Introduction to Programming in Scala

April 27th, 2017

Hungai Amuhinda

@hungai

hungaikev

hungaikev.in

hungaikevin@gmail.com

Agenda

- Introduction
- Scala Concepts
- OOP
- FP
- Collections

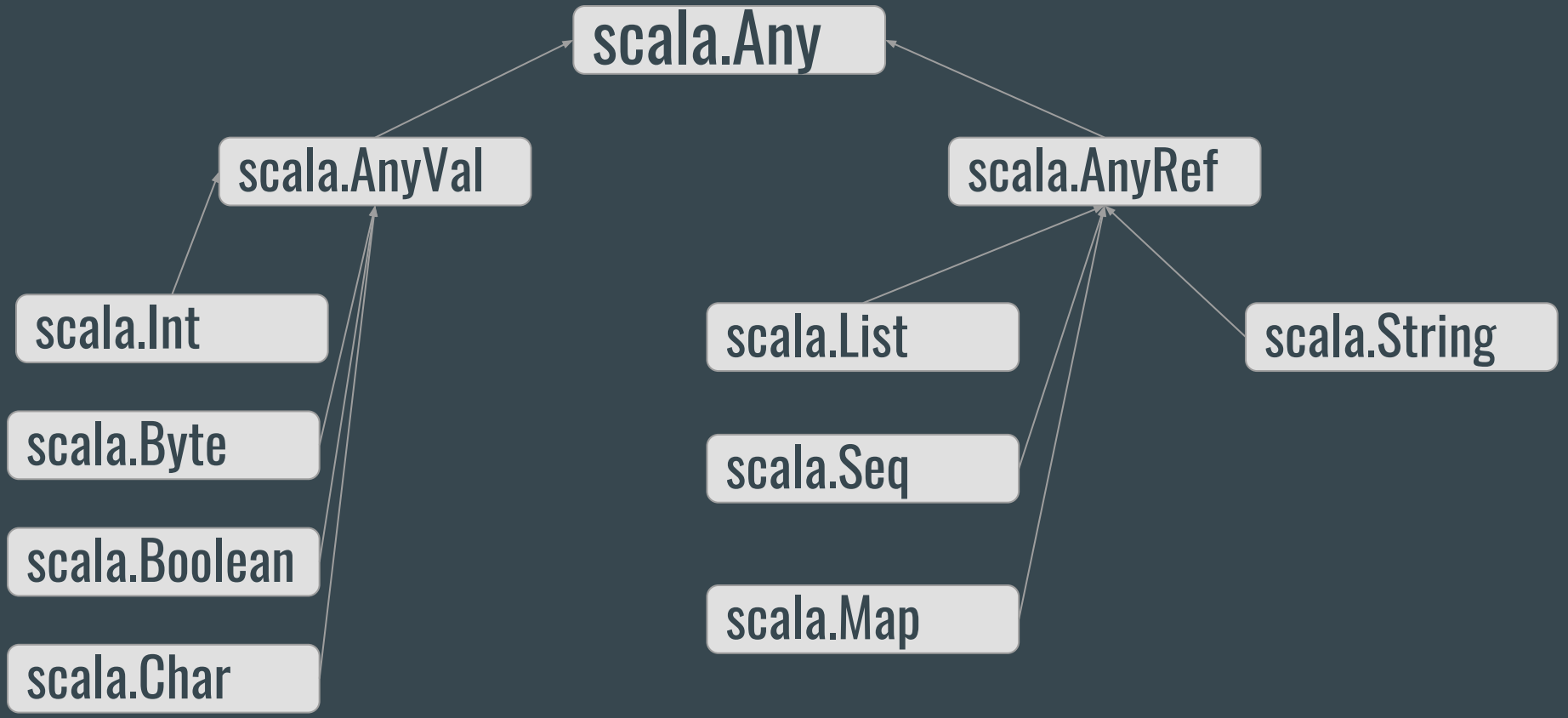
Scala:

Is a general purpose programming language providing support for functional programming and a strong type system

Scala Concepts

- Everything is an object
- Scala is an object oriented language in pure form: every value is an object and every operator is a method.
- “ + ”, “ - ”, “ * ”, “ / ” are all methods

Scala Class Hierarchy



Scala Concepts

- Everything is an expression
- Expression is an instruction to execute something that will return a value.
- An expression evaluates to a result or results in a value

Scala Concepts(Advanced Type System)

- Static
- Inferred (Type Inference)
- Structural
- Strong

Scala Data Types

- Boolean
- Byte
- Short
- Int
- Long
- Float
- Double
- Char
- String

Program with Scala (Main)

```
object Demo1 {

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

    println("Hello Everyone")

    println("Welcome to today's event")

  }

}
```

Program with Scala (App)

```
object Demo2 extends App {
    val todaysEvent = “Scavannah”
     lazy val fun = (0 to 4).map(x => “fun”).mkString(“ ”)
    println(“Hello world”)
    println(“Welcome to ” + todaysEvent + “! \n”)
}
```

Program with Scala (REPL)

Read Evaluate Print Loop

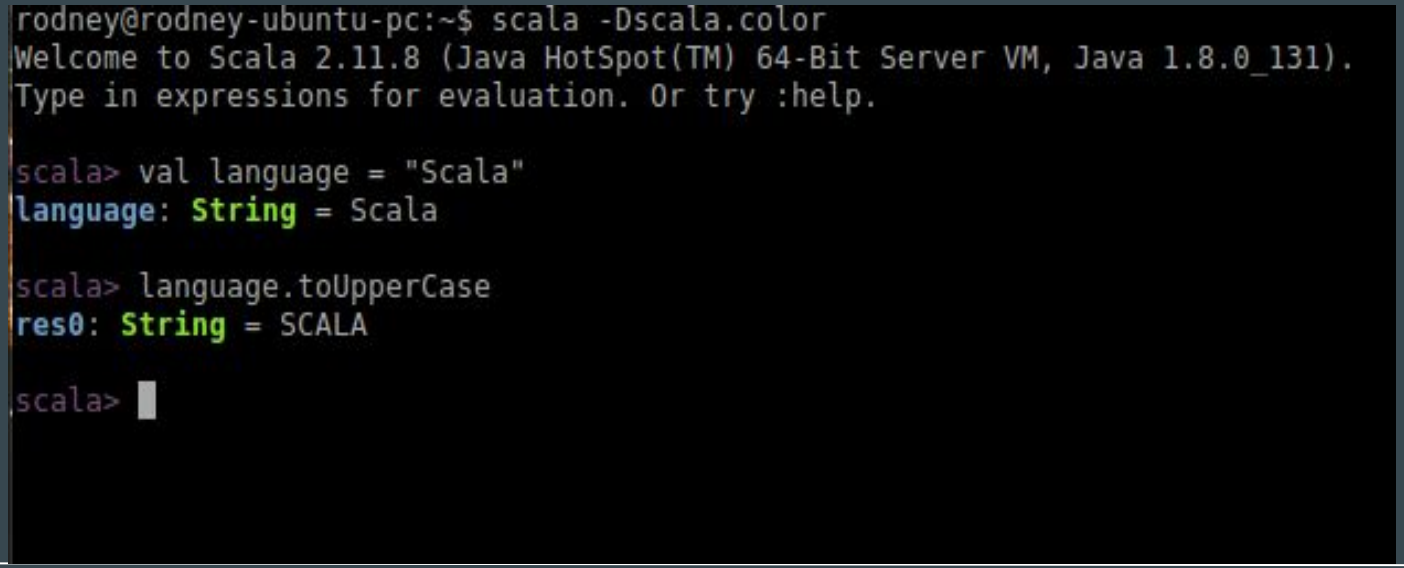
$ scala

```
rodney@rodney-ubuntu-pc:~$ scala
Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_131).
Type in expressions for evaluation. Or try :help.

scala> val language = "Scala"
language: String = Scala

scala> language.toUpperCase
res0: String = SCALA

scala>
```

Program with Scala (REPL)

Read Evaluate Print Loop

$ scala -Dscala.color



Program with Scala (Worksheet)

Work with worksheets

- IntelliJ
- Scala IDE or Eclipse with Scala Plugin
- Ensime

Scala Concepts

- var, val
- If expressions
- def
- Block expressions
- While - Loops
- For - Loops
- Nested Function
- Recursion vs Tail recursion
- Pattern Matching

Scala Concepts

- var - variable

Something that is able or likely to change or be changed (Mutable)

- val - value

A value is an expression that cannot be changed any further (Wiki) (Immutable)

It's similar to final in Java

Expression with Semicolon?

```
val x = 5566

val y = 87

val java = “Java” ; val scala = “scala”
```

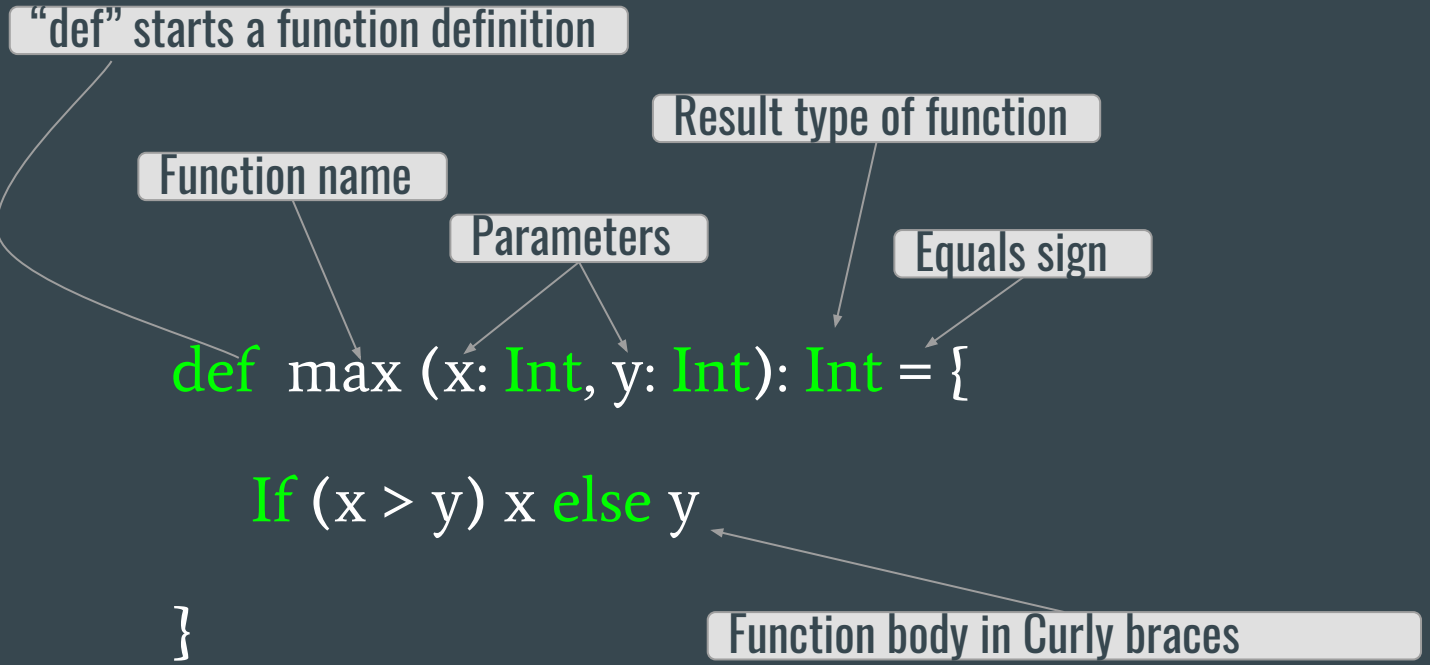
If you have multiple expressions in one line, you will need semicolons(;). Otherwise you don't need it.

If Expressions

- If has return value (expression)
- Scala has no ternary operator (? :)

```
val value  = 0

val negative = If (value < 0 ) true else false
```

def

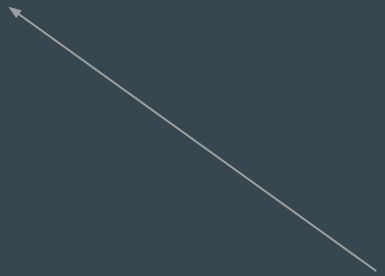

"def" starts a function definition
Result type of function
Function name
Parameters
Equals sign
def max (x: Int, y: Int): Int = {
If (x > y) x else y
}
Function body in Curly braces

def

```
def  max (x: Int, y: Int): Int = {
     If (x > y)
        return  x
      else
        return y
}
```

def

```
def  max (x: Int, y: Int)  =  {
    If (x > y)
        x
     else
        y
}
```



No Result type

def

```
def  max (x: Int, y: Int)  =
    If (x > y)
       x
     else
      y
```

No Curly brackets

Summary of def

- You don't need a return statement
  - Last expression of a block will be the return value
- You don't need return type in method or function definition.
  - Scalac will infer your return type in most cases. Its a good habit to have a return type, especially if your API has a public interface
- You don't need curly brackets
  - If you have multiple lines of code, using curly brackets ({ }) is a good habit

Block expressions (Curly brackets)

```
val n = 5

val factorial = {
  var  result = 1
    for (i <- 1 to n )
      result  = result * i
   result
}
```

Last expression (result) in block will be the return value, then it will be assign result to “factorial”

While Loop

```
var n = 10

var sum = 0
while (n > 0) {
      sum = sum + 1
    n = n - 1
}
```

For - Loop

```
var sum = 0
for  (i <- 1 to 10) {
      sum += 1
   }
println(sum)
```

___

Nested Function

```
def min (x: Int, y: Int): Int = {
        def  max (x: Int, y: Int)  =  {
            if (x > y)
                x
            else
                y
        }
   if  (x >= max(x,y)) y else x
}
```

Recursion vs Tail Recursion

Recursion vs Tail - recursion

```
def   factorial(n : Int): BigInt = {
            If (n == 0) 1
               else
          n  * factorial(n - 1)
 }
factorial (15)
factorial (150)
factorial(15000)  // java.lang.StackOverflowError
```

Recursion vs Tail - recursion

```
def   factorial(n : Int): BigInt = {
        def helpFunction(acc : Int, n: Int) : BigInt = {
              If (n == 0)
                   acc
               else
                   helpFunction(acc * n , n -1 )
             }
    helpFunction(1,n)
  }
factorial(15000)
```

"Tail - Recursion"

Recursion vs Tail - recursion

```
import scala.annotation.tailrec
def   factorial(n : Int): BigInt = {
       @tailrec
        def helpFunction(acc : Int, n: Int) : BigInt = {
                  If (n == 0)
                       acc
                   else
                      helpFunction(acc * n , n -1 )
                 }
    helpFunction(1,n)
  }
factorial(15000)
```

"Add annotation is a good habit. Compiler can check whether or not it can be optimised. "

"You have to add a return type, when the function is recursive"

Pattern Matching

- Pattern matching is a feature that is very common in functional languages
- It matches a value against an expression
- Each pattern points to an expression
- It's similar to “switch case” but its more general. There are some differences:
  - No fall through
  - Each condition needs to return a value(Everything in scala is an expression)
  - It can match anything

Pattern Matching

```
def matchString(x : String): String = {
  x match {
    case "Dog" => x
    case "Cat" => "Not a cat person"
    case _ => "Neither Dog or Cat"
  }
matchString("Dog")
matchString("Human")
```

OOP

Scala (Object Oriented)

- Classes
- Extends , with, override
- Abstract classes
- Objects, Companion objects
- Traits
- Case classes

Classes

Primary Constructor

val in constructor will give you a getter

```
class Employee(id : Int, val name: String, address: String, var salary: Long
)
```

var in constructor will give you a getter and setter

```
val employee = new Employee(1,”Hungai”,”Kakamega”,40L)
```

Extends, with, override.

Single inheritance enables a derived class to inherit properties and behaviour from a single parent class

- Scala is single inheritance like Java.
- Scala - extends = Java - extends
- Scala - with = Java - implements
- Scala - override = Java - @Override

Abstract classes.

- Abstract classes are just like normal classes but can have abstract methods and abstract fields
- In scala, you don't need the keyword abstract for methods and fields in an abstract class

Abstract classes.

```
abstract class Animal (val name: String) {
    val footNumber: Int
    val fly : Boolean
    def speaks : Unit
}
class Dog(name: String) extends Animal (name) {
    override val footNumber : Int = 4
    override val fly = false
    override def speak : Unit = println(“Spark”)
}
class Bird(name : String) extends Animal(name) {
   override val footNumber : Int = 2
   override val fly = true
   override def  speaks: Unit = println(“chatter”)
}
```

Subclasses should be in the same file.

Objects.

Scala has two types of objects

1. Singleton objects
2. Companion objects

- A singleton object is declared using the object keyword.
- When a singleton object shares the same name with a class it's referred to as a Companion object.
- A companion object and its classes can access each others private methods or fields

Singleton Object.

```
object MathUtil {
   def  doubleHalfUp(precision: Int, origin: Double): Double {
        val tmp = Math.pow(10,precision)
         Math.round(origin * tmp)/ tmp
     }
}
```

Case Classes.

Case classes are just regular classes that are :

- ➔ Immutable by default
- ➔ Decomposable through pattern matching
- ➔ Compared by structural equality instead of by reference.

When you declare a case class the scala compiler does the following for you:

- ➔ Creates a class and its companion object
- ➔ Implements the ‘apply’ method that you can use a factory. This lets you create instances of the class without the ‘new’ keyword

Case classes.

```
abstract class Notification

case class Email(sourceEmail: String, title: String, body: String) extends Notification.
case class SMS (sourceNumber: String, message: String) extends Notification.
case class VoiceRecording(contactName: String, link: String) extends Notification.


val emailNotification = Email("h@hungaikev.in","Scavannah","Todays lesson")
println(emailNotification)
```

FP

Scala (Functional)

- Functional Concepts
- First class functions
- Anonymous functions
- Higher order functions

Functional Concepts.

- Immutability (Referential Transparency - Expressions always evaluates to the same result in any context)
  - No side effects (Modifies state, Not predictable)

- Functions as values
  - Functions as objects

- Higher order functions - Input: Takes one or more functions as parameters, Output: Returns a function as result

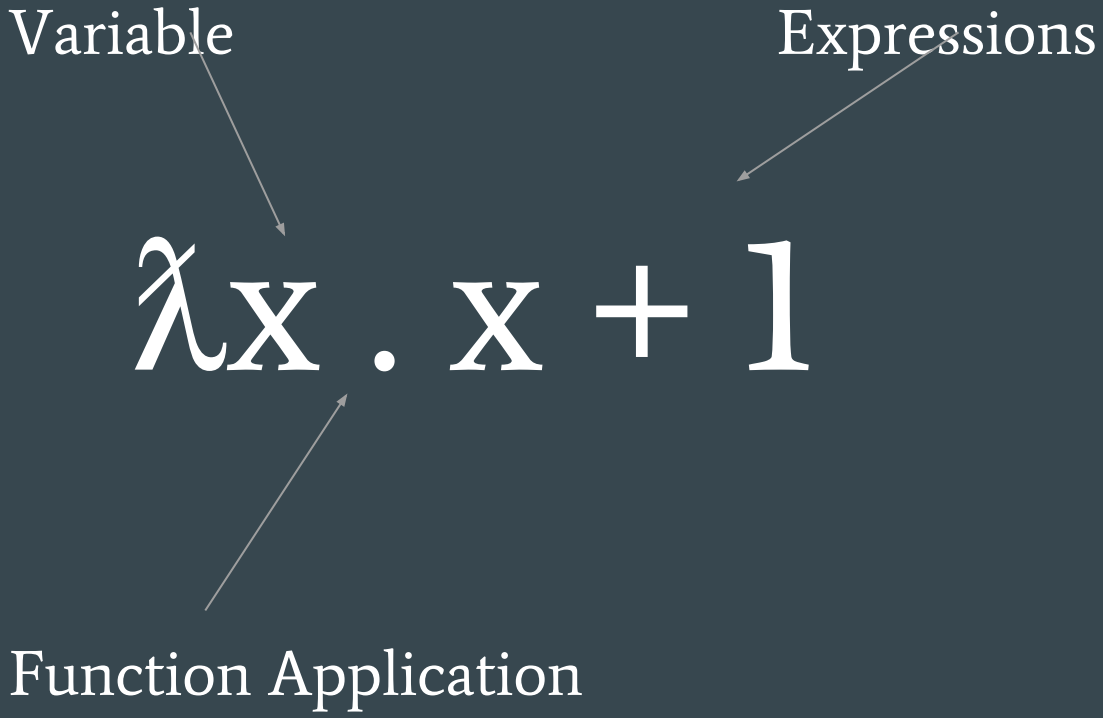
Anonymous functions (Lambdas).

The Lambda Calculus.

Alonzo Church 1930

- Theoretical foundation
- Pure functions - no state
- Not a programming language

Lambda Calculus.



Lambda Calculus.

$$\lambda x \,.\, x + 1$$

// Scala Translation

```
{ x : Int => x + 1 }
```

Anonymous functions.

```
((x : Int ) => x * x)


(0 until 10).map ((value: Int) => value * value)
(0 until 10).map (value => value * value )
(0 until 10).map (value => value + 1)
(0 until 10).map (_ + 1)
```

High-order Functions.

```
def calculateTax(rate: BigDecimal => BigDecimal, salary: BigDecimal) : BigDecimal = {
    rate(salary)
}
val kenyaTax = (salary: BigDecimal) => {
    if (salary > 413201) salary * 0.396 else salary * 0.3
}
val tzTax: BigDecimal => BigDecimal = _ * 0.25


calculateTax(kenyaTax,413201)
calculateTax(tzTax,100)
```

High-order Functions.

```
def calculateTax(rate: BigDecimal => BigDecimal, salary: BigDecimal) : BigDecimal = {
    rate(salary)
}
def kenyaTax  (salary: BigDecimal) = {
    calculateTax(x =>
            if (salary > 413201) salary * 0.396 else salary * 0.3, salary )
}
def  tzTax(salary: BigDecimal ) =
    calculateTax( _ * 0.25, salary)

kenyaTax(413202)
tzTax(100)
```

High-order Functions.

```
def calculateTax(rate: BigDecimal => BigDecimal) : (BigDecimal ) => BigDecimal = {
    rate
}
def kenyaTax  = calculateTax(x =>  if (x > 413201) x * 0.396 else x * 0.3 )

def  tzTax =  calculateTax( x =>  x * 0.25)

kenyaTax(413202)
tzTax(100)

calculateTax(kenyaTax)(413202)
calculateTax(tzTax)(100)
```

High-order Functions - Curry.

```
def calculateTax(rate: BigDecimal => BigDecimal)  (salary : BigDecimal ) :  BigDecimal =
{
    rate (salary)
}
def kenyaTax(salary : BigDecimal)  = calculateTax(x =>  if (x > 413201) x * 0.396 else x *
0.3 )(salary)

def  tzTax(salary : BigDecimal) =  calculateTax( x =>  x * 0.25)(salary)

kenyaTax(413202)
tzTax(100)
```

Collections

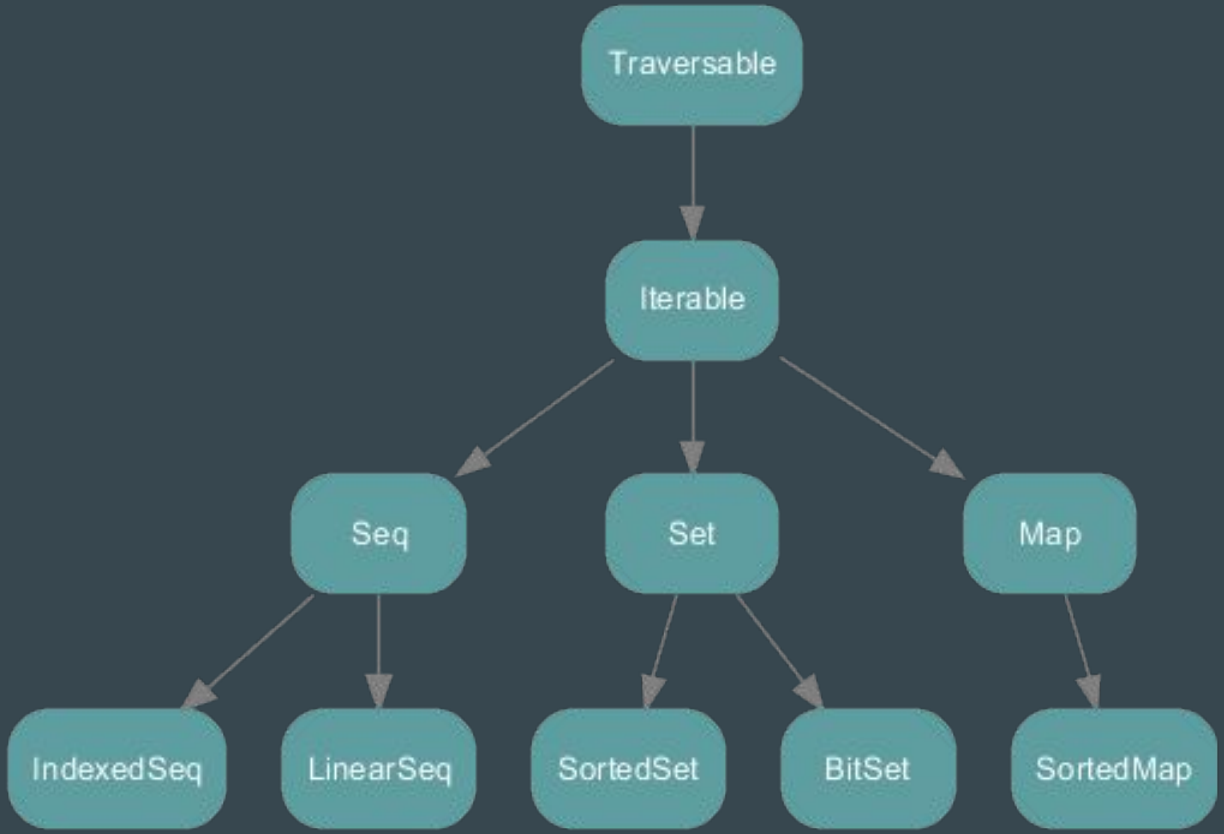
Collections

- Concept of Collections
- Hierarchy of Collections
  - Immutable
  - Mutable
- Immutable List

Collections.

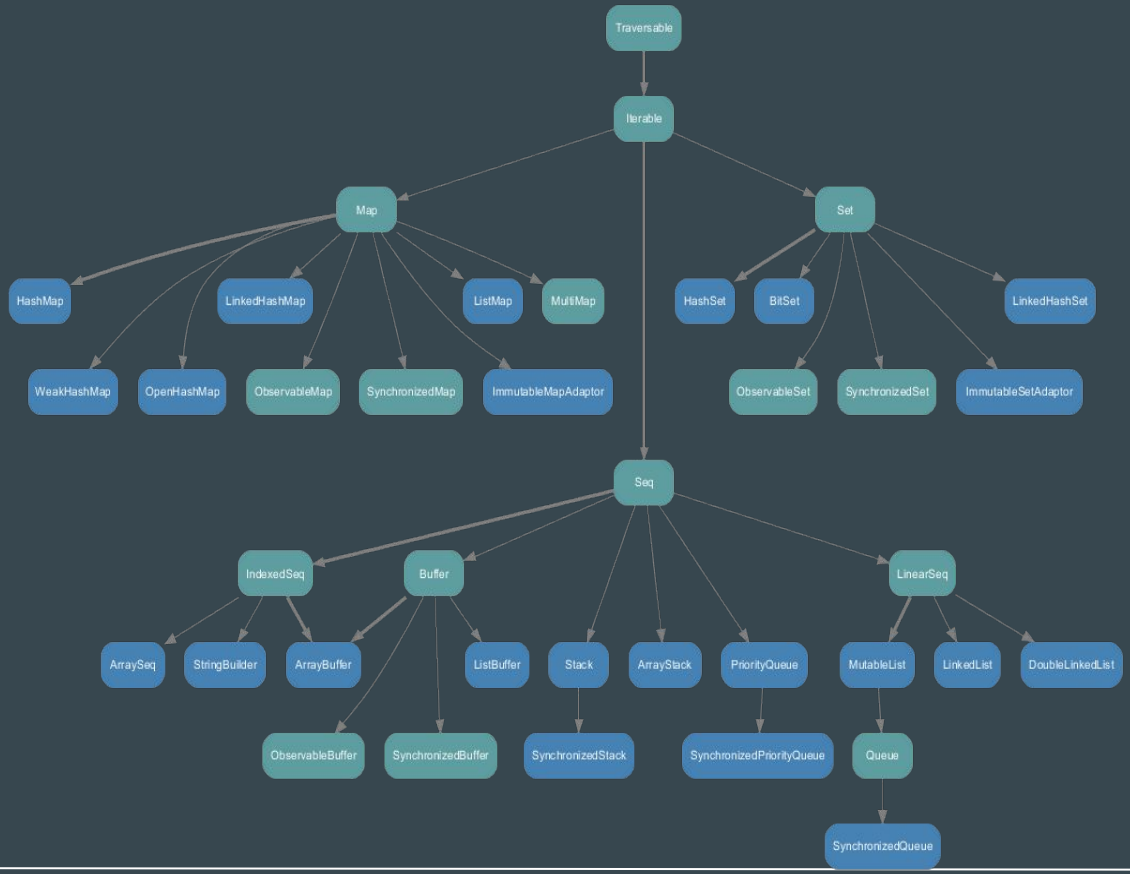
- Scala supports mutable collections and immutable collections.
- A mutable collection can be updated or extended in place. This means you can change, add or remove elements as a side effect.
- Immutable collections never change. You have still operations that stimulate additions, removals, updates by creating a new collection and leave the old unchanged.

Collections Hierarchy



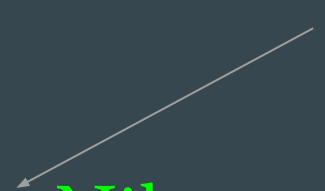
Hierarchy of Immutable Collections

Hierarchy of mutable Collections



Immutable List

```
// Construct a List
val list1 = List(1,2,3)
val list2 = 1 :: 2 :: 3 :: 4 :: 5 :: Nil
val list3 = List("apples", "oranges","bananas")
val list4 = List(5,5,6,6) ::: List(8,7)
```



Immutable List (API)

```
val list = List(4,3,0,1,2)
```

- ➔ list.head
- ➔ list.tail
- ➔ list.length
- ➔ list.max
- ➔ list.min
- ➔ list.sum
- ➔ list.contains(2)
- ➔ list.drop
- ➔ list.reverse
- ➔ list.sortWith(_ > _)
- ➔ list.map(x => x * 2)
- ➔ list.map(_ * 2)
- ➔ list.reduce((x,y) => x + y)
- ➔ list.filter(_ % 2 == 0)
- ➔ list.groupBy(x => x % 2 == 0)

Summary of Scala.

- Keep it simple
- Don’t pack too much in one expression
- Find meaningful names
- Prefer functional
- Careful with mutable objects

Further Reading $ References.

- Scala Official Docs
- Cousera - Martin Odesky
- Creative Scala
- Scala School by Twitter
- Scala 101 by Lightbend

Q & A