

What is Scala?

Friday, April 21, 2017 2:42 AM

What is Scala?

edure

- 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
- General purpose means it is not specifically build for any type of operation we can do anything from it. Like web, big data etc.
 - It is a combination of both OOP and Functional Programming Styles.

Martin Odersky and his team started developing Scala in 2001



→ 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
- Scala is more statically typed (which means we have to give type of variable) than Java.
 - Scala is FP language but not pure functional programming language, because it supports mutable state(?).

→ 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
- Everything is object in Scala (like Python) so we do not have primitive types here.
 - Traits Supported (like PHP) which are used to combine some useful functions, for better

reusability.

- » 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**
 - In Functional programming Functions acts as Object, so they can be passed to a function and returned from a function as well.
 - The functions can be assigned to a variable also.
 - **FUNCTIONS ARE ALSO OBJECTS IN SCALA.**
- All basic things in scala are in **scala** package(like java.lang package);

Why Scala?

Friday, April 21, 2017 2:49 AM

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



Scala in Other Frameworks

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

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

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

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

Neo4j - Graph Database

The Neo4j Scala wrapper library allows you the Neo4j open source graph database through a domain-specific simplified language. It is written in Scala and is intended to be used in other Scala projects.

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 provides REPL(read,execute,print,loop) shell like other scripting language, even though it is not scripting language, and it take our code and put inside a program then compile and execute it on JVM behind the scenes.

Variables in Scala

Friday, April 21, 2017 2:59 AM

Data Types

- All 8 primitive types as JAVA Wrappers are Present in Scala.

Scala has all the same data types as Java, with the same memory footprint and precision. Following is the table giving details about all the data types available in Scala:

Data Type	Description
Byte	8 bit signed value. Range from -128 to 127
Short	16 bit signed value. Range -32768 to 32767
Int	32 bit signed value. Range -2147483648 to 2147483647
Long	64 bit signed value. -9223372036854775808 to 9223372036854775807
Float	32 bit IEEE 754 single-precision float
Double	64 bit IEEE 754 double-precision float
Char	16 bit unsigned Unicode character. Range from U+0000 to U+FFFF
String	A sequence of Chars
Boolean	Either the literal true or the literal false

- There are some extra DataTypes available, which are following.

Unit	Corresponds to no value
Null	null or empty reference
Nothing	The subtype of every other type; includes no values
Any	The supertype of any type; any object is of type Any
AnyRef	The supertype of any reference type

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.

Declaring Variables

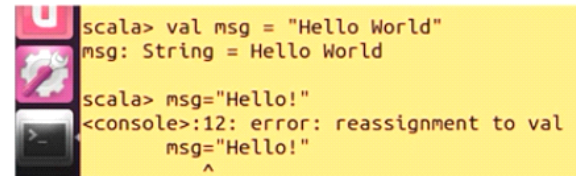
→ **Scala allows one to decide whether a variable is immutable or mutable**

→ **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!"
```



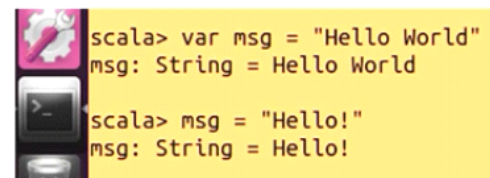
```
scala> val msg = "Hello World"
msg: String = Hello World

scala> msg="Hello!"
<console>:12: 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> var msg = "Hello World"
msg: String = Hello World

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

Type Inference

- Scala provides Type Inference so it can find out the type of variable if value is assigned on declaration.

```
scala> var name="ashish"
name: String = ashish

scala> var num=10
num: Int = 10

scala> var dnum=10.23
dnum: Double = 10.23

scala> _
```

- We can see by default decimal becomes "Double" if want "Float" use 'f' literal.
- Variable as needed to be initialised to be defined in Scala.

```
scala> var adfa:String;
<console>:11: error: only classes can have declared but undefined members
(Note that variables need to be initialized to be defined)
    var adfa:String;
      ^
```

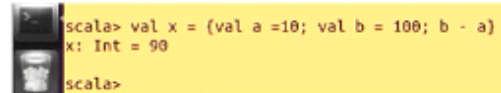
Assigning Block Expression

→ 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
```



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



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

- Here a and b will not be accessible outside as they are local.
- We can assign a code block to a variable but the value will be the result of the last expression in that block.
- Line Delimiter(;) is not required in Scala also (like python, bash) if only one expression is on 1 line, so this will give the same output as above.

```
scala> val x = {  
  | val a = 100  
  | val b = 200  
  | a + b  
  | }  
x: Int = 300
```

Lazy Values

Friday, April 21, 2017 3:39 AM

Lazy Values

- Lazy Values are a very important concept in Scala, and that is a very important reason why Scala is used for Big Data Framework like Spark.
- Only Constant Variable i.e. **val** can be made lazy, this value does not get loaded in memory as soon as it is declared, but it gets allocated memory when we make first use of it
- USE-
 - Access
 - Any computation.
- This Approach is very useful when data is very large.

→ Lazy values are very useful for delaying costly initialization instructions

- Lazy instructions are stored in memory and executed only when first use of that variable happens.
- Lazy Values can be used with any kind of data like variables, files etc. these will be loaded when needed.

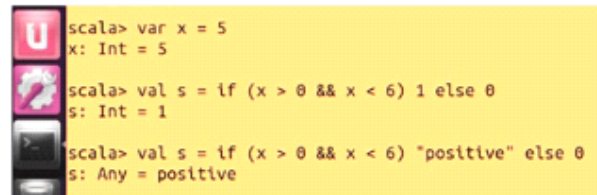
Control Structures(If else)

Friday, April 21, 2017 3:46 AM

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
```



```
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

USING CONTROL STRUCTURES

- METHOD 1 - We can use If Structures like Java/C++ like in Scala

```
if( x < 20 ){
    println("This is if statement");
}
```

- METHOD 2 - in Scala If Structures can be used to return values, like we did with ternary in JAVA>

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

scala> var ans=if(x<10) "less"; else "more";
ans: String = more

scala> _
```

Loop

Friday, April 21, 2017 3:56 AM

While and do-while loop

- While and do-while loops are same as JAVA/C++ in Scala
- Example - while

```
// Local variable declaration:
var a = 10;
    // while loop execution
while( a < 20 ){
    println( "Value of a: " + a );
    a = a + 1;
}
```

- Example - do-while

```
// Local variable declaration:
var a = 10;

    // do loop execution
do{
    println( "Value of a: " + a );
    a = a + 1;
}while( a < 20 )
```

For loop

- For loop is different in scala from java/c++.

Syntax: for loop with ranges

The simplest syntax of for loop with ranges in Scala is:

```
for( var x <- Range ){
    statement(s);
}
```

Example- for loop

```
var a = 0;

// for loop execution with a range
for( a <- 1 to 10 ){
    println( "Value of a: " + a );
}
```

OUTPUT- from 1 to 10

- By default increment is **+1** , we can modify it using **by** keyword

Example -

```
scala> for (i <- 5 to 1 by -1) println(i)
5
4
3
2
1
```

- We can use **until** also in place of **to**

```
for( a <- 1 until 10){
    println( "Value of a: " + a );
}
```

OUTPUT- from 1 to 9

You can use multiple ranges separated by semicolon (;) within **for loop**

```
var a = 0;
var b = 0;
// for loop execution with a range
for( a <- 1 to 3; b <- 1 to 3){
    println( "Value of a: " + a );
    println( "Value of b: " + b );
}
```

- We can substitute variable inside a string using **\$ sign** , with 's' which tells scala to interpret the variable(using \$) as string(using s)

```
scala> for (i <- 1 to 5; j <- 1 to 4) println(s"$i, $j")
(1, 1)
(1, 2)
(1, 3)
(1, 4)
(2, 1)
(2, 2)
(2, 3)
(2, 4)
(3, 1)
(3, 2)
(3, 3)
(3, 4)
(4, 1)
(4, 2)
(4, 3)
(4, 4)
(5, 1)
(5, 2)
(5, 3)
(5, 4)
```

- Here we can see , `println(s"$i,$j")` , here we have told scala to interpret variable **I and J** as string(by s).

Syntax: for Loop with Collections

The following syntax for loop with collections.

```
for( var x <- List ){
    statement(s);
}
```

Example -

```
var a = 0;
val numList = List(1,2,3,4,5,6);

// for loop execution with a collection
for( a <- numList ){
    println( "Value of a: " + a );
}
```

Syntax: for loop with Filters

```

for( var x <- List
      if condition1; if condition2...
    ){
    statement(s);
  }

```

- Just remember we have to add semicolon(;) after each 'if test' and 'variable list'(to separate them)

Example - 1

```

var a = 0;

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

// for loop execution with multiple filters
for( a <- numList
      if a != 3; if a < 8 ){
  println( "Value of a: " + a );
}

```

Example - 2

```

scala> for (i <- 1 to 5; j <- 1 to 4 if i == j)
        | println(s"($i, $j)")
(1, 1)
(2, 2)
(3, 3)
(4, 4)

```

- Scala goes for all combinations (n^2) and find what is appropriate.

Example - 3

- We can check any where, we just need to place "semicolon" as we are applying multiple expressions.

```

scala> for( i <- 1 to 5; if i%2==0;j<- 6 to 10; if j%2!=0)
        | println(s"$i,$j")
2,7
2,9
4,7
4,9

```

Syntax: for loop with yield

You can store return values from a "for" loop in a variable or can return through a function. To do so, you prefix the body of the 'for' expression by the keyword **yield**. The following is the syntax.

```

var retVal = for{ var x <- List
                  if condition1; if condition2...
                }yield x

```

NOTE - an interesting thing to note here it that scala is very flexible and allows us to use braces({}) or paranthesis interchangeably at many places but paranthesis provides more compiler verification than braces.

Example -

```
object myobj {  
  def main(args: Array[String]): Unit = {  
  
    val y=for(i ← 1 to 10)  
    | yield(i)  
  
    val z=for{i ← 1 to 10}  
    | yield(i)  
  
    println(y);println(z);  
  }  
}
```

Output-


```
"C:\Program Files\Java\jdk1.8.0_71\bin\java" ...  
Vector(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)  
Vector(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

Foreach

- Foreach function is available to all collections in Scala,

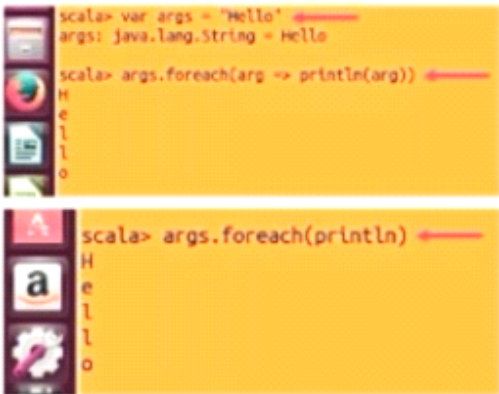
Syntax - <collection>.foreach(<function>)

Scala: Foreach Loop



→ Looping with foreach:

```
var args = "Hello"  
args.foreach(arg => println(arg))  
args.foreach(println)
```



```
scala> var args = "Hello"  
args: java.lang.String = Hello  
scala> args.foreach(arg => println(arg))  
H  
e  
l  
l  
o  
scala> args.foreach(println)  
H  
e  
l  
l  
o
```

Slide 26

www.edureka.co/apache-spark-scala-training

EXAMPLE -

- As we can see we can pass a function to foreach which will be executed on each element of collection. There is another method in which we use lambda style(element is passed to 'ch' and then we apply function)

```
scala> var name="ashish"  
name: String = ashish  
  
scala> name.foreach(println)  
a  
s  
h  
i  
s  
h  
  
scala> name.foreach(ch=>println(ch))  
a  
s  
h  
i  
s  
h  
  
scala> _
```

Functions

Sunday, April 23, 2017 8:26 PM

A function definition can appear anywhere in a source file and Scala permits nested function definitions, that is, function definitions inside other function definitions. Most important point to note is that Scala function's name can have characters like `+`, `++`, `~`, `&`, `-`, `--`, `\`, `/`, `:`, etc.

Function Declarations

A Scala function declaration has the following form:

```
def functionName ([list of parameters]) : [return type]
```

Methods are implicitly declared *abstract* if you don't use the equals sign and the method body. The enclosing type is then itself abstract.

Function Definitions

A Scala function definition has the following form:

```
def functionName ([list of parameters]) : [return type] = {  
    function body  
    return [expr]  
}
```

- As we can see, we provide function definition using **equal sign (=)**. So if "=" is not given then scala automatically declare it as **abstract**.
- The syntax is same as new C++ function syntax, using **auto**, where we also give return type after full declaration.

```
auto main()->int  
,
```

Example -

```
object add{  
    def addInt( a:Int, b:Int ) : Int = {  
  
        var sum:Int = 0  
        sum = a + b  
  
        return sum  
    }  
}
```


Returning Nothing/Void

- If we do not want to return anything we can remove return type at all, but it is good practice to return "**Unit**", which means nothing.(see in variables in scala page)

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.

Syntax

Here is the syntax:

```
object Hello{
  def printMe( ) : Unit = {
    println("Hello, Scala!")
  }
}
```

Returning Value

- If we are returning any type from a function and we do not used **return** statement, then by default last value of last statement of function is returned.

```
def areaRect(l: Float, b: Float): Float = {
  l * b
}
```

Function with Variable Arguments

Scala allows you to indicate that the last parameter to a function may be repeated.

```
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;
    }
  }
}
```

- Here we can pass any number of "strings" they all will go in argument "**args**" which will be of type **WrappedArray**.

Function Default Parameter Values

Scala lets you specify default values for function parameters.

Example -

```
object myobj {  
  def main(args: Array[String]): Unit = {  
    print_n_times("ashish",5);//will print 5 times  
    println();  
    print_n_times("patel");//will print 1 time  
  }  
  
  def print_n_times(str:String,num:Integer=1):Unit={  
    for(i← 1 to num){  
      print(str+" ");  
    }  
  }  
}
```

Output-

```
"C:\Program Files\Java\jdk1.8.0_71\bin\java" ...  
ashish ashish ashish ashish ashish  
patel  
Process finished with exit code 0
```

- Like Python, Scala also allows us to use argument name while calling. So, there will be no confusion and we need not to remember the order.

```
object myobj {  
  def main(args: Array[String]): Unit = {  
    print_n_times(num = 5,str="ashish");//will print 5 times  
  }  
  
  def print_n_times(str:String,num:Integer=1):Unit={  
    for(i← 1 to num){  
      print(str+" ");  
    }  
  }  
}
```

Generic Programming

Friday, April 28, 2017 5:35 PM

- Parametric polymorphism means we can use many parameter types in same thing, which is also called generics.

Parametric polymorphism also exists in some object-oriented languages and sometimes it is referred to as *generic programming*. For example, Java added parametric polymorphism through generics in version 5.

Here is an example of a method that uses parametric polymorphism:

```
def map[A, B] (xs: List[A]) (f: A => B): List[B] = xs map f
```

The map method takes a `List[A]` and a function from A to B as input and returns a `List[B]`. As you can see you don't mention concrete types but rather use type parameters—hence parametric polymorphism—to abstract over types. For example you can use that method to transform a `List[Int]` into a `List[String]` or, analogously, a `List[String]` into a `List[Int]`:

```
val stringList: List[String] = map(List(1, 2, 3))(_.toString)
```

```
val intList: List[Int] = map(List("1", "2", "3"))(_.toInt)
```

Roughly speaking, whenever your methods have type parameters, you're using parametric polymorphism.

Array

Sunday, April 23, 2017

9:56 PM

Scala Collections: Array

→Fixed Length Arrays:

Examples:

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

- In the first two example, array will be initialised with their default values(like java).
- We can use **Array()** function to create an array of given type.
- Array is present inside **scala** package.

```
val numbers = Array(1, 2, 3, 4)
val first = numbers(0) // read the first element
numbers(3) = 100 // replace the 4th array element with 100
val biggerNumbers = numbers.map(_ * 2) // multiply all numbers by two
```

Val Arrays

- Look at example below and, we can see that although **numbers** is constant(val) we are able to modify numbers(0) by 10.

```
object myobj {
  def main(args: Array[String]): Unit = {
    val numbers=Array(1,2,3,4,5);
    numbers(0)=10;
    numbers.foreach(println);
  }
}
```

- Explanation - like java , here **numbers** reference became constant, so we can not do something like this now,
Numbers=Array(10,20,30);
because refrence is constant, but we can change the values.

List

Friday, April 28, 2017 6:15 PM

Scala Lists

Scala Lists are quite similar to arrays which means, all the elements of a list have the same type but there are two important differences. First, lists are immutable, which means elements of a list cannot be changed by assignment. Second, lists represent a linked list whereas arrays are flat.

The type of a list that has elements of type T is written as **List[T]**.

Try the following example, here are few lists defined for various data types.

```
// List of Strings
val fruit: List[String] = List("apples", "oranges", "pears")

// List of Integers
val nums: List[Int] = List(1, 2, 3, 4)

// Empty List.
val empty: List[Nothing] = List()

// Two dimensional list
val dim: List[List[Int]] =
  List(
    List(1, 0, 0),
    List(0, 1, 0),
    List(0, 0, 1)
  )
```

All lists can be defined using two fundamental building blocks, a tail **Nil** and **::**, which is pronounced **cons**. Nil also represents the empty list. All the above lists can be defined as follows.

```
// List of Strings
val fruit = "apples" :: ("oranges" :: ("pears" :: Nil))

// List of Integers
val nums = 1 :: (2 :: (3 :: (4 :: Nil)))

// Empty List.
val empty = Nil

// Two dimensional list
val dim = (1 :: (0 :: (0 :: Nil))) ::
  (0 :: (1 :: (0 :: Nil))) ::
  (0 :: (0 :: (1 :: Nil))) :: Nil
```

- Use **::** for separating elements and Nil to tell end.

Basic Operations on Lists

All operations on lists can be expressed in terms of the following three methods.

Methods	Description
Head	This method returns the first element of a list.
Tail	This method returns a list consisting of all elements except the first.
IsEmpty	This method returns true if the list is empty otherwise false.

The following example shows how to use the above methods.

```
object Demo {  
  def main(args: Array[String]) {  
    val fruit = "apples" :: ("oranges" :: ("pears" :: Nil))
```

```
    val nums = Nil  
  
    println( "Head of fruit : " + fruit.head )  
    println( "Tail of fruit : " + fruit.tail )  
    println( "Check if fruit is empty : " + fruit.isEmpty )  
    println( "Check if nums is empty : " + nums.isEmpty )  
  }  
}
```

Output

```
Head of fruit : apples  
Tail of fruit : List(oranges, pears)  
Check if fruit is empty : false  
Check if nums is empty : true
```

Concatenating Lists

You can use either `:::` operator or `List.:::()` method or `List.concat()` method to add two or more lists. Please find the following example given below:

```
object Demo {  
  def main(args: Array[String]) {  
    val fruit1 = "apples" :: ("oranges" :: ("pears" :: Nil))  
    val fruit2 = "mangoes" :: ("banana" :: Nil)  
  
    // use two or more lists with ::: operator  
    var fruit = fruit1 ::: fruit2  
    println( "fruit1 ::: fruit2 : " + fruit )
```

```
// use two lists with Set.:::( ) method
fruit = fruit1.:::(fruit2)
println( "fruit1.:::(fruit2) : " + fruit )

// pass two or more lists as arguments
fruit = List.concat(fruit1, fruit2)
println( "List.concat(fruit1, fruit2) : " + fruit )

}
}
```

Creating Uniform Lists

You can use **List.fill()** method creates a list consisting of zero or more copies of the same element. Try the following example program.

```
object Demo {
  def main(args: Array[String]) {
    val fruit = List.fill(3)("apples") // Repeats apples three times.
    println( "fruit : " + fruit )

    val num = List.fill(10)(2)          // Repeats 2, 10 times.
    println( "num : " + num )
  }
}
```

Output

```
fruit : List(apples, apples, apples)
num : List(2, 2, 2, 2, 2, 2, 2, 2, 2, 2)
```

Reverse List Order

You can use **List.reverse** method to reverse all elements of the list. The following example shows the usage.

```
object Demo {  
  def main(args: Array[String]) {  
    val fruit = "apples" :: ("oranges" :: ("pears" :: Nil))  
    println( "Before reverse fruit : " + fruit )  
  
    println( "After reverse fruit : " + fruit.reverse )  
  }  
}
```

Save the above program in **Demo.scala**. The following commands are used to compile and execute this program.

```
\>scalac Demo.scala  
\>scala Demo
```

Output:

```
Before reverse fruit : List(apples, oranges, pears)  
After reverse fruit : List(pears, oranges, apples)
```

- We can use `toList()` method on any collection to convert it to list

```
45 | def toList: List[A]  
   | Returns a list containing all elements of this immutable set.
```

```
scala> val l = (1 to 20).toList  
l: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20)
```


ArrayBuffer

Sunday, April 23, 2017

10:49 PM

- Have to import it.

Scala Collections: ArrayBuffer

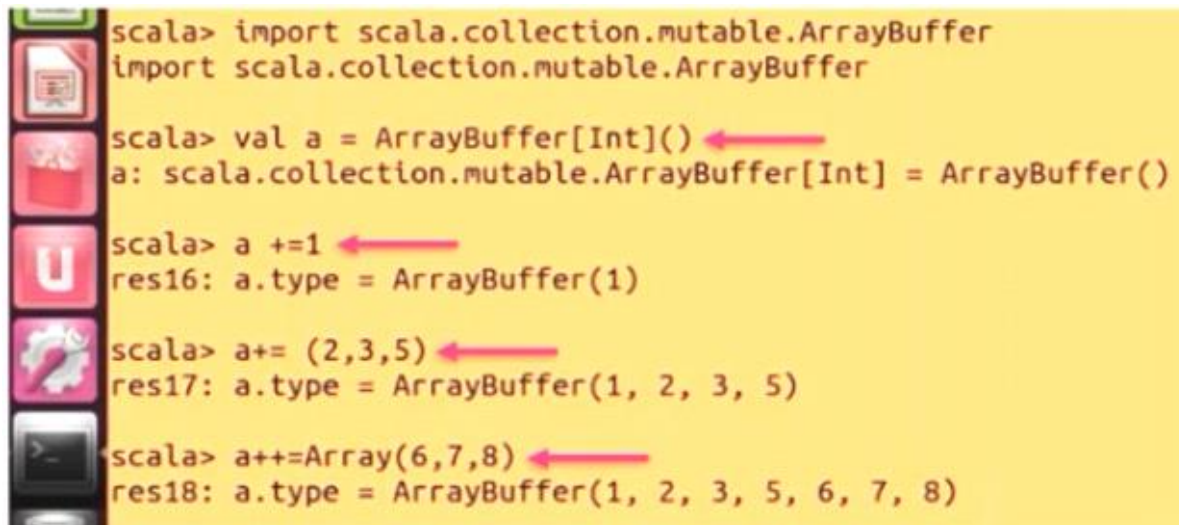
→ Variable Length Arrays (Array Buffers)

→ Similar to Java ArrayLists

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

- ArrayBuffer has variable length and can grow.
- Here we will start with no length (means 0) at the beginning (can give like Array but the size will remain 0), and then we are concatenating more and more elements.
- To append elements use **(+=)** and if we want to append collection use **(++=)** operator.

Example -



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

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

scala> a += 1
res16: a.type = ArrayBuffer(1)

scala> a += (2,3,5)
res17: a.type = ArrayBuffer(1, 2, 3, 5)

scala> a ++= Array(6,7,8)
res18: a.type = ArrayBuffer(1, 2, 3, 5, 6, 7, 8)
```

- Similarly we can remove elements from an array using **(-=) {for elements}** and **(--=) {for collections}** operators. This operator will search for given elements in array if they found then delete otherwise do nothing.

```
scala> arr -= Array(700, 800)
res17: arr.type = ArrayBuffer(100, 200, 300, 400, 500, 600)
```

Operations

Sunday, April 23, 2017

11:54 PM

Scala Collections: Arrays and ArrayBuffers

→Common Operations:

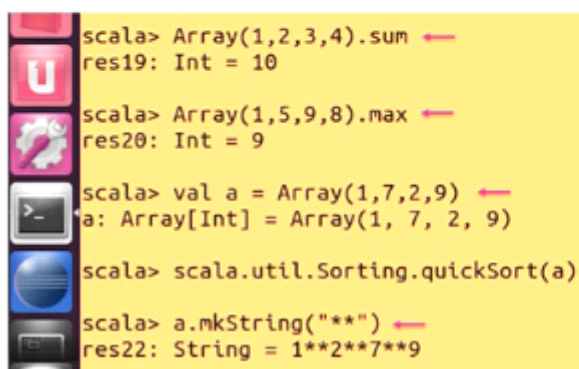
```
a.trimEnd(2) //Removes last 2 elements  
a.insert(2, 9) // Adds element at 2nd index  
a.insert(2,10,11,12) //Adds a list  
a.remove(2) //Removes an element  
a.remove(2,3) //Removes three elements from index 2
```

→Traversing and Transformation:

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

→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> Array(1,2,3,4).sum  
res19: Int = 10  
  
scala> Array(1,5,9,8).max  
res20: Int = 9  
  
scala> val a = Array(1,7,2,9)  
a: Array[Int] = Array(1, 7, 2, 9)  
  
scala> scala.util.Sorting.quickSort(a)  
  
scala> a.mkString(" ** ")  
res22: String = 1**2**7**9
```

- Similarly we have **min** as well.

Maps

Monday, April 24, 2017 12:49 AM

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

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

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

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

- Maps are hash tables, all keys need to be unique.
- Types of Map -
 - Immutable - inside **scala** package.(first line)
 - Mutable - inside **scala.collection.mutable.Map** (second line)
- Actually every collection in scala comes in two flavours
 - Mutable
 - Immutable

Mutable vs Immutable

- Map are immutable(in scala package), so if we try to do some change we get error.

Example-

```
object myobj {

  def main(args: Array[String]): Unit = {
    val x=Map(1->"ashish",2->"patel");
    for(i<-x)
      println(i)
    x-=2; //telling to remove record having key=2, give error
  }
}
```

Output-

```
Error:(12, 6) value -= is not a member of scala.collection.immutable.Map[Int,String]
x-=2; //telling to remove record having key=2, give erro
```

- If we do same with mutable map (inside **scala.collection.mutable.map** package), we do not get error.

```
object myobj {
  def main(args: Array[String]): Unit = {
    val x=scala.collection.mutable.Map(1→"ashish",2→"patel");
    for(i←x)
      println(i)
    x-=2; //telling to remove record having key=2, give error
    println("\nafter deletion");
    for(i←x)
      println(i)
  }
}
```

Output-

```
"C:\Program Files\Java\jdk1.8.0_71\bin\java" ...
(2,patel)
(1,ashish)

after deletion
(1,ashish)

Process finished with exit code 0
```

Accessing Elements

- We can use array like syntax, here also to access elements, here we use key in place of index.

```
def main(args: Array[String]): Unit = {
  val x=scala.collection.mutable.Map(1→"ashish",2→"patel");

  println(x(1));
  println(x(2));
}
```

- If we gave invalid key then we get lots of errors, so we can use following functions.
 - get(key) -> returns None
 - getOrElse(key,default) -> return 'default' value

Example -

```
object myobj {
  def main(args: Array[String]): Unit = {
    val x=scala.collection.mutable.Map(1→"ashish",2→"patel");
    println(x(1))
    println(x.get(2))
    println(x.getOrElse(20,-1))
  }
}
```

Output-

```
"C:\Program Files\Java\jdk1.8.0_71\bin\java" ...
ashish
--
```

```
"C:\Program Files\Java\jdk1.8.0_71\bin\java" ...  
ashish  
None  
-1
```

Adding and Deleting Elements

- It is same as Array/ArrayBuffer (using -= or +=) operators.

```
m -= 1  
m.type = Map(2 -> Jenny)  
  
m += (1 -> "Jenny")  
m.type = Map(2 -> Jenny, 1 -> Jenny)
```

Tuples

Monday, April 24, 2017 1:33 AM

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")
```

→ Accessing the tuple elements:

a._2 or a._2//Returns 4

```
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
```

→ 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)
```

- Tuples are used to represent heterogeneous data like data of a student, which ofcourse contain "string" and number together.
- Used for getting data from database and then using it.

Example -

```
scala> val t = (100, "John", "France")
t: (Int, String, String) = (100,John,France)

scala> t._1
res51: Int = 100

scala> t._2
res52: String = John

scala> t._3
res53: String = France

scala> t._1 = 200
<console>:10: error: reassignment to val
    t._1 = 200
      ^
```

- Tuple's value can't be changed (same as python), reference change depend upon type(val or var)

What is OOP?

Tuesday, April 25, 2017 11:33 PM

4 Pillars of OOP

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

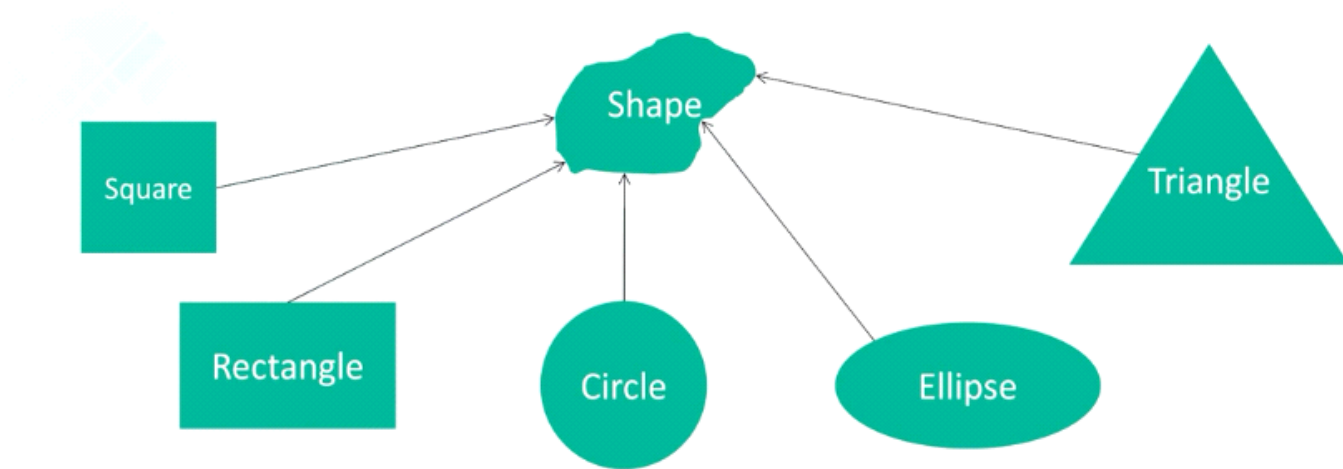
Encapsulation

- In real world we don't always reveal all our details to everyone
- What we tell others about ourselves depends on various factors
- In other words, we **encapsulate** details and only reveal it on need basis

Abstraction

- Any entity in your program that has a clear defined responsibility

Examples: Email, Savings Account, Department, Employee



Shape has properties like **name**, **area**, **location in space** which Square, Circle, etc. will inherit

Polymorphism

- Where the behavior is driven by the context



Creating and Using Class

Tuesday, April 25, 2017 11:39 PM

Creating Class

- Creating class is pretty same as JAVA, and we can simply call class methods like java.

Example -

```
class MyClass{
  def testFun():Unit={
    println(" This is test function")
  }
}

object myobj{
  def main(args:Array[String]):Unit={
    val obj=new MyClass();
    obj.testFun();
  }
}
```

Creating Constructor

- Creating constructor is different from java in Scala, here we define it with taking class_name as constructor; //class_name ko hi constructor maan lo.
- There two type of constructor
 - Primary constructor -
 - Auxillary constructor - as more than one constructor possible make other auxillary.

Creating Primary Constructor

```
class MyClass (val name: String, val age: Int) {
  def test(): Unit = {
    println(name + " " + age)
  }
}

object TestMyClass {
  def main(args: Array[String]): Unit = {
    val m = new MyClass("Test", 18)
    m.test()
    println(m.name + " " + m.age)
  }
}
```

data member

NOTE -

- Here Important thing to note that, 'name' and 'age' are passed to primary constructors automatically become **data member** of the class 'MyClass'.
- Since these are public variable we can access them directly

Making Variable private and protected

```
-class MyClass (private val name: String, private val age: Int) {  
  • |  
  - def test(): Unit = {  
    println(name + " " + age)  
  }  
}  
  
-object TestMyClass {  
  - def main(args: Array[String]): Unit = {  
    val m = new MyClass("Test", 18)  
    m.test()  
  }  
}
```

Making Auxiliary constructor

- Generally, we define 'primary constructor' which takes all data members, in this way we create all data member to that class.
And then we overload the constructor using auxiliary constructor and inside that auxiliary constructor we call the primary constructor accordingly.
- The concept is same like constructor chaining in JAVA.

```
-class MyClass (private val name: String, private val age: Int) /* primary constructor */ {  
  // auxillary constructor  
  - def this(name: String) {  
    this(name, 100) // primary constructor  
  }  
  
- def test(): Unit = {  
  println(name + " " + age)  
}  
}  
  
-object TestMyClass {  
- def main(args: Array[String]): Unit = {  
  val m = new MyClass("Test", 18)  
  m.test()  
  
  val m1 = new MyClass("Test1")  
  m1.test()  
}  
}
```

OUTPUT -

```
<terminated> TestMyClass$ (1) [Scala Application] /usr/lib/jvm/jdk1.7.0_67/bin/java (Nov 15, 2016,  
Test 18  
Test1 100
```

- Here we can see we have created a constructor which takes only "string name" and then pass the values to primary constructor.

Class Code Execution

- As we know in Scala, class acts like constructor, and we can define nested function so we are using this concept of primary constructor, which acts like a function.
- Now question arises when the code written inside class definition is run.(as we can write simple code inside class) answer class code is run when a object is made(because it acts like constructor), obviously function will not execute unless you call them.

```
class MyClass (private val name: String, private val age: Int) /* primary constructor */ {
  println("some message...")
  // auxillary constructor
  def this(name: String) {
    this(name, 100) // primary constructor
  }

  def test(): Unit = {
    println(name + " " + age)
  }
}

object TestMyClass {
  def main(args: Array[String]): Unit = {

    val m = new MyClass("Test", 18)
    m.test()

    val m1 = new MyClass("Test1")
    m1.test()
  }
}
```

OUTPUT -

```
<terminated> TestMyClass$ (1) [Scala Application] /usr/lib/jvm/jdk1.7.0_67/
some message...
Test 18
some message...
Test1 100
```

- As we can see "some message" executed each time a object is made.

Singleton(Object)

Wednesday, April 26, 2017 2:06 AM

Singleton Objects

Scala is more object-oriented than Java because in Scala, we cannot have static members. Instead, Scala has **singleton objects**. A singleton is a class that can have only one instance, i.e., Object. You create singleton using the keyword **object** instead of class keyword. Since you can't instantiate a singleton object, you can't pass parameters to the primary constructor. You already have seen all the examples using singleton objects where you called Scala's main method.

- 'Singleton Object' are like **static class** in java. Everything inside it will be static so we need not to create object of it.
we can call everything just using its name. it is different from interface that we can define methods inside it.

```
object MyObject {  
  def test(): Unit = {  
    println("test message")  
  }  
}  
  
object TestMyClass {  
  def main(args: Array[String]): Unit = {  
    MyObject.test()  
  }  
}
```

OUTPUT -

```
<terminated> TestMyClass$ (1) [Scala Application]  
test message
```

Singletons Use-cases

- Singletons can be used in Scala as:
 - When a singleton instance is required for coordinating a service
 - When a single immutable instance could be shared for efficiency purposes
 - When an immutable instance is required for utility functions or constants

Companion Objects

Wednesday, April 26, 2017 2:13 AM

Companion Object

A companion object is an object with the same name as a class or trait and is defined in the same source file as the associated file or trait. A companion object differs from other objects as it has access rights to the class/trait that other objects do not. In particular it can access methods and fields that are private in the class/trait.

An analog to a companion object in Java is having a class with static methods. In Scala you would move the static methods to a Companion object.

- There are many cases when we need both **static** and **non-static** method, in java or c++ we make some method static using **static** keyword, but in scala we do not have **static** keyword and we can't make anything static. So, we make a **object** with same name as 'class' and put all function which we wanted to that **object**, which will make them automatically.
- Companion (object and class) can access even private things of each other, which is like friend function in C++.

[Pause] Companion Objects

edu

- In many programming languages, we typically have both instance methods and static methods in same class
- In Scala, it is achieved by Companion Object of same name as of class

Example:

```
Class Account {  
    val id = Account.newNum()  
    private var bal = 0.0  
    ....  
}  
object Account {  
    private var lastNum = 0;  
    private def newNum() = { lastNum +=1; lastNum }  
}
```

- The class and its companion objects need to be in same source file
- The class and its companion object can access each other's private features
- The companion object of the class is accessible, but NOT in scope

Example - 1

```

class MyClass {
    def test(): Unit = { MyClass.printMsg() }
}

object MyClass {
    def printMsg(): Unit = { println("test message") }
}

object Main {
    def main(args: Array[String]): Unit = {
        val m = new MyClass()
        m.test()
    }
}

```

Example -2 Counting number of instances?

```

class Test {
    Test.instanceCnt += 1
}

object Test {
    var instanceCnt: Int = 0

    def printInstanceCnt(): Unit = { println(instanceCnt) }
}

object Main {
    def main(args: Array[String]): Unit = {
        for (i <- 1 to 10)
            new Test()

        Test.printInstanceCnt()
    }
}

```

- Here we have created **instanceCnt** in object so that it become static.

Case Classes

Friday, April 28, 2017 2:03 AM

- Immutability is provided by case class, case object and tuples in Scala
- And ADT (abstract data type) is provided by abstract, trait and sealed

Defining

- Defining a case class is simple using **keywords case and class**.
- No body is required we are just telling to make a tuple kind of object.

```
case class MyCaseClass (empId: Int, empName: String, empAddr: String)
```

Using

- Case classes automatically get many functions other functionality, like following
 - Equality operators(==) - for comparing if two case class object have same content or not
 - Copy() - function for making copy of case class object
 - Equals() - same as java (? Not sure)
 - toString() - same as java invoked when we try to print the object reference
 - hashCode() - same as java (? Not sure)
- Case classes can be used as any simple class, it only have every attribute public and some predefined features.

```
// equals, copy, hashCode, toString
// very useful for data transfer objects
case class MyCaseClass (empId: Int, empName: String, empAddr: String)

object Test {
  def main(args: Array[String]): Unit = {

    val m1 = new MyCaseClass(100, "Test1", "Somewhere1...")
    val m2 = new MyCaseClass(200, "Test2", "Somewhere2...")

    println(m1 == m2)

    val m3 = m1.copy()
    println(m1 == m3)
    println(m1)
  }
}
```

OUTPUT -

```
<terminated> Test$ (2) [Scala Application] /usr/lib/jvm/jdk1.7.0_67/bin/java (Nov 15, 2016, 8:57:0)
false
true
MyCaseClass(100,Test1,Somewhere1...)
```

- **Case classes** are immutable so we can override the content of a object after it has been made.

```
var m1 = new MyCaseClass(100, "Test1", "Somewhere1...")
val m2 = new MyCaseClass(200, "Test2", "Somewhere2...")
|
m1.empId = 200
```

- we can see we are getting red line into assignment, so we can only assign values when it is being created or being copied.
- While copying we can pass the argument we want to change, others will be same as the object from which we are copying.

```
case class MyCaseClass (empId: Int, empName: String, empAddr: String)

object Test {
  def main(args: Array[String]): Unit = {
```

```

case class MyCaseClass (empId: Int, empName: String, empAddr: String)

object Test {
  def main(args: Array[String]): Unit = {

    val m1 = new MyCaseClass(100, "Test1", "Somewhere1...")
    val m2 = new MyCaseClass(200, "Test2", "Somewhere2...")

    println(m1 == m2)

    val m3 = m1.copy(empId = 200)
    println(m1 == m3)
    println(m1.hashCode())
  }
}

```

- Here we get everything from m1, only **empId** get changed to 200.
- While copying caseClasses we do not need to use **copy()** method as default assignment operator also clone the object and assign(not assigning reference only).

```

case class MyCaseClass (empId: Int, empName: String, empAddr: String)

object Test {
  def main(args: Array[String]): Unit = {

    val m1 = new MyCaseClass(100, "Test1", "Somewhere1...")
    val m2 = new MyCaseClass(200, "Test2", "Somewhere2...")

    println(m1 == m2)

    val m3 = m1
    println(m1 == m3)
    println(m1.hashCode())
  }
}

```

RULE OF THUMB - use simple **assignment operator(=)** for simple copying, and **copy()** method if we want to override some values.

THEORY

Scala has a feature called case class, which is a fundamental structure to represent an immutable data. A case class contains various helper methods, which provide automatic accessors, and methods such as copy (return new instances of the case class with modified values), as well as implementations of hashCode and equals.

```

case class User(id: Int, firstName: String, lastName: String)

User(1, "Bob", "Elvin").copy(lastName = "Jane") // returns User(1, "Bob", "Jane")

```

These combined features allow you to compare case classes directly. Case classes also provide a generic toString method to pretty print the constructor values.

```

User(1, "Bob", "Elvin").toString // returns "User(1,Bob,Elvin)"

```


Inheritance and Abstract Class in scala

Friday, April 28, 2017 2:12 AM

Abstract Class

- Abstract class is same as java, we can't not create object of abstract class, and any one function will be with out their body.

Inheritance

- Like java a class can extends from only 1 class at a time,so for more traits(like interfaces in java)

Extending a Class

You can extend a base Scala class and you can design an inherited class in the same way you do it in Java (use **extends** key word), but there are two restrictions: method overriding requires the **override** keyword, and only the **primary** constructor can pass parameters to the base constructor. Let us extend our above class and add one more class method.

overriding the parents value , so

use override.

this will automatically pass the value coming to this constructor to the constructor of parent.

Example

Let us take an example of two classes Point class (as same example as above) and Location class is inherited class using extends keyword. Such an '**extends**' clause has two effects: it makes Location class inherit all non-private members from Point class, and it makes the type *Location* a subtype of the type *Point* class. So here the *Point* class is called **superclass** and the class *Location* is called **subclass**. 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.

- Point is base class and Location is derived class. And object Demo is used for running.

```
class Point(val xc: Int, val 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);  
    }  
}
```

- Here we are overriding the xc (or using xc of Point)
- **Point** ka hi use kar rahe hai lekin ye batane ke liye ki ye iska nahi hai data member
- Either we are overriding(or using and implementing in our way), datamember or function we have to use keyword **override**.

```

class Location(override val xc: Int, override val yc: Int,
    val zc :Int) extends Point(xc, yc){
    var z: Int = zc

    def move(dx: Int, dy: Int, dz: Int) {
        x = x + dx
        y = y + dy
        z = z + dz
        println ("Point x location : " + x);
        println ("Point y location : " + y);
        println ("Point z location : " + z);
    }
}

```

Note: Methods move() method in Point class and **move() method in Location class** do not override the corresponding definitions of move since they are different definitions (for example, the former take two arguments while the latter take three arguments).

- It is **same** as Cross class overloading in JAVA(in C++ it would have been overhiding).

```

object Demo {
    def main(args: Array[String]) {
        val loc = new Location(10, 20, 15);

        // Move to a new location
        loc.move(10, 10, 5);
    }
}

```

OUTPUT -

```

"C:\Program Files\Java\jdk1.8.0_71\bin\java" ...
Point x location : 20
Point y location : 30
Point z location : 20

```

Example - 2

- In this example we have following hierarchy
 - MyAbstractFile
 - MyFile extends MyAbstractFile
 - MyCompressedFile extends MyFile
- Now we have to implement this situation.
- Here not variable is passed to any Class so no need to use keyword 'override' for variable.

```

= abstract class MyAbstractFile {
    def open(filename: String): Unit
    def save(filename: String): Unit
}

= class MyFile extends MyAbstractFile {
    override def open(filename: String): Unit = {
        println("MyFile.open method called...")
    }

    override def save(filename: String): Unit = {
        println("MyFile.save method called...")
    }
}

```

- In we have overridden abstract methods in 'MyFile' which will to to MyCompressedFile
- We have only overridden save() for MyCompressedFile as open() coming from MyFile is not abstract.

```

= class MyCompressedFile extends MyFile {
    override def save(filename: String) = {
        println("MyCompressedFile.save method called...")
        println(">>> Implementing compression logic")
        println(">>> Calling the immediate base save method now...")
        super.save(filename)
    }
}

```

- For calling parent function we can use 'super' as java.
- As we can see runtime polymorphism is applied by assigning child's object to parent's reference(like java and C++)
- There is not difference between java and scala inheritance.

```

object TestInhertanceDriver {
    def main(args: Array[String]) {
        var f: MyAbstractFile = new MyFile()

        println("*** Testing MyFile ***")

        f.open(null)
        f.save(null)

        | println("*** Testing MyCompressedFile ***")

        f = new MyCompressedFile()
        f.open(null)
        f.save(null)
    }
}

```

OUTPUT -

```

<terminated> TestInhertanceDriver$ (2) [Scala Application] /usr/lib/jvm/jdk1
|*** Testing MyFile ***
MyFile.open method called...
MyFile.save method called...
*** Testing MyCompressedFile ***
MyFile.open method called...
MyCompressedFile.save method called...
>>> Implementing compression logic
>>> Calling the immediate base save method now...
MyFile.save method called...

```


Traits

Friday, April 28, 2017 4:04 AM

- Traits are similar to interfaces in java, but with couple of differences.
 - Traits can have state variables(to store state)
 - Traits can have implementations of methods
- Child class must have to implement all the methods which are not implemented (this property is same as **abstract class** of java and scala)

Traits/Interface vs abstract class - in any OOP

- A class can inherits may **trait/interface** but only one **abstract class** as it is class.
- Abstract class is used when we wanted to put some common functionality In super class(**abstraction in common hierarchy**) where traits are used for **providing specifications**.

Example -

- **Case 1** - like for abstraction in 'File' hierarchy

`files -> normal files, compressed, encrypted,...`

A common abstract class 'File' can be created to store all common methods like, `getDetail()`, `getName()` etc and some common property, in this case we use **abstract classes**.

- **Case 2**- let consider case of remotes working on different kind of tvs, following buttons(functions)

`tv remote -> red, blue, green, vol +/-, ch +/-,...`

We can provide functions like `redpressed()`, `greenpressed()` etc in **a trait**, and then we can provide specifications in the inherited class as follows.

`sony tv -> red will show the date`
`lg tv -> red will show mails...`

In this case we use trait/interface, like follows

```
traits TVRemote {  
    def onRed(): Unit = {}  
    ...  
}
```

RULE OF THUMB -

- Use abstract class for hierarchy and trait for adding some functionality/specifications

Making Traits

- As we can see body of `isNotEqual()` is provided and so it user is not bound to override it.

```
trait Equal {  
    def isEqual(x: Any): Boolean  
    def isNotEqual(x: Any): Boolean = !isEqual(x)  
}
```

This trait consists of two methods **isEqual** and **isNotEqual**. Here, we have not given any implementation for `isEqual` where as another method has its implementation. Child classes extending a trait can give implementation for the un-implemented methods. So a trait is very similar to what we have **abstract classes** in Java.

```
class Point(xc: Int, yc: Int) extends Equal {  
  var x: Int = xc  
  var y: Int = yc  
  def isEqual(obj: Any) =  
    obj.isInstanceOf[Point] &&  
    obj.asInstanceOf[Point].x == x  
}
```

```
object Demo {  
  def main(args: Array[String]) {  
    val p1 = new Point(2, 3)  
    val p2 = new Point(2, 4)  
    val p3 = new Point(3, 3)  
  
    println(p1.isNotEqual(p2))  
    println(p1.isNotEqual(p3))  
    println(p1.isNotEqual(2))  
  }  
}
```

Output:

false
true
true

Functional Programming

→ **Apart from being a pure object oriented language, Scala is also a functional programming language**

→ **Functional programming is driven by mainly two ideas:**

» First main idea is that functions are first class values. They are treated just like any other type, say String, Integer etc. So functions can be used as arguments, could be defined in other functions

» The second main idea of functional programming is that the operations of a program should map input values to output values rather than change data in place. This results in the immutable data structures

→ **Scala supports both, immutable and mutable data structures. However, the choice is for immutable ones**

- Based on lambda calculations.
- There are total 7 properties of functional programming.

Features

- Immutability
- Higher order functions
- lazy loading
- Pattern matching
- Currying
- Partial application
- Monads

- Function is a first class citizen and so it could be assigned to a variable.
- When only one statement need to be there in a function we need not to use {}, like loops and control statements.

```
scala> def areaRect(l: Int, b: Int): Int = l * b
areaRect: (l: Int, b: Int)Int

scala> areaRect(10, 9)
res0: Int = 90
```

- Now we can assign it to some variable and see it's signature.

```
scala> val f = areaRect
f: (Int, Int) => Int = <function2>

scala> f(10, 9)
res1: Int = 90

scala> f
res2: (Int, Int) => Int = <function2>
```

- As we can see 'f' is a function having signature as above.

Function Passing

Monday, April 24, 2017 2:22 AM

Higher-Order Functions

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

Try the following example program, `apply()` function takes another function `f` and a value `v` and applies function `f` to `v`.

```
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() + "]"  
  
}
```

There are many predefined higher order functions(like fold,reduce,map), which will learn later.

Function Passing

- Example - here we have created a function "operation" which is taking 3 arguments
 - 1 Function and 2 integers.
- We can see the syntax of passing function(signature of function)
 - `Fun_name:(arglist....) => <return_type>`

```

object myobj {
  def main(args: Array[String]): Unit = {
    println(operation(add,10,20));
    //illustrating fn passing using signature
    println(operation((a,b)⇒a*b,10,20));
  }
  def operation(fun:(Int,Int)⇒Int,op1:Int,op2:Int):Int={
    fun(op1,op2);
  }
  def add(a:Int,b:Int):Int={
    a+b;
  }
}

```

Anonymous Functions

Scala provides a relatively lightweight syntax for defining anonymous functions. Anonymous functions in source code are called **function literals** and at run time, function literals are instantiated into objects called **function values**.

Scala supports **first-class** functions, which means functions can be expressed in function literal syntax, i.e., $(x: \text{Int}) \Rightarrow x + 1$, and that functions can be represented by objects, which are called function values.

Try the following expression, it creates a successor function for integers:

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

Variable `inc` is now a function that can be used the usual way:

```
var x = inc(7)-1
```

It is also possible to define functions with multiple parameters as follows:

```
var mul = (x: Int, y: Int) => x*y
```

Variable `mul` is now a function that can be used the usual way:

```
println(mul(3, 4))
```

- We need not to create separate function for passing, we can easily pass the anonymous function using signature, like above in rounded.
- Anonymous functions are same as lambda expressions.

It is also possible to define functions with no parameter as follows:

```
var userDir = () => { System.getProperty("user.dir") }
```

Variable userDir is now a function that can be used the usual way:

```
println( userDir() )
```

```
= object GenericOp {  
  def genericOp[I <: Any](f: (I, I) => I, op1: I, op2: I): I = {  
    f(op1, op2)  
  }  
}
```

- Here [T<:Any] is same as (? Extends <class_name>) in java.
It means T should be subclass of Any.
Here ofcourse this is redundant as every class in Scala is subclass of 'Any' class (same like Object in java)

Example - 2 writing generic functions.

```
package test.hof  
  
/**  
 * @author edureka  
 */  
object GenericOp {  
  def genericOp[T <: Any](f: (T, T) => T, op1: T, op2: T): T = {  
    f(op1, op2)  
  }  
  def op(f: (Int, Int) => Int, op1: Int, op2: Int): Int = {  
    f(op1, op2)  
  }  
  def multiply(x: Int, y: Int): Int = {  
    x * y  
  }  
  def myfilter[T](f: (T) => Boolean, l: List[T]): List[T] = {  
    var res = new scala.collection.mutable.ArrayBuffer[T]()  
    l.foreach { element =>  
      if (f(element))  
        res += element  
    }  
    res.toList  
  }  
}  
  
def main(args: Array[String]): Unit = {  
  // calling by name  
  println(op(multiply, 2, 3))  
  
  // using Function Literals/Anonymous Functions/Lambda functions  
  println(op((x, y) => x * y, 2, 3))  
  
  // calling with any numeric types  
  println(genericOp[Float]((x, y) => x * y, 2.5f, .3f))  
}
```

```
def main(args: Array[String]): Unit = {  
  // calling by name  
  println(op(multiply, 2, 3))  
  
  // using Function Literals/Anonymous Functions/Lambda functions  
  println(op((x, y) ⇒ x * y, 2, 3))  
  
  // calling with any numeric types  
  println(genericOp[Float]((x, y) ⇒ x * y, 2.5f, .3f))  
  // alternative syntax  
  println(genericOp((x: Int, y: Int) ⇒ x * y, 2, 3))  
  
  val l = (1 to 10).toList  
  myfilter[Int](n ⇒ n % 2 == 0, l).foreach(println)  
}
```

Friday, April 28, 2017 6:59 PM

1.7.2.8 Higher-Order Methods

These are all methods that take a function as an argument. That gives them a great deal of power that you do not get from the standard methods. Odds are that you have not used these types of methods before, so we will discuss the three you will likely use the most, then give a quick list of the others.

The simplest higher-order method to understand is `foreach`. It takes a single argument of a function, and calls that function with each element of the collection. The `foreach` method results in `Unit`, so it is not calculating anything for us, the function that is passed in needs to have side effects to make it useful. Printing is a common use case for `foreach`.

```
nums.foreach(println) // Print all the values in nums, one per line.
words.foreach(println) // Print all the values in words, one per line.
```

In this usage, we have simply passed in `println` as a function. That works just fine, but if we wanted to print twice the values of each of the numbers, we would likely use an explicit lambda expression.

```
nums.foreach(n => println(2*n)) // Print twice the values in nums.
```

The fact that `foreach` does not produce a value limits its usefulness. The higher-order methods you will probably use the most do give back values. They are `map` and `filter`. The `map` method takes a function that has a single input of the type of our collection, and a result type of whatever you want. This function is applied to every element of the collection and you get back a new collection with all the values that function produces. Here are two ways to get back twice the values in `nums` using `map`.

```
nums.map(n => n*2)    // Array(12, 16, 6, 8, 2, 14)
nums.map(_*2)         // Array(12, 16, 6, 8, 2, 14)
```

This example used a function that took an `Int` for input and resulted in an `Int`. Those types do not have to match though. Here we use `map` to calculate the lengths of all the values in `words`.

```
words.map(_.length)  // List(5, 2, 3, 2, 4, 2)
```

The third commonly used higher-order method that we will discuss is `filter`. As the name implies, this method lets some things go through and stops others. It is used to select elements from a collection that satisfy a certain PREDICATE. A predicate is a function that takes an argument and results in a `Boolean` that tells us if the value satisfies the predicate. Here are three different example uses of `filter`, all of which use the underscore shorthand for the lambda expression that is the predicate.

```
nums.filter(_ < 5)      // Array(3, 4, 1)
nums.filter(_ % 2 == 0) // Array(6, 8, 4)
words.filter(_.length > 2) // List("Scala", "fun", "code")
```

- **Predicate Methods** - These methods all take a predicate as the main argument. The `filter` method would be part of this list.

- `count(p: A => Boolean)` - Gives back the number of elements that satisfy the predicate.

```
nums.count(_ < 5)           // 3
```

- `dropWhile(p: A => Boolean)` - Produces a new collection where all members at the front the satisfy the predicate have been removed.

```
nums.dropWhile(_ % 2 == 0)  // Array(3, 4, 1, 7)
```

- `exists(p: A => Boolean)` - Tells you where or not there is an element in the sequence that satisfies the predicate.

```
nums.exists(_ > 8)          // false
words.exists(_ == "code")   // true
```

- `filterNot(p: A => Boolean)` - Does the opposite of `filter`. It gives back a new collection with the elements that do not satisfy the predicate.

```
nums.filterNot(_ < 5)       // Array(6, 8, 7)
nums.filterNot(_ % 2 == 0)  // Array(3, 1, 7)
words.filterNot(_.length > 2) // List("is", "to", "in")
```

- `find(p: A => Boolean)` - Gives back an `Option[A]` with the first value that satisfied the predicate. See section 1.7.3 for details of the `Option` type.

```
nums.find(_ % 2 == 1)       // Some(3)
words.find(_.length < 2)    // None
```

- `forall(p: A => Boolean)` - The counterpart to `exists`. This tells you if the predicate is true for all values in the sequence.

```
nums.forall(_ < 9)          // true
words.forall(_ == "code")   // false
```


- `indexWhere(p: A => Boolean)` - Gives back the index of the first element that satisfies the predicate.

```
nums.indexWhere(_ < 5)          // 2
words.indexWhere(_.contains('o')) // 3
```

- `lastIndexWhere(p: A => Boolean)` - Gives back the index of the last element that satisfies the predicate.

```
nums.lastIndexWhere(_ < 5)      // 4
words.lastIndexWhere(_.contains('o')) // 4
```

- `partition(p: A => Boolean)` - Produces a tuple with two elements. The first is a sequence of the elements that satisfy the predicate and the second is a sequence of those that do not. So `s.partition(p)` is a shorter and more efficient way of saying `(s.filter(p), s.filterNot(p))`.

```
nums.partition(_ < 5)           // (Array(3, 4, 1), Array(6, 8, 7))
words.partition(_.length > 2) // (List("Scala", "fun", "code"),
                                List("is", "to", "in"))
```

- `prefixLength(p: A => Boolean)` - Tells how many elements at the beginning of the sequence satisfy the predicate.

```
nums.prefixLength(_ > 5)       // 2
words.prefixLength(!_.contains('o')) // 3
```

- `takeWhile(p: A => Boolean)` - Produces a new sequence with the elements from the beginning of this collection that satisfy the predicate.

```
nums.takeWhile(_ % 2 == 0)     // Array(6, 8)
```

- **Folds** - These methods process through the elements of a collection moving in a particular direction. The ones that go from left to right are listed here, but there are others that go from right to left. These methods are a bit more complex and can take a while to understand. Once you understand them, you can replace most code that would use a loop and a `var` with one of them.

- `foldLeft(z: B)(op: (B, A) => B)` - This curried method produces an element of type `B`, which is the type of the first argument, the first argument to the operation, and the output of the operation. It works by going from left to right through the collection, and for each element it passes the accumulated value and the next collection element into the `op` function. The result is used as the accumulated value for the next call. `z` is the first value for the accumulator, and the last one is the result of the function.

```
nums.foldLeft(0)(_ + _)        // 29
nums.foldLeft(1)(_ * _)        // 4032
words.foldLeft(0)(_ + _.length) // 18, which is the sum of all the
                                lengths of the strings
```

- Left wale yaha diye hai ,similary we have Rights

- `reduceLeft(op: (A, A) => A)` - This method works like `foldLeft`, but it is a little more restricted. There is no initial accumulator, as the first call to `op` is passed the first and second elements. The output of that is then passed in with the third element and so on. The value after the last element is the result of the `reduce`.²²

```
nums.reduce(_ + _)           // 29
nums.reduce(_ * _)           // 4032
```

- `scanLeft` - Does what a `foldLeft` does, but returns a sequence with all the intermediate results, not just the last one.

```
nums.scanLeft(0)(_ + _)      // Array(0, 6, 14, 17, 21, 22, 29)
nums.scanLeft(1)(_ * _)      // Array(1, 6, 48, 144, 576, 576, 4032)
words.scanLeft(0)(_ + _.length) // List(0, 5, 7, 10, 12, 16, 18)
```

• Other Methods

- `flatMap(f: A => Seq[B])` - This works in the same way as `map`, but is useful when the function produces sequences and you do not want a `List[List[B]]` or `Array[Array[A]]`. It flattens the result to give you just a basic sequence. The example here shows a comparison with `map`.

```
Array(2, 1, 3).map(1 to _)    // Array(Range(1, 2), Range(1), Range(1, 2, 3))
Array(2, 1, 3).flatMap(1 to _) // Array(1, 2, 1, 1, 2, 3)
```

- `maxBy(f: A => B)` - Gives back the maximum element based on comparison of the B values produced by passing the elements of this collection into `f`.

```
words.maxBy(s => s(0))        // "to", this is looking at the first letter
                               of each word in the sequence and finding the max
```


- `minBy(f: A => B)` - Gives back the minimum element based on comparison of the B values produced by passing the elements of this collection into `f`.

```
words.minBy(s => s(0))      // "Scala", this is looking at the first
                             letter of each word in the sequence and finding the min
```

- `sortBy(f: A => B)` - Produces a new collection with the elements sorted based on the B values produced by passing the elements of this collection to `f`. The following example sorts the `Strings` based on the second character.

```
words.sortBy(s => s(1))     // List("Scala", "in", "to", "code", "is",
                             "fun")
```

- `sortWith(lt: (A, A) => Boolean)` - Produces a new collection with the elements sorted based on the comparison function that is passed in. The `lt` function, short for “less than”, should return true if the first argument should come before the second argument. The following example uses a greater-than comparison to sort in reverse order.

```
words.sortWith((s1, s2) => s1 > s2) // List("to", "is", "in", "fun",
                                             "code", "Scala")
```

Map function

Friday, April 28, 2017 6:14 PM

Theory-

A common pattern is to take an element from one array, transform it, and insert it into another. Let's put aside the version with `while`, because we already know why it is considered bad. But look at the previous code: doesn't it have the same problems? It has a mutable state coupled with side effects, and poor method extraction. The fact that this is called micro management is another reason and looks like what is shown in Figure 2-3.



FIGURE 2-3

You need to transform the pile in Figure 2-3 on the left into the pile on the right. When doing `for` loops, you are telling each one of the workers to go to the first pile, take the box, change its color, go to the second pile, and put it there. This is for every loop and iteration. This is quite tiresome for real life management. Why not just tell them to transform this pile of white boxes into the dark ones and leave them to do their job? Luckily, in functional programming there is a function that does such a thing. It is called `map()`, and here is how you can refactor the previous iterative code:

```
val users = List(User("Alex", 26), User("Sam", 24))

var names = users.map(user => user.name)

Or its shortened and, arguably, more concise version:

var names = users.map(_.name)
```

The function that is passed to the method is called `lambda expression`. If at first the `map()` doesn't sound like a "transform" to you, don't worry, because after time in functional programming, you will find it normal because the `map()` method is everywhere. But there is more on this in Chapter 10 on monadic types.

- We can use wildcard(`_`) or assignment syntax.
- Map function can be applied to any collection, here we have taken definition from Array's Documentation so it show array, but `map()` is present in in all collections.

```
def map[B](f: (A) => B): Array[B]
```

[use case]
Builds a new collection by applying a function to all elements of this array.

B	the element type of the returned collection.
f	the function to apply to each element.
returns	a new array resulting from applying the given function <code>f</code> to each element of this array and collecting the results.

Implicit This member is added by an implicit conversion from `Array[T]` to `ArrayOps[T]` performed by method `genericArrayOps` in `scala.Predef`.

Definition Classes `TraversableLike` → `GenTraversableLike` → `FilterMonadic`

Full Signature

- `Map()` function has nothing to do with `Map` class.
- `Map()` function takes a function as argument and do transformation on values of collection and return new formed collection(it do not change original).

Example -

```
scala> val l = (1 to 20).toList
l: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20)

scala> l
res0: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20)

scala> l.map(n => n * 2)
res1: List[Int] = List(2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40)
```

- We can also use wild card instead of variable.

```
scala> l.map(_ * 2)
res5: List[Int] = List(2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40)
```

Filter function

Friday, April 28, 2017 6:40 PM

The second most common operation on a collection is ridding it of unwanted elements:

```
val users = List(User("Alex", 12), User("Sam", 22))

var adults = List[User]()

for (user <- users) {
  if (user.age >= 18) {
    adults = adults :+ user
  }
}

println(users)
```

Here you are like club security checking everyone personally, instead of just filtering the flow:

```
val adults = users.filter(_.age >= 18)
```

- The function passed to **filter()** must be a predicate(return type of function should be boolean only)

Example -

```
scala> l
res6: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20)

scala> l.filter(n => n % 2 != 0)
res7: List[Int] = List(1, 3, 5, 7, 9, 11, 13, 15, 17, 19)

scala> l.filter(_ % 2 != 0)
res8: List[Int] = List(1, 3, 5, 7, 9, 11, 13, 15, 17, 19)
```

Reduce function

Friday, April 28, 2017 6:48 PM

reduce -> accumulates the value of the collection

```
scala> l
res10: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20)

scala> l.reduce((acc, n) => acc + n)
res11: Int = 210

scala> l.reduce(_ + _)
res12: Int = 210
```

- In reduce the accumulate variable is automatically initialised to identity value according to that operation, so if the operation is sum then 0, product then 1.

Example - take element from above list and do following operation in one line

- Filter out odd
- Square each element
- Add them to get ans.

```
scala> list
res8: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20)

scala> list.filter(_%2!=0).map((n)=>n*n).reduce(_+_ )
res10: Int = 1330
```

FlatMap functions

Friday, April 28, 2017 7:18 PM

- FlatMap() is used to make 2dimensional collection=ion to 1d.
- flatMap(f: A => Seq[B]) - This works in the same way as map, but is useful when the function produces sequences and you do not want a List[List[B]] or Array[Array[A]]. It flattens the result to give you just a basic sequence. The example here shows a comparison with map.

```
Array(2, 1, 3).map(1 to _) // Array(Range(1, 2), Range(1), Range(1, 2, 3))
Array(2, 1, 3).flatMap(1 to _) // Array(1, 2, 1, 1, 2, 3)
```

- maxBy(f: A => B) - Gives back the maximum element based on comparison of the B values produced by passing the elements of this collection into f.

```
words.maxBy(s => s(0)) // "to", this is looking at the first letter
of each word in the sequence and finding the max
```

Example -

- we have a array of string "words", Now if we want to get all words we can split using split(" ") function(like java)

```
val words = Array("Hi there", "How are you", "Are you still there")
Array[String] = Array(Hi there, How are you, Are you still there)

words
Array[String] = Array(Hi there, How are you, Are you still there)

words.map(_.split(" "))
Array[Array[String]] = Array(Array(Hi, there), Array(How, are, you), Array(Are, you, still, there))
```

- But we get array[array[string]] here, not array[string] having words , so we have to flatten this using FlatMap() function
- This function first map and then flatten to create a single list.

```
scala> words.flatMap(_.split(" "))
res20: Array[String] = Array(Hi, there, How, are, you, Are, you, still, there)
```