Scala Ecosystem

By
ADITYA PRABHAKARA

Introducing Myself

Aditya S P (sp.aditya@gmail.com)
Freelance trainer and technologist

Boring Stuff about me:

- 14+ years of experience in development and training
- Started with Java, moved to Android and now working on Big Data Technologies

Interesting Things about me:

- Actually Nothing !

Getting to know you

Show of hands please!

- What is the general development experience of this group
  - 0-2 years, 0-5 years, 5 and above
- What programming area are you currently working on?
  - Java Programming, .Net, C++
- How many of you are familiar with functional programming?
- How many of you are already familiar with Scala as a programming Language
- How many of you are already familiar with Spark

Agenda

- Scala Programming
- Play framework
- Akka framework
- Kafka

Chapter: Introduction

Scala

- High Level Programming Language
- Multi-paradigm language
  - OO
  - Functional
- You can write code in imperative style, procedural style or functional style
- Created by Martin Odersky

Scala History

- Fairly new programming language
- 2001 - Work on Scala started in 2001
- 2004 – General release on Java Platform and .Net Platform!
- 2012 – Support for .Net platform was dropped
- Typesafe – official Scala support and services company

Why should I love Scala?

Scala is very concise

```
public class HelloWorld
{
        public static void main
(String[] args)        {

        System.out.println(“Hello
world!”);
        }
}
```

```
println(“Hello world!”)
```

Why should I love scala?

Scala is readable and very expressive langugage

Expressive – say a lot in a few words

```
import scala.io.Source
val filename = "fileopen.scala"
for (line <- Source.fromFile(filename).getLines()) { println(line) }
```

How should I learn Scala?

Learning of Scala happens on these three different streams

1. Functional Programming
   - Understanding to deal with a program as functions

2. Object Oriented Programming
   - Understanding to represent stake holders as objects

3. New syntax structures
   - Scala's conciseness comes from a lot of new syntax structures
   - Scala is influenced by a lot of programming languages and has taken the best of a lot of worlds !

Difference between a procedure and a function

- Procedures take in a set of parameters and performs tasks
- No pressure on what a procedure should return
- Can lead to breakage of separation of concerns as SOC is on tasks
- Can lead to spaghetti code

- Functions map a set of parameters to a return value
- High importance to separation of concerns

Side effects

- Why side-effects are troublesome?

```
def summer(x: Int, y: Int){
        println(x + y)
}
```

Chapter: Setup

Scala setup

- Java 8
- Idea
- Scala plugin

Chapter: Hello World!

Hello World

- In scala REPL
- In scala Worksheet
- In scala program

Creating variables

- vals and vars

```
> 1 + 2
res0: Int = 3
> val a = 1 + 2
a: Int = 3
> var b = 1 + 2
b: Int = 3
```

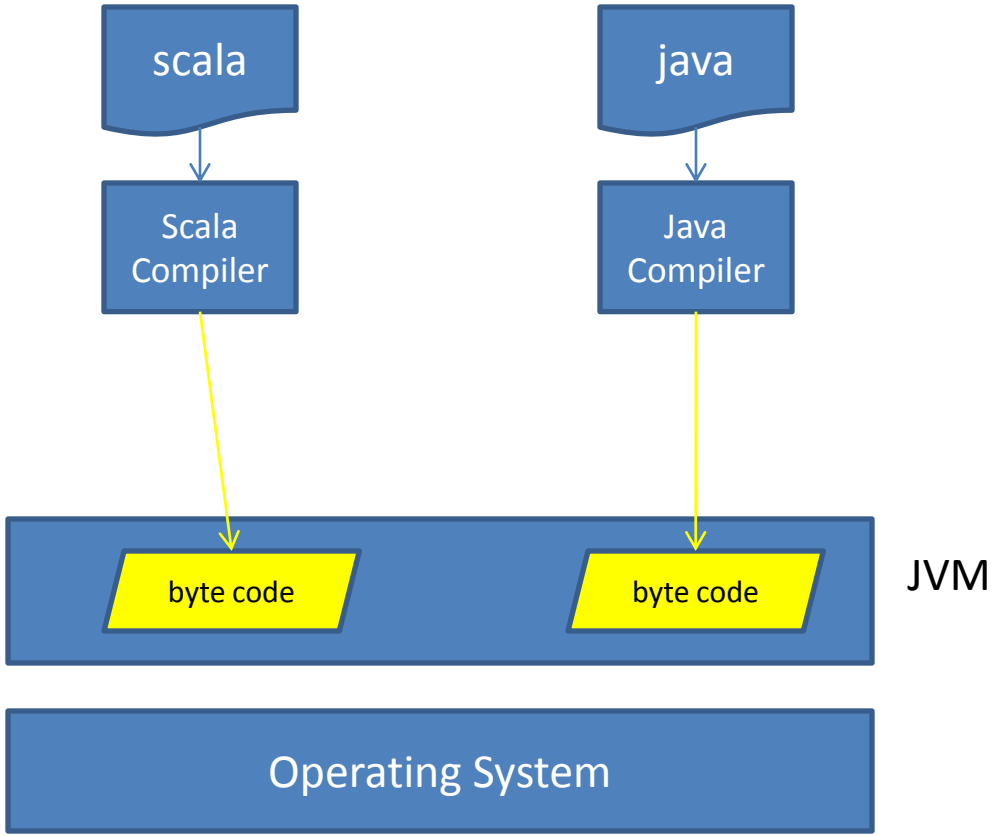
```
> val c :Int = 1 + 2
c: Int = 3
> var d :Int = 1 + 2
d: Int = 3
```

Let us work with a few strings

- Type inference
- So what is java.lang.String doing here?

```
➢val msg1 = "Hello World"
msg1: String = Hello World

> val msg2 :String = "Hello World"
msg2: String = Hello World

> val msg3 :java.lang.String = "Hello World"
msg3: String = Hello World
```

Scala and Java



Blocks

We will revisit this after function definition

```
> val a = 10

> val x = {
  var b = 3
  a + b + 20
}
```

Chapter: Lists and Arrays

Creating an Array

Parameterized Creation or Parameterization=> configuring an instance during the time of creation
Type Parameterization means providing the Type in [ ] brackets
Now the type of value “arr” is Array[Int]

```
val arr = new Array[Int](3)
arr(0)=1
arr(1)=3
arr(2)=300

// reading values of arrays
arr(0)
arr(1)
```

Creating an Array

Why does not array use [] ?
Conceptual simplicity by considering everything as objects with methods!

```
arr(0)=1
arr.update(0, 1)
arr(1)=3
arr.update(1,3)


arr(0)
arr.apply(0)
arr(1)
arr.apply(1)
```

Creating an Array

An easier and a more scala way of creating an array
Use Array as a factory method

```
// using a factory method
val arr = Array(1,2,3,4)

// the above is actually
val arr = Array.apply(1,2,3,4)
```

Creating a List

- Lists are immutable
- Homogeneous

```
val evenList = List(2,4,6,8)
val anotherList = List.range(1,5)
```

Creating a List

- Type inference

```
val evenList = List(2,4,6,8)

// mentioning type
val evenList :List[Int] = List(2,4,6,8)

// error
val evenList :List[Int] = List(2,4,6,8,”Hello”)
```

Creating a List

- Type inference – Answer these questions

```
//why does this work?
val evenList = List(2,4,6,8, “Hello There”)

//If it infers the type of List then what is the type of below val
val emptyList = List()
```

Creating a List

- . infix operator – also called a method!
- So range is a method of List
- Scala has no operator overloading

```
val anotherList = List.range(1,5)

//List.range(1,5) can be written as

List range (1,5)
```

Creating a List

- This is strange
- Its called the cons operator.
- What is “Nil” doing at the end of this?

```
val yetanotherList = 1 :: 2 :: 3 :: 4 :: Nil
val empty = Nil

// Whats happening here?
yetanotherList :+ 3
```

Some List operations

```
//concatenate
val newList = list1 ::: list2

// empty check
empty.isEmpty

//empty check is better than length
//length requires a list traversal to return the value
//hence newList.isEmpty is not same as newList.length == 0
```

Some List operations

```
//take
val li = List(1,2,3,4,5,6)
li.head
li.tail
li.init
li.last
//generalizes init
li.take(3)
//generalizes tail
li.drop(3)
li.indices
li.indices zip li
li.toString
li.mkString(“,”)
```

Some List operations

```
//flatten
val lol = List(List(1,2,3), List(4,5,6))
lol.flatten
```

Differences between Array and List

Performance differences

	Array	List
Access the ith element	O(1)	O(i)
Discard the ith element	O(n)	O(i)
Insert an element at i	O(n)	O(i)
Reverse	O(n)	O(n)
Concatenate (length m,n)	O(n+m)	O(n)
Calculate the length	O(1)	O(n)

Memory differences

	Array	List
Get the first i elements	O(i)	O(i)
Drop the first i elements	O(n-i)	O(1)
Insert an element at i	O(n)	O(i)
Reverse	O(n)	O(n)
Concatenate (length m,n)	O(n+m)	O(n)

Arrays are mutable
Lists are immutable
Arrays are actually Java Arrays
Lists is a Scala datastructure

Chapter: Tuples

What is a tuple

Unlike lists tuple can contain items of different data types
Tuple is not actually a collection
Each tuple will have different data type depending on number and type of parameters.
22 such tuples are available

```
val mytuple = (20, “Twenty”, true)
println(mytuple._1)
println(mytuple._2)
println(mytuple._3)
```

Chapter: Map

Map

Mutable Map and Immutable Map – We will look more into this in while we study collections of Scala

```
val mutMap = mutable.Map[Int, String]()
mutMap.size
mutMap.put(1, "One")
mutMap.put(2, "Two")
mutMap.keys
mutMap += (1 -> "one")

val immMap = Map( 1 -> "One", 2 -> "Two", 3 -> "Three")
```

Chapter: Functions

Functions

- Creating a function
- Function parameters and return types
- Writing functions concisely
- Named parameters
- Variable length parameters
- Default parameters
- Local functions
- Recursion
- Tail Recursive functions
- Function literals
- Function Types
- Higher order functions
- Target typing
- Placeholders
- Closures

Creating a Function

- Use keyword “def”

```
def sayHi(){
  println("Hi")
}

>sayhi()
```

Function returning a value

- Explicitly say what it returns

```
def sayHi() :String = {
  return "Hi"
}

>a = sayhi()
```

Function accepting parameters

- Can explicitly mention of type for type safety or even let scala infer it

```
def sayHi(n :String) :String = {
  return "Hi " + n
}

>a = sayhi()
```

Some syntax reduction

- We have already eliminated “;”

```
def sayHi(n :String) = "Hi " + n
```

Ternary operator

- Write a function which takes in two strings
  and returns the string with maximum characters

```
> var a = "Hi"
> var b = "Hello"
> var c = maxlength(a,b)
c: String = Hello
```

Ternary operator

- Write a function which takes in two strings and returns the string with maximum characters

```
def maxlength(one: String, two:String) :String ={
  if (one.length > two.length)
    one
  else
    two
}
```

```
def maxlength(one: String, two:String) = if (one.length > two.length)  one else  two
```

```
var c = if (a.length > b.length) a else b
```

Revisiting Blocks

What if I change “val” to “def”

```
> var a = 10

> def x = {
  var b = 3
  a + b + 20
}
```

Variable length parameters

* is not a pointer

```
def f1(x :Int*)=println(x)
var myIntArr = Array(1,2,3,4)
f1( Array(1,2,3,4):_* )
```

Named Arguments

Which of the two calls is more readable?

```
def f1(fname :String, lname :String, title :String)={
  title + " " + fname + " " + lname
}

f1("Aditya", "Prabhakara", "Mr")
f1(title="Mr", fname="Aditya", lname="Prabhakara")
```

Default parameters

```
def f1(fname :String, lname :String, title :String ="Mr"){
  title + " " + fname + " " + lname
}

f1("Aditya", "Prabhakara”)
f1(title=“Dr", fname="Aditya", lname="Prabhakara")
```

Recursion

In the slides I will use extra syntax to enhance readability on slides and to avoid line breaks. Writing code can be as concise and expressive as possible

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


OR

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

Repercussions of this recursion

Recursions will use stack frames to remember function calls.
Let manufacture an exception and see what happens.

```
def factorial(x :Int) :Int ={
        if (x == 0) throw new Exception("Manufactured Exception")
        else x * factorial(x-1)
}
```

Optimizations

Tail recursion optimization
But unfortunately, we have changed the signature of the call itself to the end user!

```
def factorial(x :Int, curr :Int) :Int ={
        if (x == 0) curr
        else factorial(x-1, curr * x)
}

factorial(3,1)
```

How do we solve the previous issue

Answer: Local Functions

```
def fact(x:Int): Int ={
  def tailfact(x :Int, curr :Int) :Int = {
    if (x==0) curr
    else tailfact(x-1, curr * x)
  }
  tailfact(x, 1)
}


fact(4)
```

tailrec annotation

```
import scala.annotation.tailrec
@tailrec
def summer(x :Int) :Int = {
  if (x ==0) 0
  else x + summer(x-1)
}

//Gives an error which says:

error: could not optimize @tailrec annotated method factorial: it
contains a recursive call not in tail position
```

Tail recursion – Scala limitations

- Tail recursion optimization in scala is limited due to JVM

- Not always something can be tailrec-ed
  - Examples –
    1. indirect recursion
    2. Partial functions

- Tail call optimizations limited to situations in which a method or nested function calls itself directly as its last operation without going through a function value or some other intermediary

Introducing function type

Introducing “type” of a function
Introducing => . Can be read as “transforms”

```
def sq(x:Int) = x * x
type Functor = (Int) => Int

def ch(x:String, y:Int) = x.charAt(y)
type Functor = (String, Int) => Int
```

Literals

Integer and String literals

```
10
“Hello”
“Hello”.contains(“H”)
```

Function Literals

Writing a function as a literal
Function literal exists in source code . Function value is the function object

```
// Just a function literal. Basically useless as I cannot do anything
else with it
(x :Int) => x+1

// Function literal assigned to a variable. Now it beings to be useful
val a = (x :Int) => x+1
a(3)
```

Passing functions around

As now a function has become a variable, we can pass it around !

```
def greet(msg :String, deco :(String)=>String) = {
  println(deco(msg))
}

val decorator = (msg: String) => {
  "****************\n" + msg + "\n****************\n"
}

greet("Hello", decorator)
```

Write a logger

Write a logger which logs a message with date time format

```
// hint statements for getting date as a string
import java.text.SimpleDateFormat
import java.util.{Calendar, Date}
var sdf :SimpleDateFormat = new SimpleDateFormat("dd-mm-yyyy")

// expected call
logger("Starting the system", datelog)

// expected output
01-03-2017:Hello There
```

Returning a function

```
def f1() :(String)=>Unit ={
  val f2 = (n :String) => {
    println("Hi There" + n)
  }
  f2
}

var myf = f1()
myf("Aditya")
```

Higher order functions

- Either take in a function as an argument
- Or Return function as an argument
- Or Both of the above

- The previous few examples we saw are **higher order functions** !

Other functions are called as **first order functions**

Ability to assign functions to variables and pass it around like just like any other objects is called as
“Treating functions like first-class constructs”
or “First class functions”
or “Treating functions as first-class citizens”

Syntax conciseness – “target typing”

```
type Functor = (Int)=>Int
def nextinseries(x:Int, inc :Functor) = inc(x)
nextinseries(2,(x :Int)=> x + 2)

// target typing
nextinseries(2, (x)=>x +2)

// remove parenthesis of type inferred argument
nextinseries(2, x=>x+2)
```

Syntax conciseness – “Placeholder”

```
type Functor = (Int)=>Int
def nextinseries(x:Int, inc :Functor) = inc(x)
nextinseries(2,(x :Int)=> x + 2)

// target typing
nextinseries(2, (x)=>x +2)

// remove parenthesis of type inferred argument
nextinseries(2, x=>x+2)

// nextinseries(2, _ + 2)
```

Closure – It gets crazy here!

```
def f1(n: String) :() => Unit ={
  val f2 = () => {
    println("Hi There" + n)
  }
  f2
}

var myf = f1("Aditya")
myf()
```

Closure – A more serious example!

```
type F = (Double)=>Double

def bill(i1 :Double, i2 :Double, i3 :Double) :F ={

  def withtax(tax :Double) :Double =
        i1 + i2 + i3 + tax * (i1 + i2 + i3)

  withtax
}

val bill1 = bill(10,20,30)
val bill2 = bill(2,3,4)
bill1(.145)
bill2(.05)
```

Closure – Why the name closure

- Function value (the object) that gets returned is called a closure
- Called a closure as it captures a free variable
- (x :String) => x.toUpperCase() is called a closed “term”
- Where “term” is the code itself

Functions

- Creating a function
- Function parameters and return types
- Writing functions concisely
- Named parameters
- Variable length parameters
- Default parameters
- Local functions
- Recursion
- Tail Recursive functions
- Function literals
- Function Types
- Higher order functions
- Target typing
- Placeholders
- Closures

Chapter: H.O.Functions with Lists, Maps etc

Some List operations – H.O. methods

```
//map
val loi = List.range(1,10)
loi.map(_ * 3)

val lol = List(List(1,2,3), List(4,5,6))
lol.flatMap(x => {println(x)
  x.map(_ * 20)
})

//foreach
var sum = 0
loi.foreach(sum += _)
```

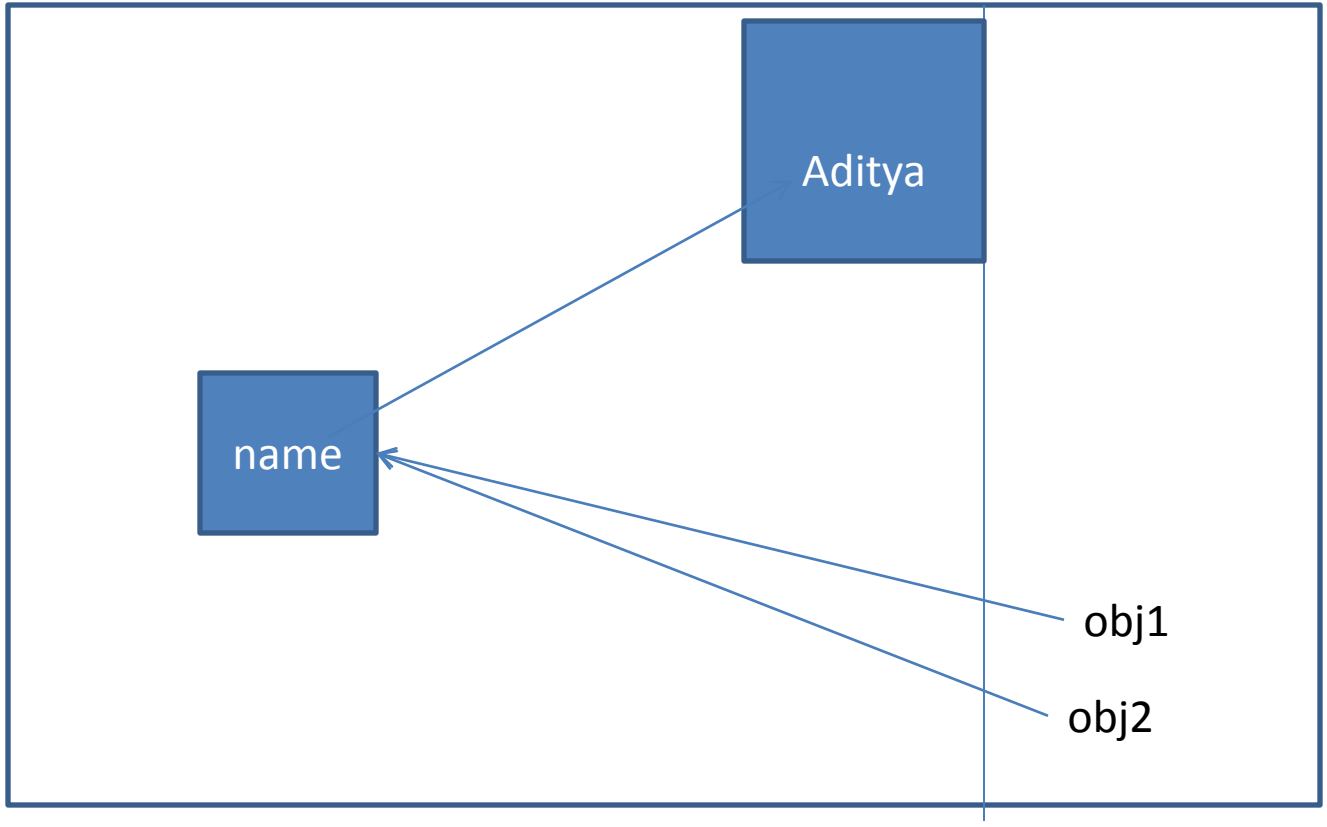
Some List operations

```
//filter
val loi = List.range(1,10)
loi filter (_%2 ==0)

var a, b= loi partition (_%2==0)
//
var op = loi find (_%100==0)
op.get

val loi = List.range(1,10)
loi.forall( _< 11 )
loi.exists( _ == 7 )
```

Working example

```
import scala.io.Source._
val lines =
fromFile("D:\\scalalearning\\March6\\src\\main\\resources\\data.txt").
getLines
val linelist = lines.toList
```

Chapter: Control Structures


Aditya
name
obj1
obj2

While loop

```
var count = 0
while (count <10){
  print(count)
  count +=1
}
```

For expressions

Very expressive
More functional as it works on iterators
Can be used in comprehensions -> concise way of building iterations

```
for( i <- 1 to 3)
    println("Looping through" + i)

for( i <- 1 until 3)
  println("Looping through" + i)
```

For expressions

```
val arr = Array(1,2,3,4)
val mylist = List(1,2,3,4,5)
for (i <- arr)
    println(i)
for (i <- mylist)
    println(i)
```

For expressions - yield

```
val arr = Array(1,2,3,4)
for (i <- arr if i %2 == 0)
    yield i+1

val trial = List(1,2,3,4)
for (i <- trial)
  yield i + 1
```

Chapter: OOP

Creating a class

```
class Student{
  var name = "NA"
}

val s1 = new Student()
s1.name = "Aditya"
s1.name
val s2 = new Student
```

Creating a class

```
class Student{
  private var name = "NA"
  def getName(): String={
    name
  }
  def setName(n :String) = {
    name = n
  }
}

val s1 = new Student()
s1 setName "Aditya”
```

Printing it pretty

```
class Student{
  private var name = "NA"
  def getName(): String={
    name
  }
  def setName(n :String) = {
    name = n
  }
  override def toString()={

  }
}

val s1 = new Student()
s1 setName "Aditya”
```

Printing it pretty

```
class Student{
  private var name = "NA"
  def getName(): String={
    name
  }
  def setName(n :String) = {
    name = n
  }
  override def toString = "Student[name-> " + name + "]"
}

val s1 = new Student()
s1 setName "Aditya"
s1
```

Basic flaw

After creation, leaves the object in an unstable state !

```
val s1 = new Student()
s1
// the name of student is “NA” which doesn’t make sense. There cannot
// be a student with “NA” as the name!
```

Parameterize object creation

```
class Student(name :String) {
  println("Created student " + name)
  override def toString = "Student[name-> " + name + "]"
  def getName()=name
}

val s1 = new Student("Aditya")
s1.getName
```

All good but there is a problem!

```
class Student(name :String, marks :Double) {
  println("Created student " + name)
  override def toString = "Student[name-> " + name + "]"
  def getName()=name
  def compare(x :Student) :Int= {
    if (marks > x.marks) 1 else 0
  }
}

val s1 = new Student("Aditya", 90.0)
val s2 = new Student("Arun", 89.0)
s1.compare(s2)
```

Solution

```
class Student(name :String, marks :Double) {
  private val n = name
  private val m = marks
  override def toString = "Student[name-> " + name + "]"
  def getName()=this.n
  def >(x :Student) :Boolean= {
    if (this.m > x.m) true else false
  }
}

val s1 = new Student("Aditya", 90.0)
val s2 = new Student("Arun", 89.0)
s1.>(s2)
s1 > s2
```

Building in robustness

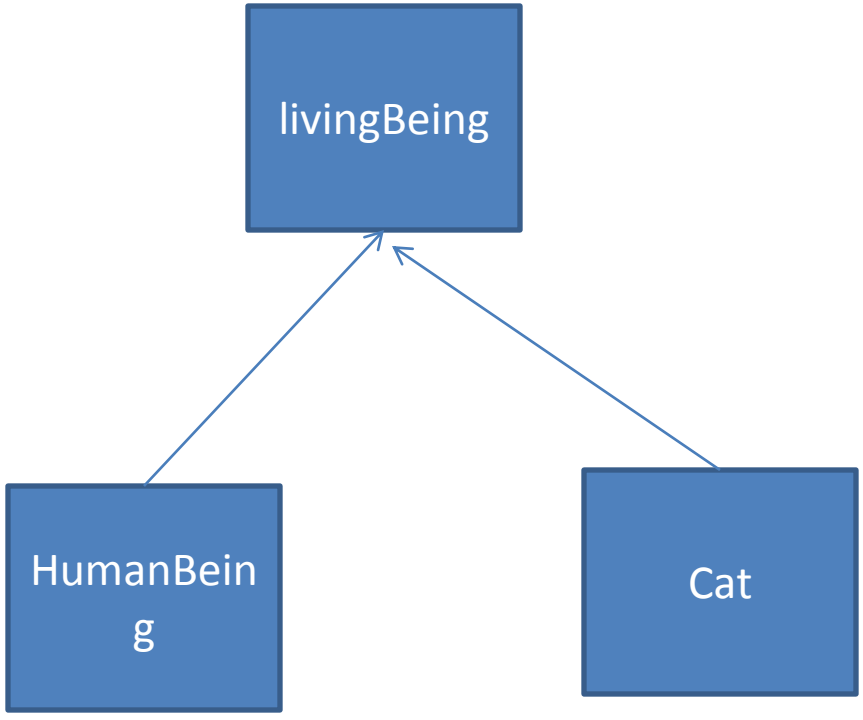
```
require( marks > 60.0)
require( name.length > 4)
```

Auxiliary constructor

```
class Student(name :String, marks :Double, gender :String) {
…
…
  def this(name :String, marks :Double) = this(name, marks, “Male”)
…
…
}
```

Create class for complex numbers

```
val cmp1 = new Complex(1,2)
//cmp1 Should print 1 + 2j
val cmp2 = new Complex(3,4)
//cmp2 Should print 3 + 4j

//Supported methods
Add two complex numbers (1 + 5j) + (3+6j) = 4 + 11j

Multiply a complex number with an integer
(1 + 5j) * 3 = 3 + 15j
```


livingBeing
HumanBeing
Cat

Chapter: Inheritance and Composition

Inheritance

A way of reusing code
A way bunching together classes of same type

```
class A {
   val value = 10
}
class B extends A {

}
```

An example

A way of reusing code
A way bunching together classes of same type

```
class LivingBeing {
   def breathe() = {
        println("I am breathing therefore I a living")
   }
}
class HumanBeing extends LivingBeing {

}
```

Type

A way of reusing code
A way bunching together classes of same type

```
class A {

}
class B extends A {

}
class C extends A {

}
```

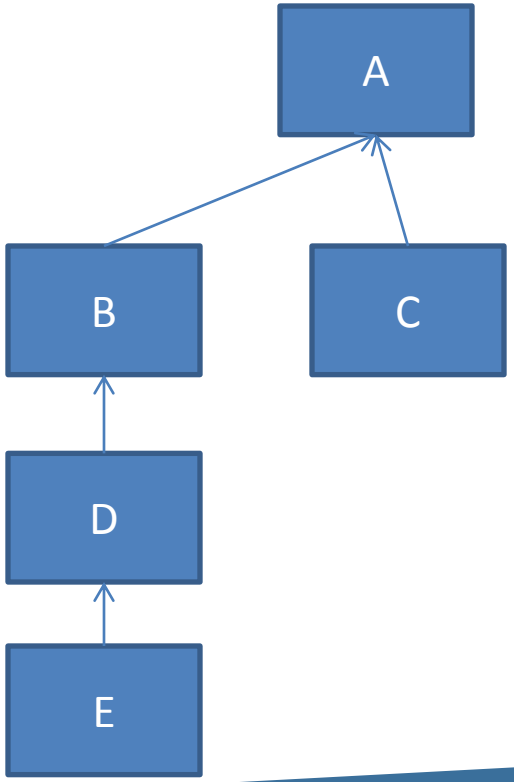
Override

Breathe through lungs!

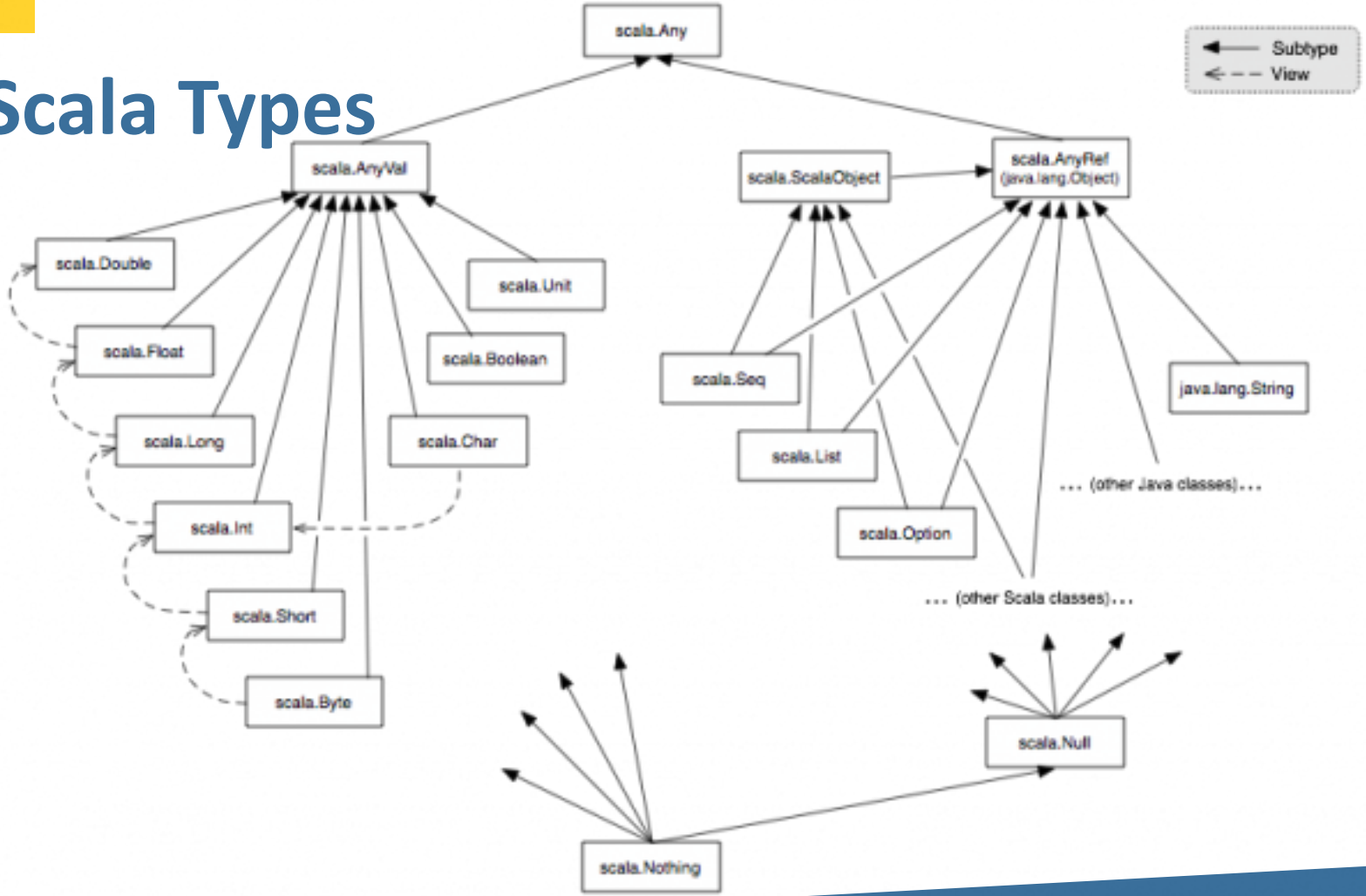
```
class LivingBeing {
   def breathe() = {
        println("I am breathing therefore I a living")
   }
}

class HumanBeing extends LivingBeing {
   override def breathe() = {
        println("Breathe through lungs")
   }
}
```

Class hierarchy

```
class A { val a = "A" }
class B extends A { val b = "B" }
class C extends A { val c = "C" }
class D extends B { val d = "D" }
class E extends D { val e = "E" }
```



Scala Types


scala.Any
Subtype
View
scala.AnyVal
scala.ScalaObject
scala.AnyRef
(java.lang.Object)
scala.Double
scala.Unit
scala.Float
scala.Boolean
scala.Long
scala.Char
scala.Int
scala.Short
scala.Byte
scala.Seq
java.lang.String
scala.List
... (other Java classes)...
scala.Option
... (other Scala classes)...
scala.Null
scala.Nothing

Chapter: Traits

Traits

Ability to add behaviour

```
class LivingBeing {

}
Class HumanBeing extends LivingBeing {

}
Class Fish extends LivingBeing {

}
Class Dog extends LivingBeing {
}
```

Traits

Ability to add behaviour

```
class LivingBeing {

}
Class HumanBeing extends LivingBeing with Metathink with MakeSound{

}
Class Fish extends LivingBeing {

}
Class Dog extends LivingBeing with Metathink with MakeSound {
}
```

Chapter: Singleton Objects

Singleton objects

Analogous to static of java.

```
Object MyUtils{
  def toMyArray(s :String) = s.split(“,”)
}
```

Companion objects

An object associated with a class

```
class MyUtils{
}

Object MyUtils{
  def toMyArray(s :String) = s.split(“,”)
}
```

Chapter: Pattern matching

Match Expression

- A java's switch case can be easily replaced by a scala's "**match expression**"
- It is an expression meaning always results in a value
- In scala, expression do not fall through. (compare with – if a break is missing in a switch statement, the control flows into next case)
- If no match is found, then it raises a MatchError

```
// Constant pattern
def getErrorName(status :Int): String = status match {
  case 404 => "Page not found"
  case 501 => "Internal server error"
  case 202 => "OK"
  case _ => "Unknown error"
}
getErrorName(404)
```

Match Expression – with variables

- Be careful with variable names in match patterns
- As variable names can have any value Scala would not know how to match
- If you wanted to really match then use ` (back tick) to surround the variable name

```
// The below will fail

val a = 10
val b = 20

20 match {
  case a => "value is 10"
  case b => "value is 20"
  case _ => "Value is something
else"
}
```

```
// The below will succeed

val a = 10
val b = 20

20 match {
  case `a` => "value is 10"
  case `b` => "value is 20"
  case _ => "Value is something
else"
}
```

Match Expression – Constructor Patterns

- _ in a constructor will match anything and ignore
- Can match deep patterns Car(_, Tyre(radial))

```
case class Indian(fname: String, lname :String)

def stereoType(indian: Indian): String = indian match {

  case Indian(_, "Singh") => "May be from Punjab"
  case Indian(_, "Kulkarni") => "May be from Maharashtra"
  case Indian(_, "Roy") => "May be from Bengal"

}

stereoType(Indian("Harbajan",  "Singh"))
stereoType(Indian("Manoj", "Kulkarni"))
```

Match Expression – Typed Patterns

- _ in a constructor will match anything and ignore
- Can match deep patterns Car(_, Tyre(radial))

```
def getsize(x :Any) = x match {
  case s :String => s.length
  case m :Map[_,_] => m.size
  case l :List[_] => l.length
  case _ => -1
}
```

Match Expression – Sealed classes

```
sealed abstract class Vehicle
case class Suv(tyres: Int) extends Vehicle
case class Bike(tyres: Int) extends Vehicle
case class Auto(tyres: Int) extends Vehicle

def somematch(v :Vehicle) = v match {
  case c: Suv => "Is a car"
}
somematch(Suv(2))
```

Match Expression – Variable binding

```
//variable binding
case class Tyre(name :String)
case class Car(name: String, tyre :Tyre)

Car("XUV", Tyre("tubeless")) match {
  case Car(_, t @ Tyre("tubeless")) => t
  case Car(_, t @ Tyre("radial")) => t
}
```

Match Expression – Pattern Guard

- Try running the below and deliberate on what is happening

```
var l = List((1,2), (2,2), (3,3))

l.foldLeft(List[(Int,Int)]())((x,y) => {
  y match {
    case (a,b) if a ==b => (a,b) :: x
    case _ => x
  }
})
```

Option type

- Standard type. Such a type can have two values Some[A] or None

```
var x = Map(1->"one", 2->"two")

x.get(1) match {
  case Some(x) => x
  case _ => 0
}

//How to solve the below issue?

val mylist = List("1","2","3","")
mylist.map(_.toInt)   // this will fail because it cannot convert "" to int
```

Chapter: case classes

case classes

Java pojos!

```
case class Student (name :String, marks :Int)
```

Setup

Setup

- Download Java
- Set JAVA_HOME
- Download scala 2.11
- Install scala
- Set SCALA_HOME
- Download spark
- Extract libraries
- Set Spark Home
- Download winutils

https://github.com/steveloughran/winutils

- Set Hadoop home
- Give file permission
- winutils.exe chmod 777 D:\tmp\hive

Starting spark shell

```
spark-shell
(0 to 1).toDF.show
sc.parallelize(1 to 9)
```

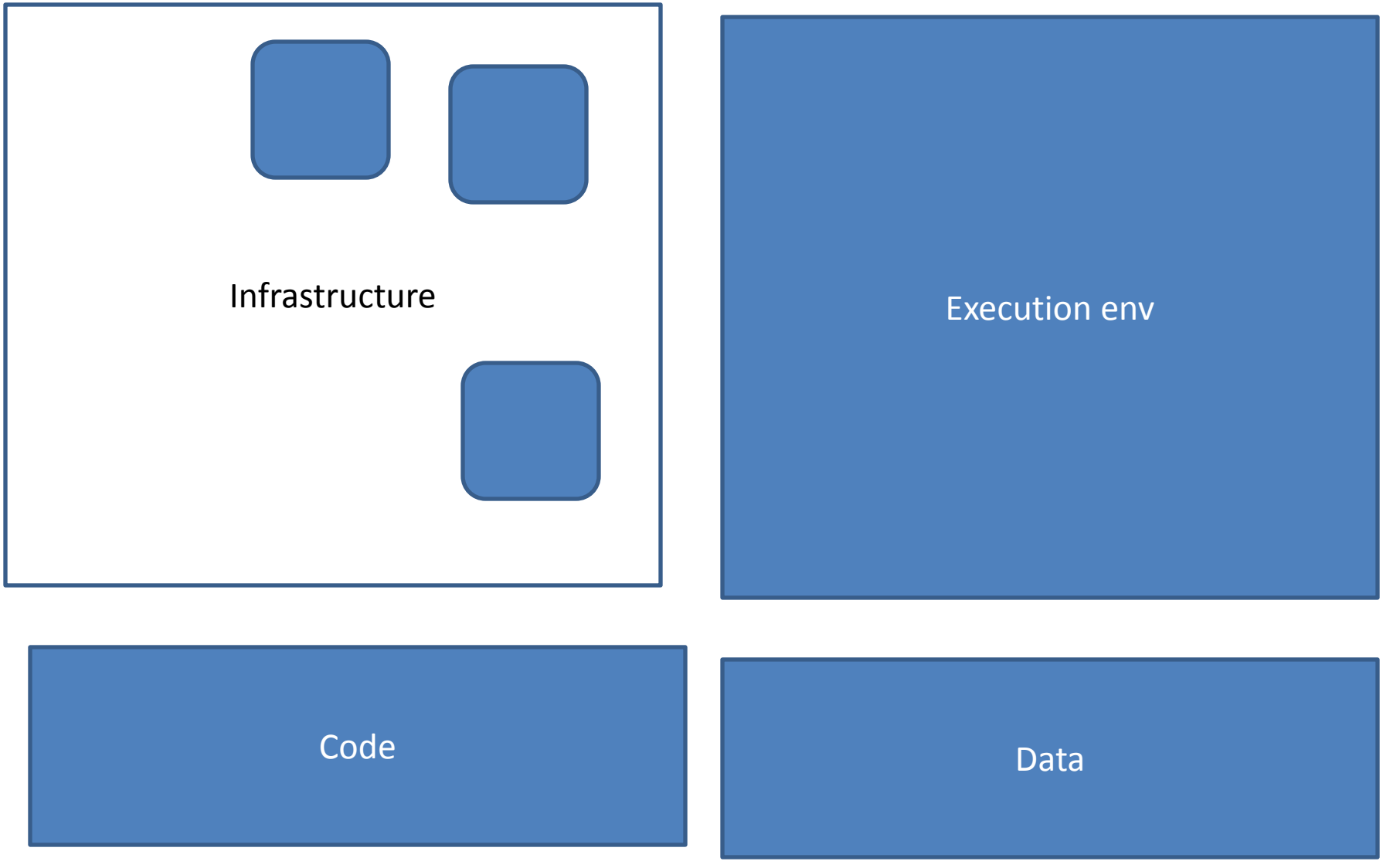
Configure IntelliJ to run a spark app

Starting Spark master and worker

```
spark-class2.cmd org.apache.spark.deploy.master.Master

spark-class2.cmd org.apache.spark.deploy.worker.Worker spark://12.1.1.1:7077
```

Submitting a job

```
spark-sumbit2.cmd --class org.apache.spark.examples.SparkPi --master
spark://12.1.1.1:7077 examples\jars\spark-examples_2.11-2.1.0.jar
```


Infrastructure
Execution env
Code
Data

Including external jars

```
spark-shell --jars <>
```

Including external jars

```
spark-shell --jars <>
```