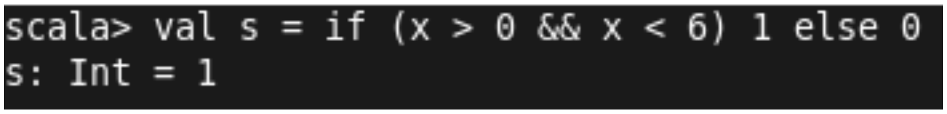
Scala: If - Else

```
var x=5
```



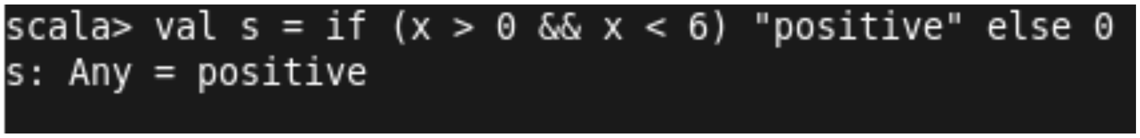
```
scala> var x=5
x: Int = 5
```

```
val s = if (x > 0 && x < 6) 1 else 0
```



```
scala> val s = if (x > 0 && x < 6) 1 else 0
s: Int = 1
```

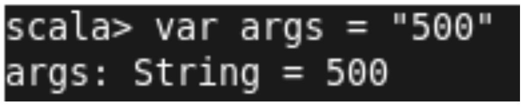
```
val s = if (x > 0 && x < 6) "positive" else 0
```



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

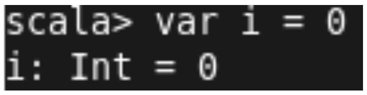
Scala: While Loop

```
var args = "500"
```



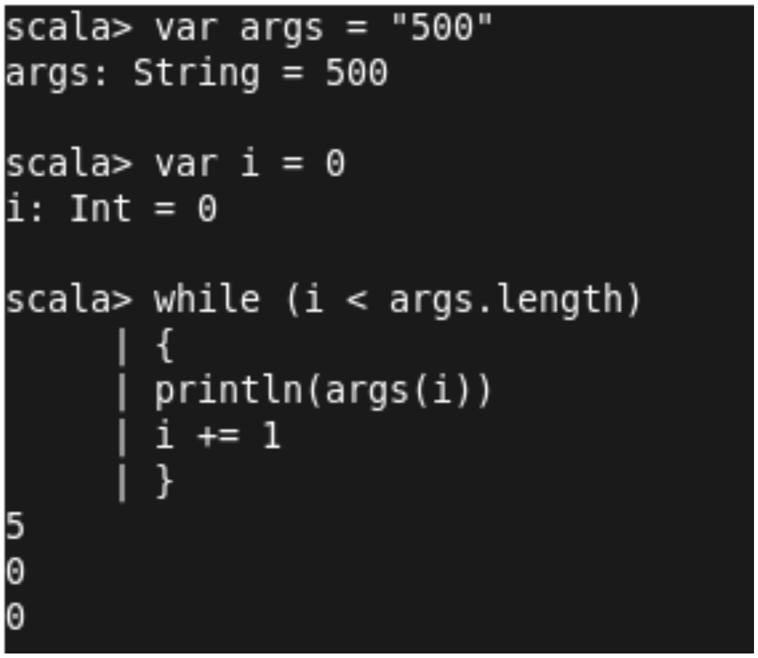
```
scala> var args = "500"
args: String = 500
```

```
var i = 0
```



```
scala> var i = 0
i: Int = 0
```

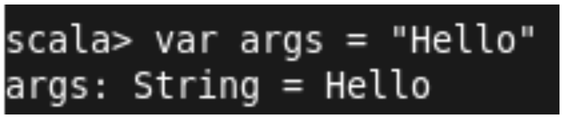
```
while (i < args.length)
    {
    println(args(i))
    i += 1
    }
```



```
scala> var args = "500"
args: String = 500

scala> var i = 0
i: Int = 0

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

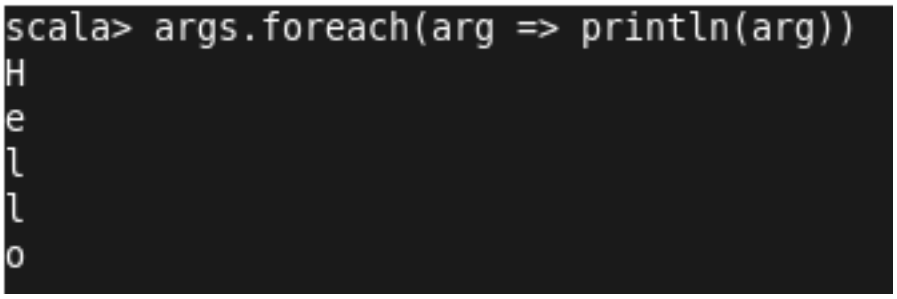
Scala: Foreach loop

```
var args = "Hello"
```



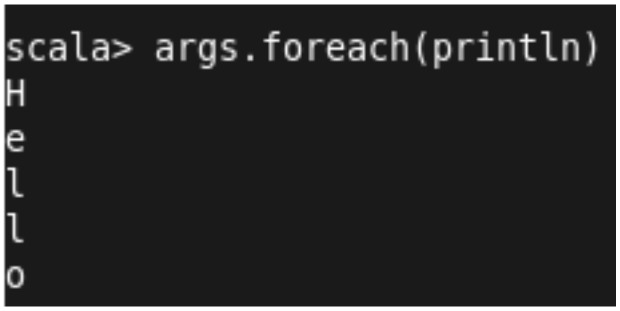
```
scala> var args = "Hello"
args: String = Hello
```

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



```
scala> args.foreach(arg => println(arg))
H
e
l
l
o
```

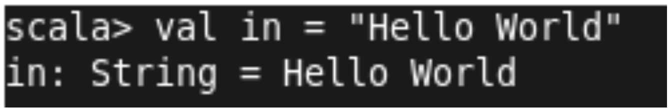
```
args.foreach(println)
```



```
scala> args.foreach(println)
H
e
l
l
o
```

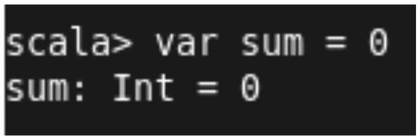
Scala: For loop

```
val in = "Hello World"
```



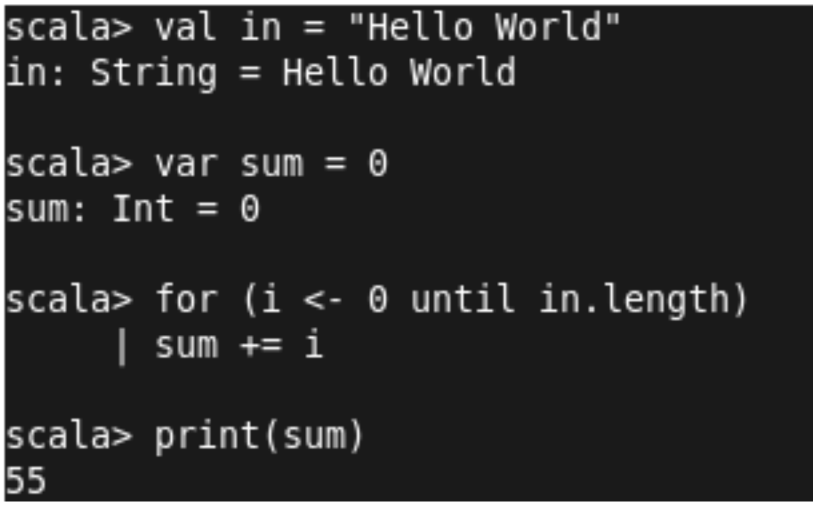
```
scala> val in = "Hello World"
in: String = Hello World
```

```
var sum = 0
```



```
scala> var sum = 0
sum: Int = 0
```

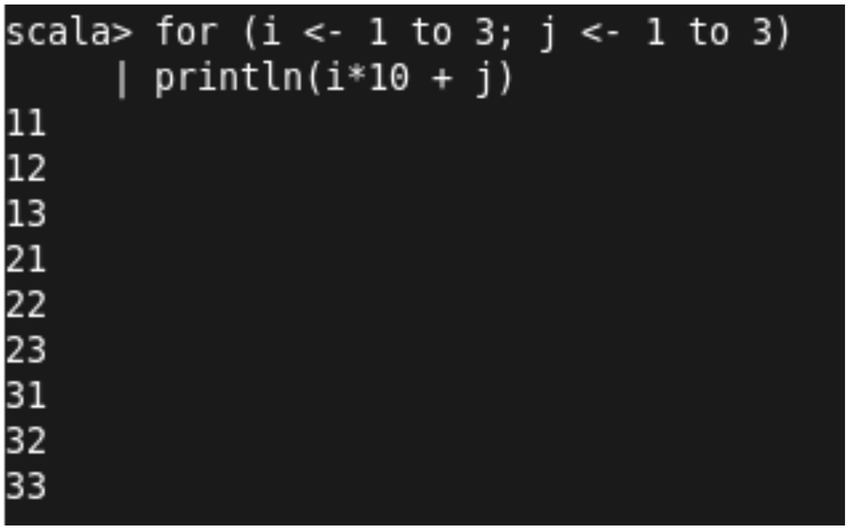
```
for (i <- 0 until in.length)
    sum += i

print(sum)
```



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

scala> var sum = 0
sum: Int = 0

scala> for (i <- 0 until in.length)
     | sum += i

scala> print(sum)
55
```

Scala: Advanced For Loop

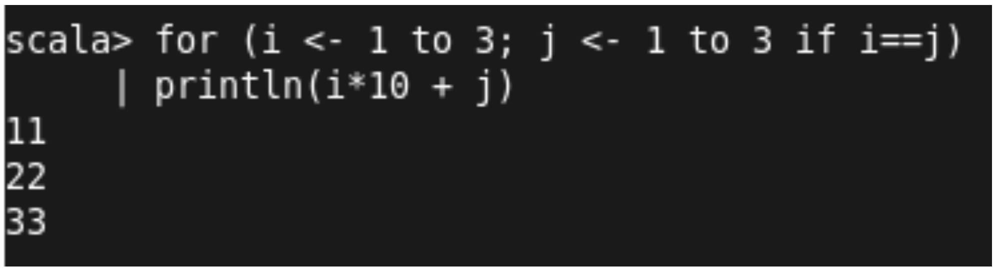
```
for (i <- 1 to 3; j <- 1 to 3)
    println(i*10 + j)
```



```
scala> for (i <- 1 to 3; j <- 1 to 3)
     | println(i*10 + j)
11
12
13
21
22
23
31
32
33
```

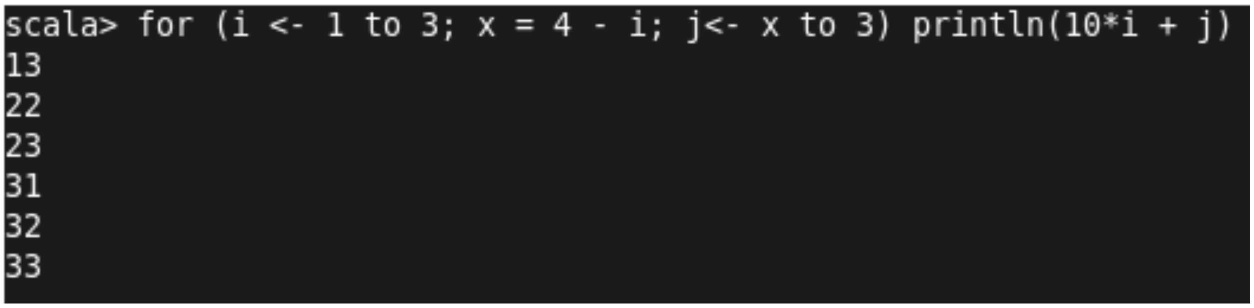
Scala: Conditional for loop

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


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

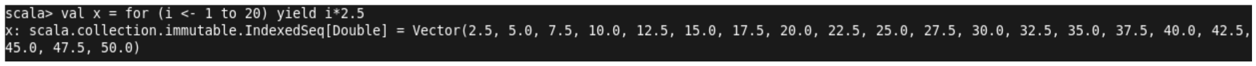
Scala: Variables in for loop

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

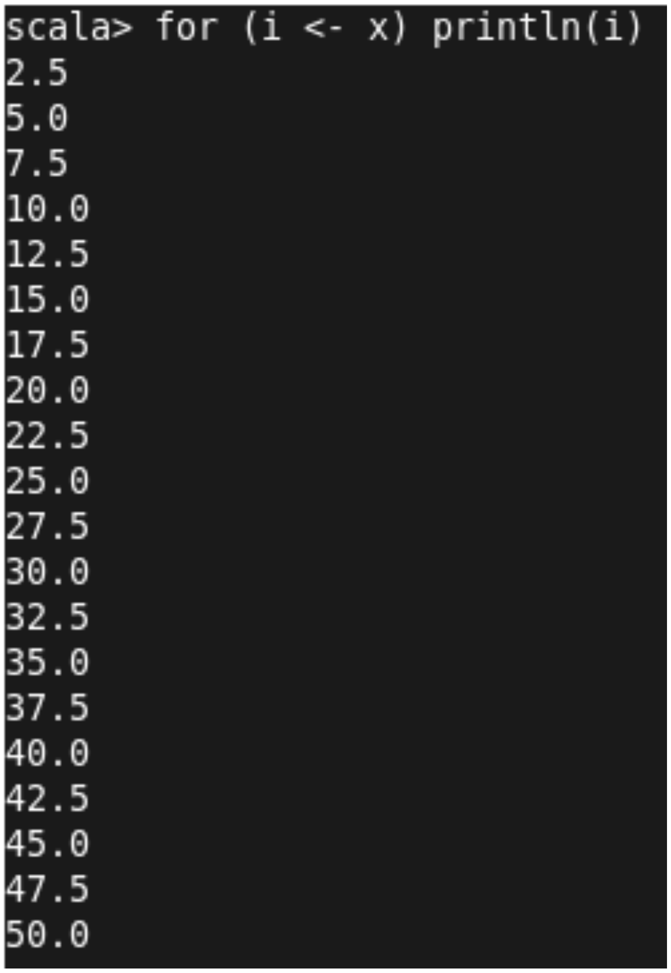

```
scala> for (i <- 1 to 3; x = 4 - i; j<- x to 3) println(10*i + j)
13
22
23
31
32
33
```

Scala: For Loop with Yield concept

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


```
scala> val x = for (i <- 1 to 20) yield i*2.5
x: scala.collection.immutable.IndexedSeq[Double] = Vector(2.5, 5.0, 7.5, 10.0, 12.5, 15.0, 17.5, 20.0, 22.5, 25.0, 27.5, 30.0, 32.5, 35.0, 37.5, 40.0, 42.5,
45.0, 47.5, 50.0)
```

```
for (i <- x) println(i)
```



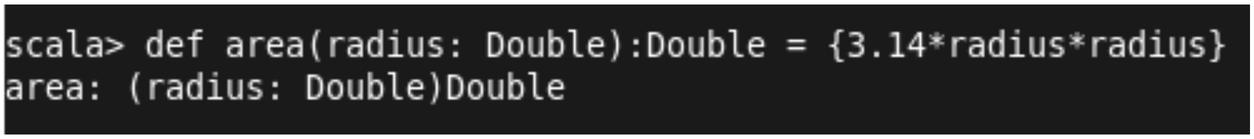
```
scala> for (i <- x) println(i)
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
22.5
25.0
27.5
30.0
32.5
35.0
37.5
40.0
42.5
45.0
47.5
50.0
```

Scala: Functions

def area(radius: Int): Double = {3.14*radius*radius}
Note: There is no need of return statement in scala functions

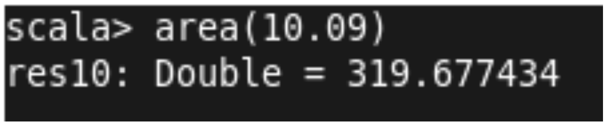
def factorial(n:Int):Int = if (n==0) 1 else n*factorial(n-1)
Note: We need to specify the datatype for recursive function

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



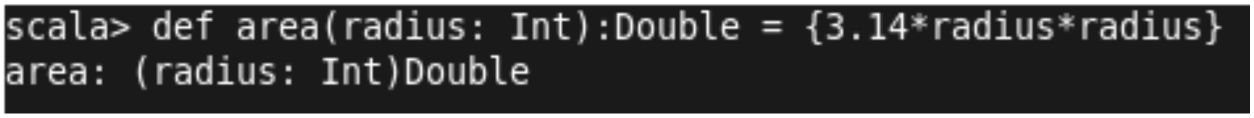
```
scala> def area(radius: Double):Double = {3.14*radius*radius}
area: (radius: Double)Double
```

```
area(10.09)
```



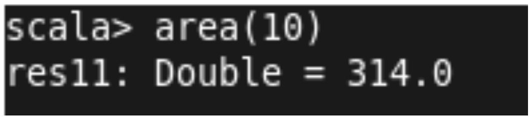
```
scala> area(10.09)
res10: Double = 319.677434
```

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



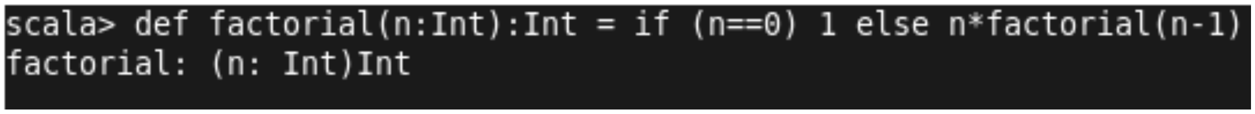
```
scala> def area(radius: Int):Double = {3.14*radius*radius}
area: (radius: Int)Double
```

```
area(10)
```



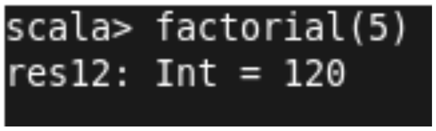
```
scala> area(10)
res11: Double = 314.0
```

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



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

```
factorial(5)
```



```
scala> factorial(5)
res12: Int = 120
```

*Scala: Functions with Argument*s

Default Argument

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

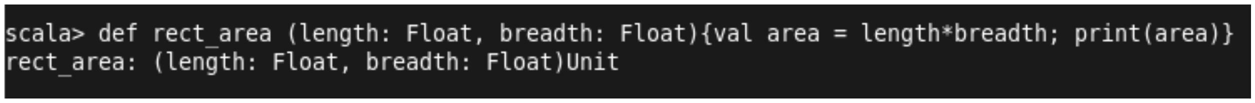
We can specify argument names in function calls. In named invocations the order of arguments is not necessary. We can mix unnamed and named arguments, if the unnamed argument is the first one

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

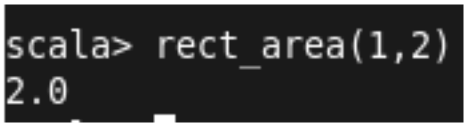
Procedures do not return any value in Scala.

```
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
```

```
rect_area(1,2)
```



```
scala> rect_area(1,2)
2.0
```

Note: Same rules of default and named arguments apply on procedures as well

Scala: Collections

Different type of collections available in Scala are:

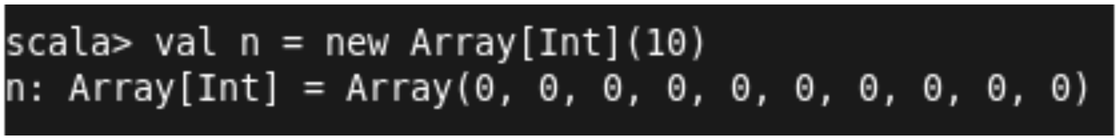
- Array

- ArrayBuffers
- Maps
- Tuples
- Lists

Scala Collections: Array

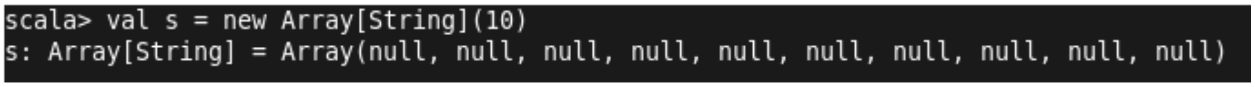
Fixed length arrays

```
val n = new Array[Int](10)
```



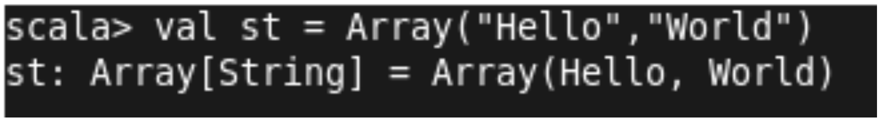
```
scala> val n = new Array[Int](10)
n: Array[Int] = Array(0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
```

```
val s = new Array[String](10)
```



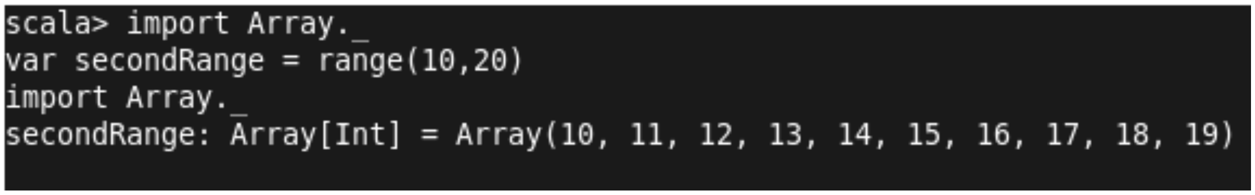
```
scala> val s = new Array[String](10)
s: Array[String] = Array(null, null, null, null, null, null, null, null, null, null)
```

```
val st = Array("Hello","World")
```



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

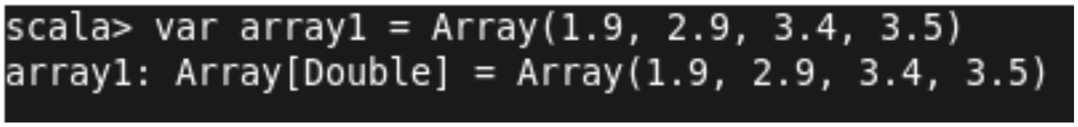
```
import Array._
var secondRange = range(10,20)
```



```
scala> import Array._
var secondRange = range(10,20)
import Array._
secondRange: Array[Int] = Array(10, 11, 12, 13, 14, 15, 16, 17, 18, 19)
```

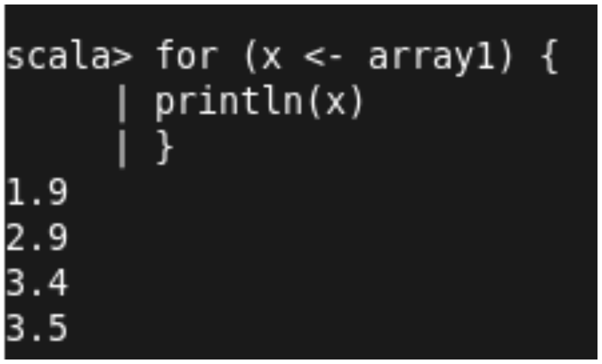
Single dimensional array

```
var array1 = Array(1.9, 2.9, 3.4, 3.5)
```



```
scala> var array1 = Array(1.9, 2.9, 3.4, 3.5)
array1: Array[Double] = Array(1.9, 2.9, 3.4, 3.5)
```

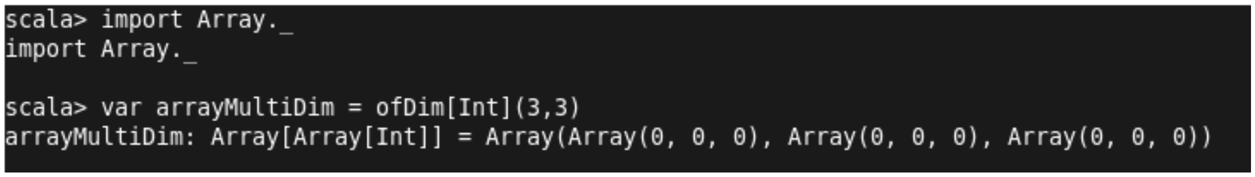
```
for (x <- array1) {
    println(x)

    }
```



```
scala> for (x <- array1) {
     | println(x)
     | }
1.9
2.9
3.4
3.5
```

Array Multi dimensional

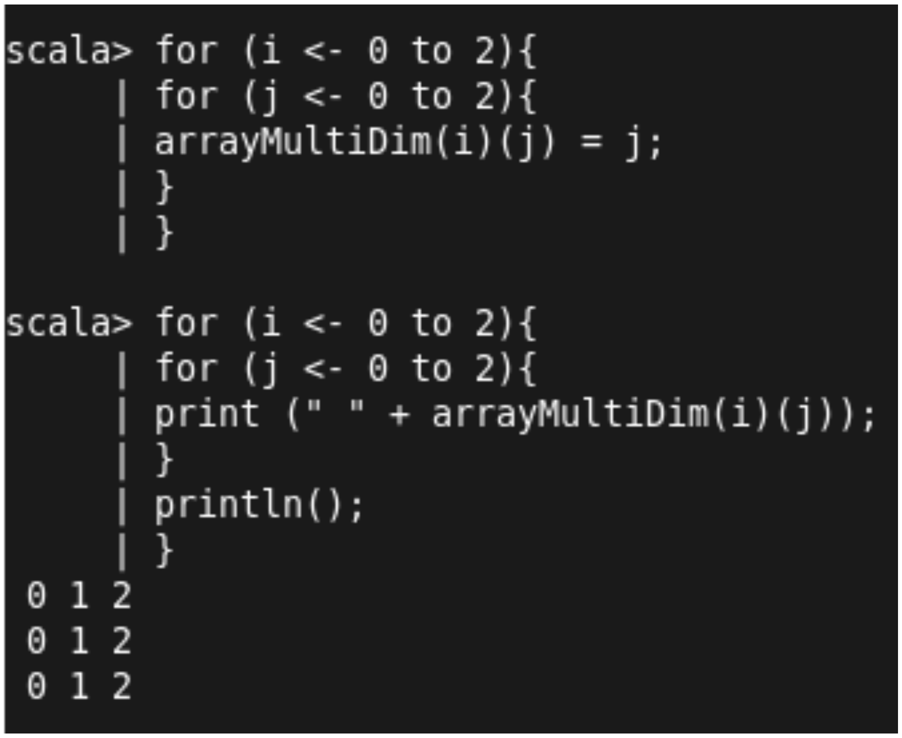
```
import Array._

var arrayMultiDim = ofDim[Int](3,3)
```



```
scala> import Array._
import Array._

scala> var arrayMultiDim = ofDim[Int](3,3)
arrayMultiDim: Array[Array[Int]] = Array(Array(0, 0, 0), Array(0, 0, 0), Array(0, 0, 0))
```

```
for (i <- 0 to 2){
    for (j <- 0 to 2){
    arrayMultiDim(i)(j) = j;
    }
    }
```

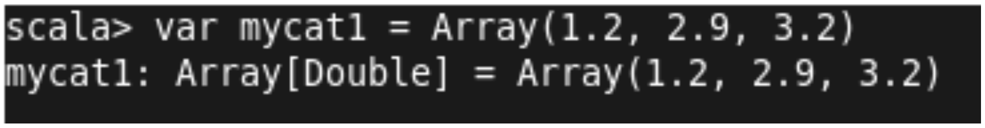
```
for (i <- 0 to 2){
    for (j <- 0 to 2){
    print (" " + arrayMultiDim(i)(j));
    }
    println();
    }
```



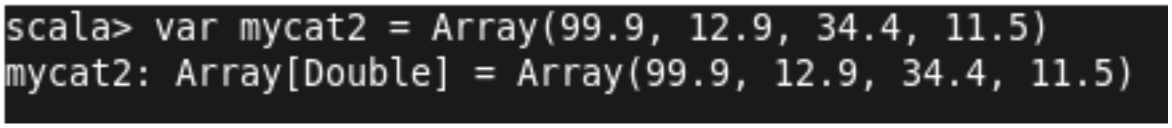
```
scala> for (i <- 0 to 2){
     | for (j <- 0 to 2){
     | arrayMultiDim(i)(j) = j;
     | }
     | }

scala> for (i <- 0 to 2){
     | for (j <- 0 to 2){
     | print (" " + arrayMultiDim(i)(j));
     | }
     | println();
     | }
 0 1 2
 0 1 2
 0 1 2
```

Array Concatenation

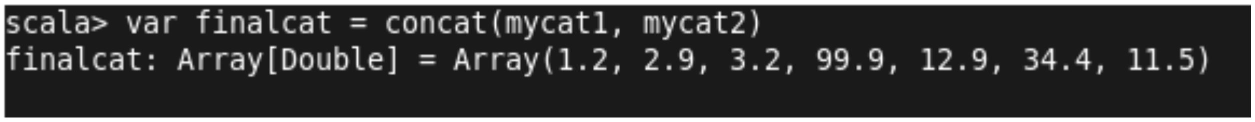
```
var mycat1 = Array(1.2, 2.9, 3.2)
```


```
scala> var mycat1 = Array(1.2, 2.9, 3.2)
mycat1: Array[Double] = Array(1.2, 2.9, 3.2)
```

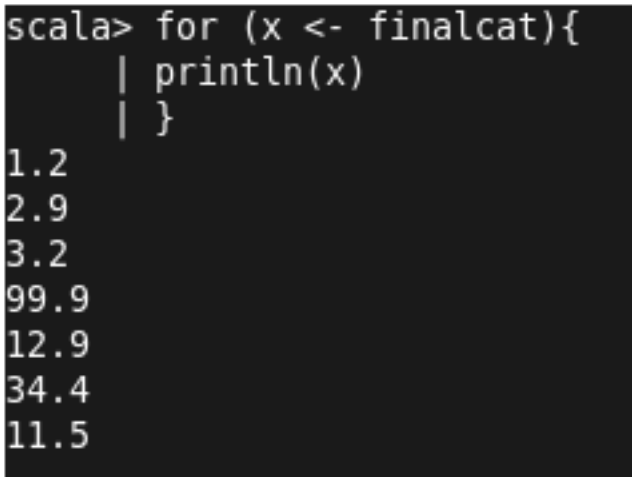
```
var mycat2 = Array(99.9, 12.9, 34.4, 11.5)
```


```
scala> var mycat2 = Array(99.9, 12.9, 34.4, 11.5)
mycat2: Array[Double] = Array(99.9, 12.9, 34.4, 11.5)
```

```
var finalcat = concat(mycat1, mycat2)
```

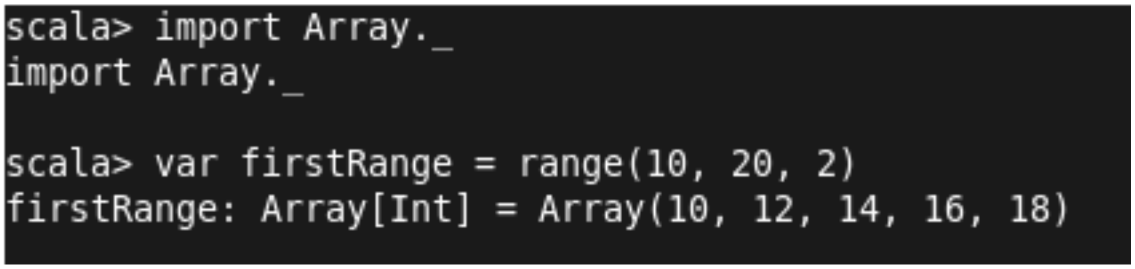

```
scala> var finalcat = concat(mycat1, mycat2)
finalcat: Array[Double] = Array(1.2, 2.9, 3.2, 99.9, 12.9, 34.4, 11.5)
```

```
for (x <- finalcat){
    println(x)
    }
```


```
scala> for (x <- finalcat){
     | println(x)
     | }
1.2
2.9
3.2
99.9
12.9
34.4
11.5
```

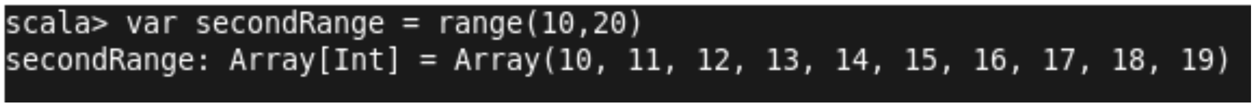
```
import Array._
```

```
var firstRange = range(10, 20, 2)
```



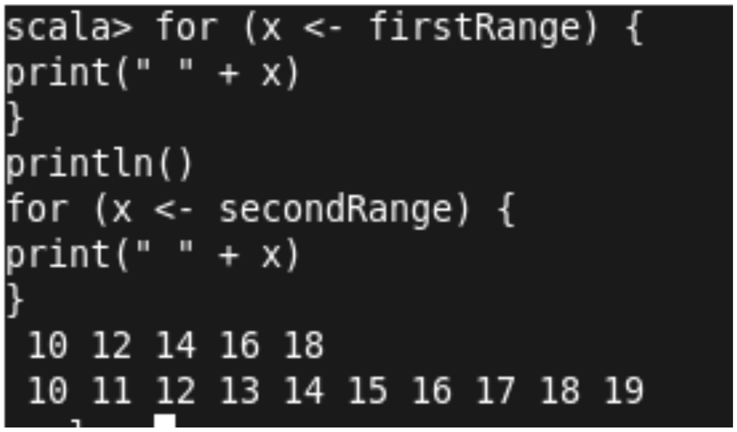
```
scala> import Array._
import Array._

scala> var firstRange = range(10, 20, 2)
firstRange: Array[Int] = Array(10, 12, 14, 16, 18)
```

```
var secondRange = range(10,20)
```



```
scala> var secondRange = range(10,20)
secondRange: Array[Int] = Array(10, 11, 12, 13, 14, 15, 16, 17, 18, 19)
```

```
for (x <- firstRange) {
print(" " + x)
}
println()
for (x <- secondRange) {
print(" " + x)
}
```

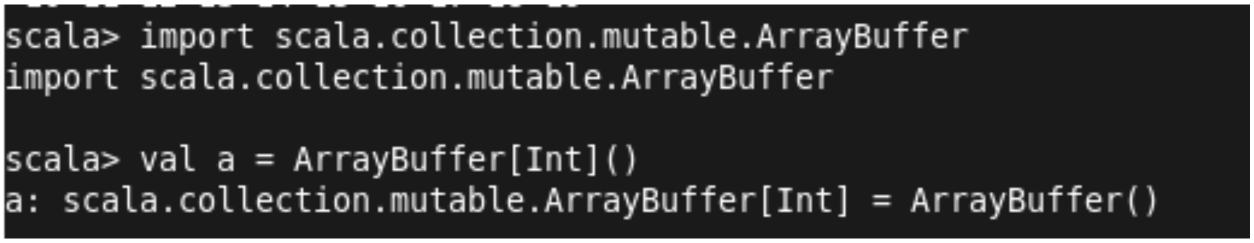


```
scala> for (x <- firstRange) {
print(" " + x)
}
println()
for (x <- secondRange) {
print(" " + x)
}
 10 12 14 16 18
 10 11 12 13 14 15 16 17 18 19
```

Scala Collections: ArrayBuffer

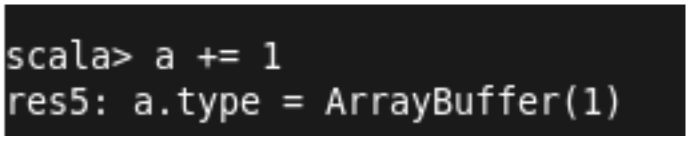
Variable length arrays(Array Buffers)
Similar to java ArrayLists

```
import scala.collection.mutable.ArrayBuffer

val a = ArrayBuffer[Int]()
```



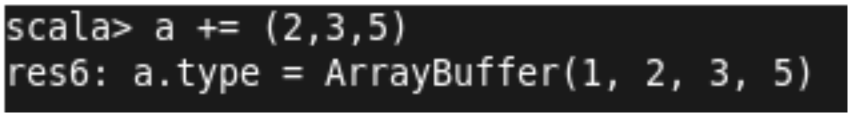
```
scala> import scala.collection.mutable.ArrayBuffer
import scala.collection.mutable.ArrayBuffer

scala> val a = ArrayBuffer[Int]()
a: scala.collection.mutable.ArrayBuffer[Int] = ArrayBuffer()
```

```
a += 1
```



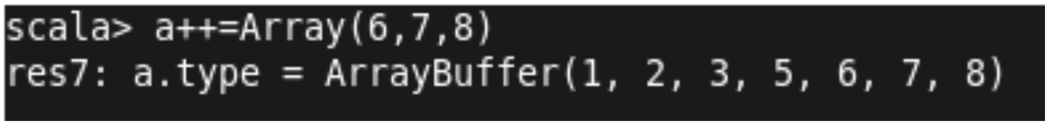
```
scala> a += 1
res5: a.type = ArrayBuffer(1)
```

```
a += (2,3,5)
```



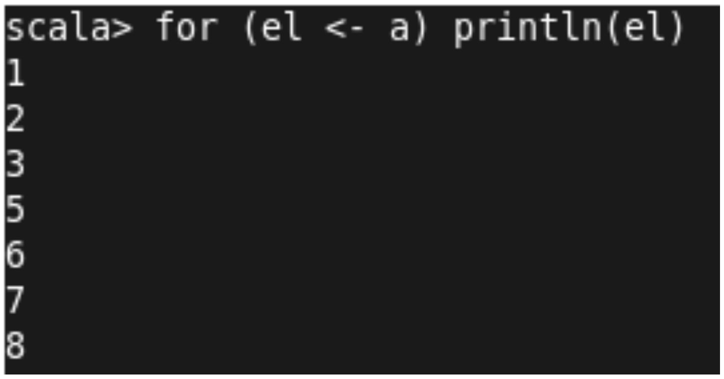
```
scala> a += (2,3,5)
res6: a.type = ArrayBuffer(1, 2, 3, 5)
```

```
a++=Array(6,7,8)
```



```
scala> a++=Array(6,7,8)
res7: a.type = ArrayBuffer(1, 2, 3, 5, 6, 7, 8)
```

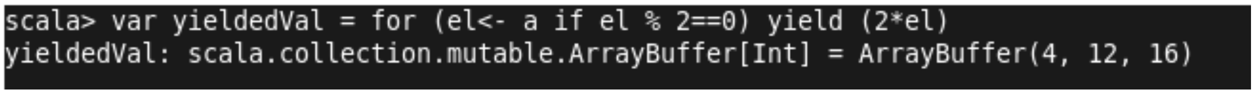
```
for (el <- a) println(el)
```



```
scala> for (el <- a) println(el)
1
2
3
5
6
7
8
```

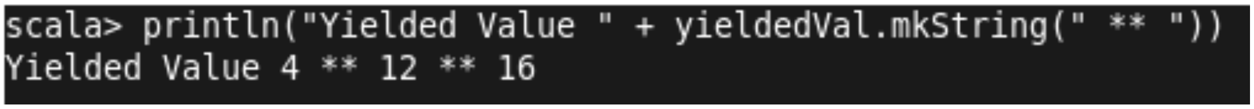
Transformation

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



```
scala> var yieldedVal = for (el<- a if el % 2==0) yield (2*el)
yieldedVal: scala.collection.mutable.ArrayBuffer[Int] = ArrayBuffer(4, 12, 16)
```

```
println("Yielded Value " + yieldedVal.mkString(" ** "))
```

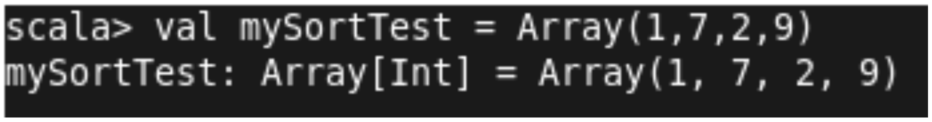


```
scala> println("Yielded Value " + yieldedVal.mkString(" ** "))
Yielded Value 4 ** 12 ** 16
```

Note : mkString will bring delimeter.

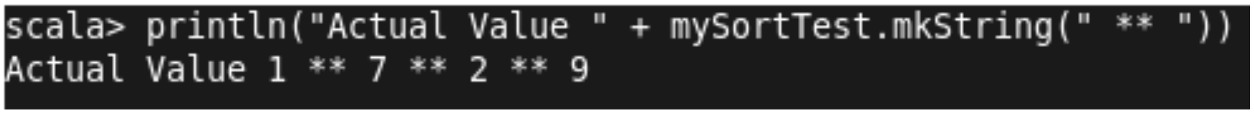
Sorting

```
val mySortTest = Array(1,7,2,9)
```



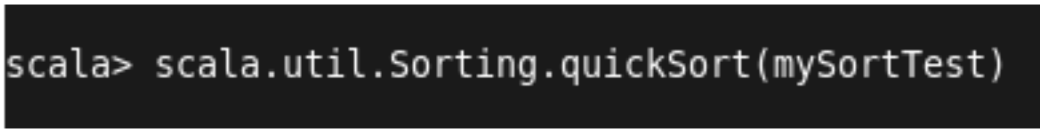
```
scala> val mySortTest = Array(1,7,2,9)
mySortTest: Array[Int] = Array(1, 7, 2, 9)
```

```
println("Actual Value " + mySortTest.mkString(" ** "))
```



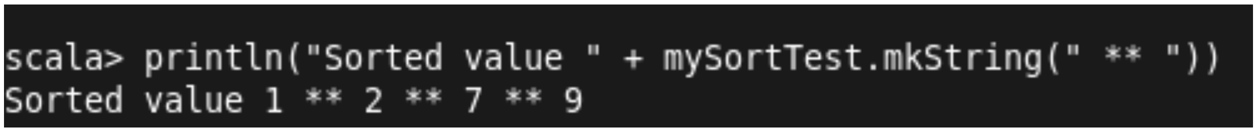
```
scala> println("Actual Value " + mySortTest.mkString(" ** "))
Actual Value 1 ** 7 ** 2 ** 9
```

```
scala.util.Sorting.quickSort(mySortTest)
```



```
scala> scala.util.Sorting.quickSort(mySortTest)
```

```
println("Sorted value " + mySortTest.mkString(" ** "))
```

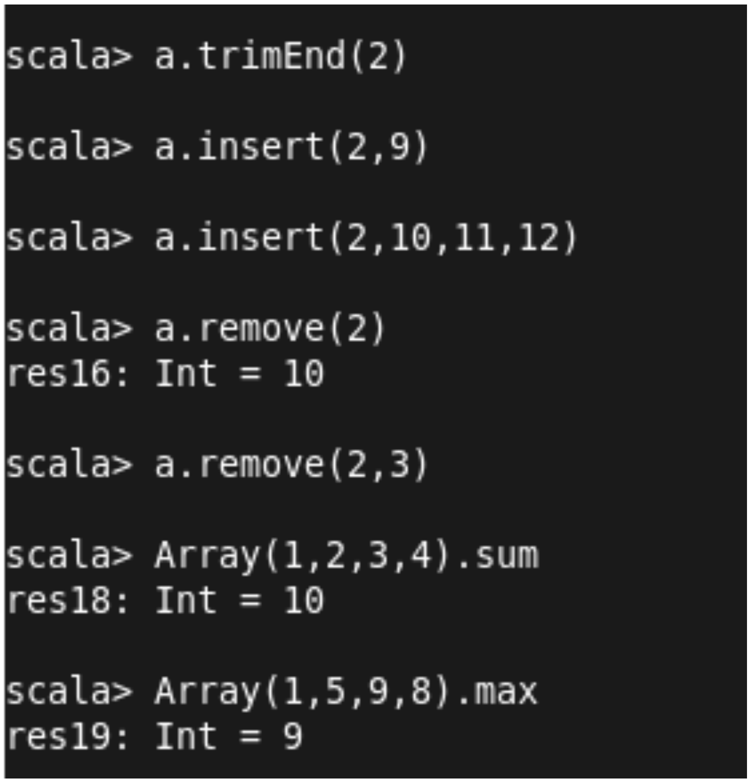


```
scala> println("Sorted value " + mySortTest.mkString(" ** "))
Sorted value 1 ** 2 ** 7 ** 9
```

Scala Collections: Arrays and ArrayBuffers

Common Operations:

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

Array(1,2,3,4).sum
Array(1,5,9,8).max
```



```
scala> a.trimEnd(2)

scala> a.insert(2,9)

scala> a.insert(2,10,11,12)

scala> a.remove(2)
res16: Int = 10

scala> a.remove(2,3)

scala> Array(1,2,3,4).sum
res18: Int = 10

scala> Array(1,5,9,8).max
res19: Int = 9
```

Scala Collections: Maps

What is key value pair

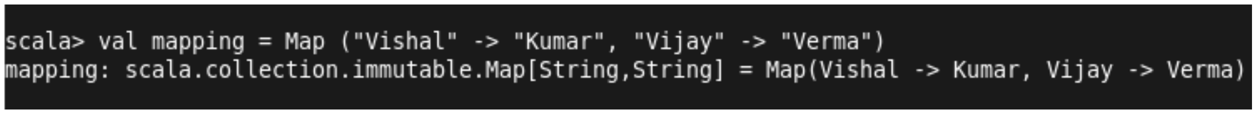
Name: Muthu
Age: 30

Key -- Value

What is a map
Ans : Unique Keys are called as maps(Values can be duplicate)

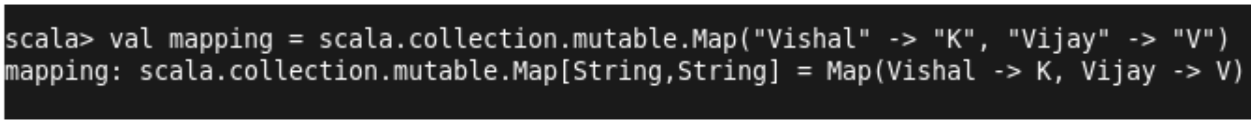
In Scala, a map is a collection of pair
A pair is a group of two values (Not necessarily of same type)

```
val mapping = Map ("Vishal" -> "Kumar", "Vijay" -> "Verma")
```



```
scala> val mapping = Map ("Vishal" -> "Kumar", "Vijay" -> "Verma")
mapping: scala.collection.immutable.Map[String,String] = Map(Vishal -> Kumar, Vijay -> Verma)
```

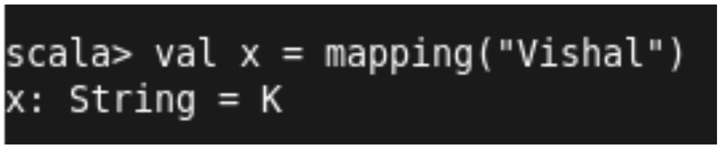
```
val mapping = scala.collection.mutable.Map("Vishal" -> "K", "Vijay" -> "V")
```



```
scala> val mapping = scala.collection.mutable.Map("Vishal" -> "K", "Vijay" -> "V")
mapping: scala.collection.mutable.Map[String,String] = Map(Vishal -> K, Vijay -> V)
```

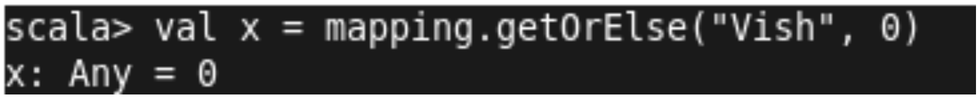
Accessing Maps

```
val x = mapping("Vishal")
```



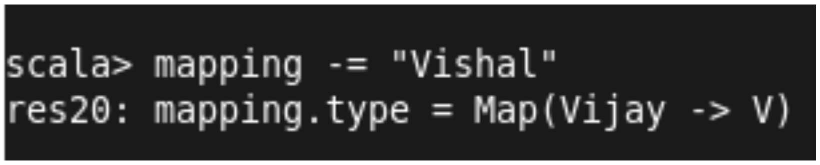
```
scala> val x = mapping("Vishal")
x: String = K
```

```
val x = mapping.getOrElse("Vish", 0)
```



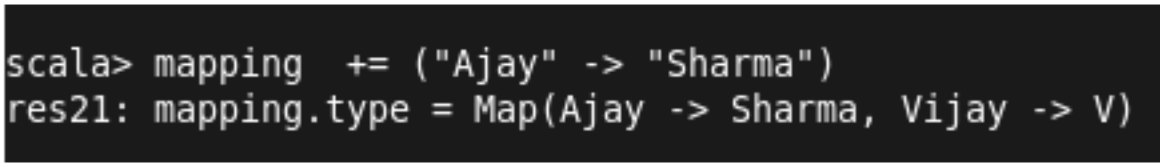
```
scala> val x = mapping.getOrElse("Vish", 0)
x: Any = 0
```

```
mapping -= "Vishal"
```



```
scala> mapping -= "Vishal"
res20: mapping.type = Map(Vijay -> V)
```

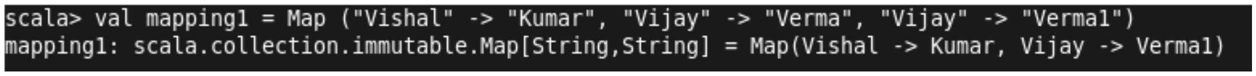
```
mapping  += ("Ajay" -> "Sharma")
```



```
scala> mapping  += ("Ajay" -> "Sharma")
res21: mapping.type = Map(Ajay -> Sharma, Vijay -> V)
```

Trying to give duplicate keys in Map

```
val mapping1 = Map ("Vishal" -> "Kumar", "Vijay" -> "Verma", "Vijay" ->
"Verma1")
```

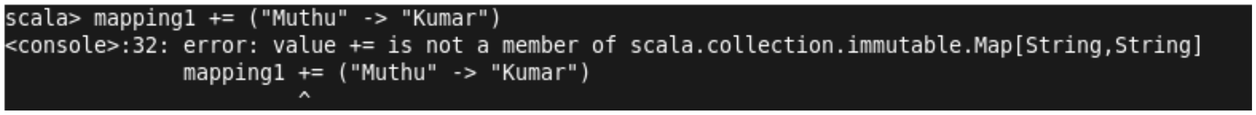


```
scala> val mapping1 = Map ("Vishal" -> "Kumar", "Vijay" -> "Verma", "Vijay" -> "Verma1")
mapping1: scala.collection.immutable.Map[String,String] = Map(Vishal -> Kumar, Vijay -> Verma1)
```

Note: Mapping took only 2 values and not three

Trying to modify an immutable map

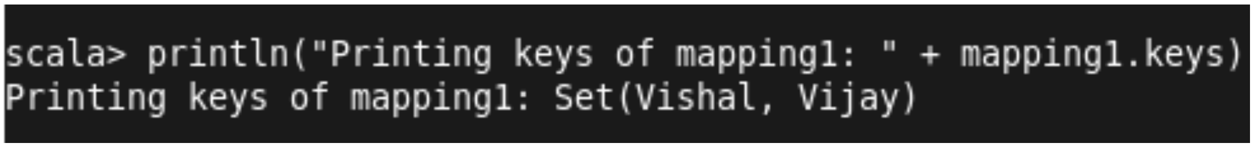
```
mapping1 += ("Muthu" -> "Kumar")
```



```
scala> mapping1 += ("Muthu" -> "Kumar")
<console>:32: error: value += is not a member of scala.collection.immutable.Map[String,String]
              mapping1 += ("Muthu" -> "Kumar")
                       ^
```

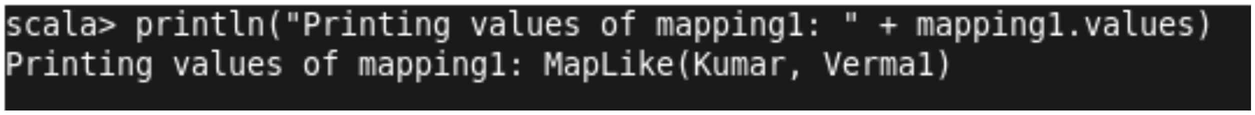
Printing

```
println("Printing keys of mapping1: " + mapping1.keys)
```



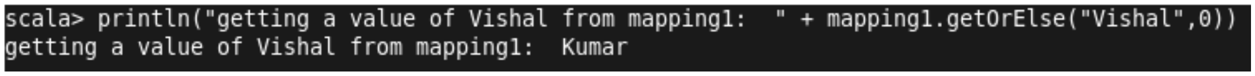
```
scala> println("Printing keys of mapping1: " + mapping1.keys)
Printing keys of mapping1: Set(Vishal, Vijay)
```

```
println("Printing values of mapping1: " + mapping1.values)
```



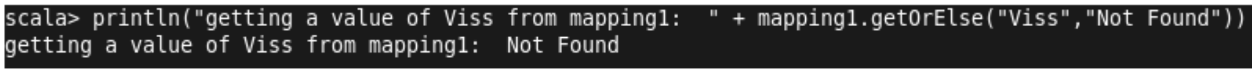
```
scala> println("Printing values of mapping1: " + mapping1.values)
Printing values of mapping1: MapLike(Kumar, Vermal)
```

```
println("getting a value of Vishal from mapping1:  " +
mapping1.getOrElse("Vishal",0))
```



```
scala> println("getting a value of Vishal from mapping1:  " + mapping1.getOrElse("Vishal",0))
getting a value of Vishal from mapping1:  Kumar
```

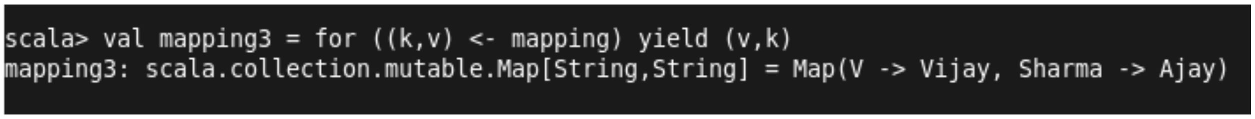
```
println("getting a value of Viss from mapping1:  " +
mapping1.getOrElse("Viss","Not Found"))
```



```
scala> println("getting a value of Viss from mapping1:  " + mapping1.getOrElse("Viss","Not Found"))
getting a value of Viss from mapping1:  Not Found
```

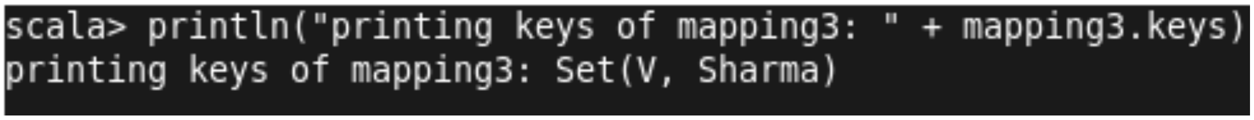
Iterating Maps

```
val mapping3 = for ((k,v) <- mapping) yield (v,k)
```



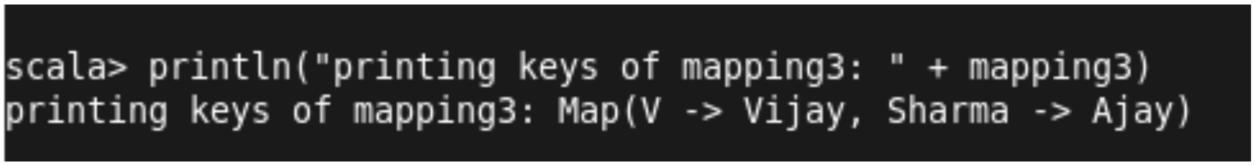
```
scala> val mapping3 = for ((k,v) <- mapping) yield (v,k)
mapping3: scala.collection.mutable.Map[String,String] = Map(V -> Vijay, Sharma -> Ajay)
```

```
println("printing keys of mapping3: " + mapping3.keys)
```



```
scala> println("printing keys of mapping3: " + mapping3.keys)
printing keys of mapping3: Set(V, Sharma)
```

```
println("printing keys of mapping3: " + mapping3)
```

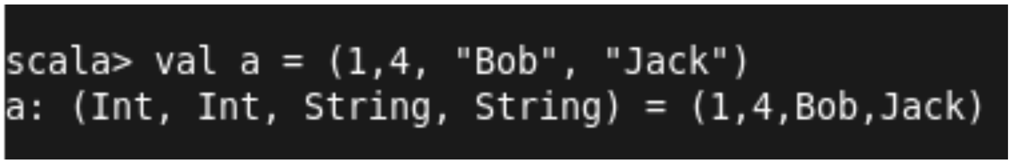

```
scala> println("printing keys of mapping3: " + mapping3)
printing keys of mapping3: Map(V -> Vijay, Sharma -> Ajay)
```

Scala Collections: Tuples

Tuple is more generalized form of pair

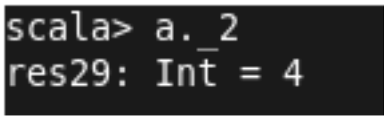
Tuple has more than two values of potentially different types

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


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

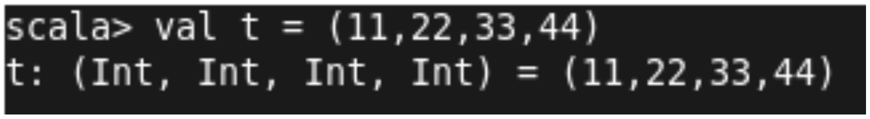
Accessing the tuple elements

```
a._2 or a _2 //returns 4
```

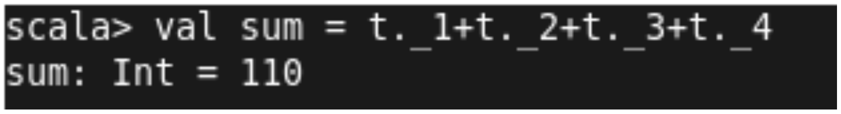

```
scala> a._2
res29: Int = 4
```

In tuples, offset starts with 1 and not 0. Tuples are typically used for functions which return more than one value.

```
val t = (11,22,33,44)
```

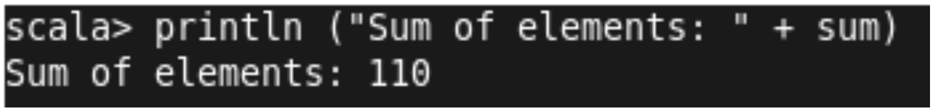

```
scala> val t = (11,22,33,44)
t: (Int, Int, Int, Int) = (11,22,33,44)
```

```
val sum = t._1+t._2+t._3+t._4
```



```
scala> val sum = t._1+t._2+t._3+t._4
sum: Int = 110
```

```
println ("Sum of elements: " + sum)
```



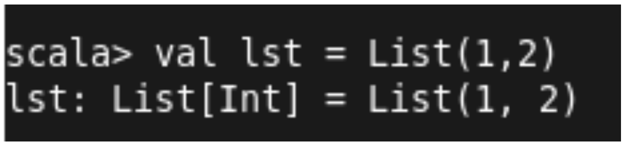
```
scala> println ("Sum of elements: " + sum)
Sum of elements: 110
```

Scala Collections: Lists

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

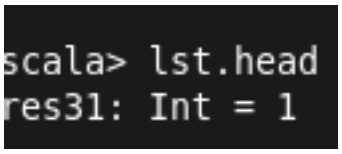
Eg:

```
val lst = List(1,2)
```



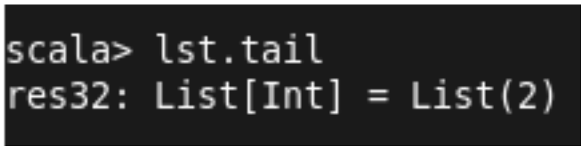
```
scala> val lst = List(1,2)
lst: List[Int] = List(1, 2)
```

```
lst.head
```



```
scala> lst.head
res31: Int = 1
```

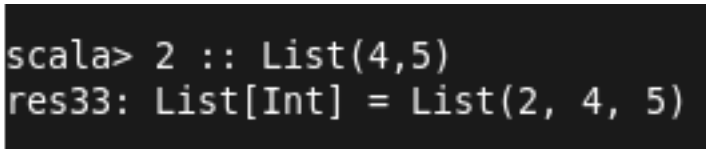
```
lst.tail
```



```
scala> lst.tail
res32: List[Int] = List(2)
```

:: operator adds a new list from given head and tail

```
2 :: List(4,5)
```



```
scala> 2 :: List(4,5)
res33: List[Int] = List(2, 4, 5)
```

Note: List is comparable to ArrayBuffer. ArrayBuffer is used where suppose a log file is read continuously and error is getting fetch and added. Whereas List should be used where it is more read intensive.

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

Eg:

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