INTRODUCTION TO SPARK WITH SCALA

Introduction to Scala

- Scala Overview
- Functional Programming Overview
- Why Scala?
  - Compatible
  - Concise
  - High level
  - Statically typed
- Scala Basics
- Scala Ecosystem

Scala Overview

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

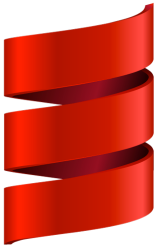
- James Gosling
  - Father of Java

Scala Overview

I can honestly say if someone had shown me the Programming Scala book back in 2003, I'd probably had never created Groovy

- James Strachan
  - Creator of Groovy

Scala Overview



2004

A general-purpose programming language that runs on the JVM.

Concise, elegant, and type-safe.

1995

Scala Overview

+

λ

Object+Data

Pure Functions

Scala unifies object-oriented and functional programming in a statically typed language

Scala Overview

- Scala is more object-oriented than Java
  - Every value is an object (no primitives)
    - No primitive types (int, long, boolean, etc)
  - Every operation is a method call (no static methods)
  - Composition - traits & mixin

```
1 + 2  ==> 1.+(2)
```

Scala Overview

- Scala is functional
  - Full blown functional language
  - Lisp Scheme, SML, Erlang, Haskell, Ocaml, F#
  - First-class functions and efficient immutable data structures

What is function programming?

Scala Overview

Functional programming is a programming paradigm that models computation as the evaluation of expressions, which are built using functions that don't have mutable state and side effects

"Functional programming is programming with functions"

Functional Programming Overview

- Function programming
  - Functions are first class citizens
    - A value just like integer or string
    - High-order functions
      - Take one or more functions as input or return a function
    - Useful for abstracting over operations and creating new control structures
  - Functions should have not have any side effects
    - The output value of a function depends on only its inputs
      - `y = sin(x)`
    - **Easier to reason about, understand and test**

```
def currentTime = Calendar.getInstance().getTime
```

Functional Programming Overview

```
// assign a function to variable
val inc = (x : Int) => x + 1
inc(7)

// passing a function as parameter
(1 to 5) map (inc)  ==> (2,3,4,5,6)

// take even number, multiply each value by 2 and sum them up
(1 to 7) filter (_ % 2 == 0 ) map (_ * 2) reduce (_ + _)

// longer version
(1 to 7) filter (x => x % 2 == 0 ) map (x => x * 2) reduce
((x,y) => x + y)
```

Functional Programming Overview

- Functional language features
  - Higher-order functions
  - Lexical closures
  - Pattern matching
  - Lazy evaluation
  - Type inference
  - List comprehensions
  - Tail call optimization

Why Scala?

- Runs on the JVM
  - Compile down to JVM byte codes
  - Call Java methods, access fields, inherit from Java class
  - Scala code can be invoked from Java code
  - Harness all the benefits of the JVM
- Concise
  - Half the number of lines of the same Java program
  - Less typing, less effort to read, understand
  - Few possibilities for defects

Why Scala?

```
// Java
class Course {
   private String name;
   private int number;

   public Course(String name, int number) {
        this.name = name;
        this.number = number;
   }
}
```

```
// Scala
class Course(name:String, number:Int)
```

Programming in Scala

Why Scala?

- High-level
  - Raise the level of abstraction

```
// Java
boolean hasUpperCase = false;
String name = “Scala”
for (int i = 0; i < name.length(); i++) {
    if (Character.isUpperCase(name.charAt(i))) {
         hasUpperName = true;
         break;
    }
}
```

```
// Scala
val name = “Scala”
val hasUpperCase = name.exists(_.isUpper)
```

Programming in Scala

Why Scala?

- Statically typed
  - Classify variables and expressions based on values they hold and compute
  - Parametric polymorphism – generic programming
  - Support type inference to avoid verbosity
    - `val name = “Hien Luu”`
    - `val courses = new HashMap[Int, String]()`
  - Flexibility through pattern matching
    - Generalization of switch statement to class hierarchies

Why Scala?

- Scalable and extensible
  - Designed to grow and scale with demand

```
var i = 10

loopWhile(i > 0) {
  println(i)
  i -= 1
}
```

```
def loopWhile(cond: => Boolean)(f: => Unit) : Unit = {
    if (cond) {
        f
        loopWhile(cond)(f)
    }
}
```

Leveraging closure and function as object

Scala Basics

- Scala HelloWorld

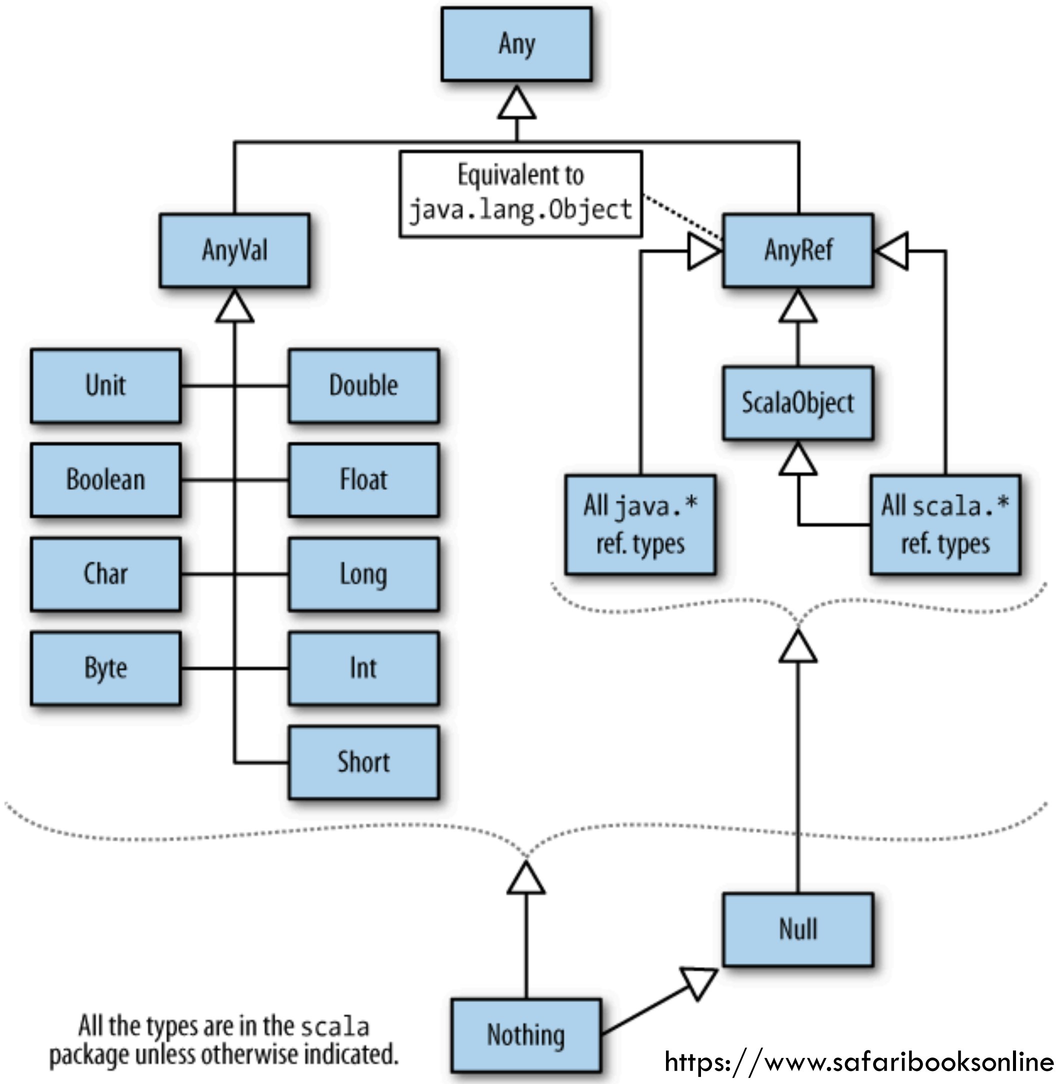
```
object HelloWorld {
   def main(args: Array[String]) {
      println("Hello World!!!")
   }
}
```

- **`object`** declares singleton object
- Static members don't exist in Scala
- Semicolons are options

```
➢ scalac HelloWorld.scala

➢ scala –classpath . HelloWorld
```

Scala Basics



https://www.safaribooksonline.com/library/view/programming-scala

Scala Basics

- Basic types
  - Byte, Short, Int, Long, Float, Double, Boolean, Char
- Define variables

```
// mutable variables
var counter:Int = 10
var d = 0.0
var f = 0.3f

// immutable variables
val msg = "Hello Scala"
println(msg)
s"Greeting: $msg"

val ? = scala.math.Pi
println(?)
```

Scala Basics

- String Interpolation
  - Allow embedding variable references

```
val course = "Spark With Scala"
println(s"I am taking course $course.")

// support arbitrary expressions
println(s"2 + 2 = ${2 + 2}")

val year = 2015

println(s"Next year is ${year + 1}")
```

Scala Basics

Looping Constructs

```
var i = 0
do {
  println(s"Hello, world #$i")
  i = i + 1
} while (i <= 5)

for (j<- 1 to 5) {
  println(s"Hello, world #$j")
}

// what will be printed?
for (i <- 1 to 3) {
  var i = 2
  println(i)
}
```

Scala Basics

□ Defining functions

```
def hello(name:String) : String = { "Hello " + name }
```

```
def hello1() = { "Hi there!" }
def hello2() = "Hi there!"
def hello3 = "Hi there!"
```

```
def max(a:Int, b:Int) : Int = if (a > b) a else b

max(4,6)
max(8,3)
```

Scala Basics

- Function literals

```
(x: Int, y: Int) => x + y
val sum = (x: Int, y: Int) => x + y
sum(1,70)

val prod = (x: Int, y: Int) => x * y
```

```
def doIt(msg:String, x:Int, y:Int, f: (Int, Int) => Int) = {
  print(msg + f(x,y))
}

doIt("sum: ", 1, 80, sum)

doIt("prod: ", 2, 33, prod)
```

Scala Basics

- Tuple
  - Light weight immutable data structure
    - Contains 1 to 22 elements
  - Each element can be of different type
  - Useful for returning multiple objects from a function

```
val pair = ("Scala", 1)
println(pair._1)  // "Scala"
println(pair._2)  // 1

val pair2 = ("Scala", 1, 2015)
println(pair._3)  // 2015
```

Scala Basics

- Class

```
// constructor with two private instance variables
class Movie(name:String, year:Int)

// With two getter methods
class Movie(val name:String, val year:Int)
val m1 = new Movie("100 days", 2010)
println(m1.name + " " + m1.year)

// With two getter and setter methods
class Movie(var name:String, var year:Int)
val m2 = new Movie("100 days", 2010)
m2.name = "100 Hours"
```

Scala Basics

- Constructor Overloading

```
class Movie(name:String, year:Int) {
  // ????
}
```

Scala Basics

- Case class
  - Good for immutable data holding objects
  - Auto generate toString, equals, and hashCode
  - Decompose using pattern matching

```
case class Movie(name:String, year:Int)

val m = Movie("100 days", 2010)
m.toString   // Movie(100 days,2010)

println(m.name + " " + m.year)
```

Scala Basics

- Pattern matching
  - Similar to Java switch statement in Java
  - Useful for extracting data from data structure

```
def errorMsg(n:Int) = n match {
   case 1 => println("Not a problem")
   case 2 => println("You may want to double check")
   case 3 => println("System is shutting down")
}

def range(n:Int) = n match {
  case lessThan10 if (lessThan10 <= 10) => println("0 .. 10")
  case lessThan50 if (lessThan50 <= 50) => println("11 .. 50")
  case _ => println("> 50")
}
range(8)  // "0 .. 10"
range(25) // "11 .. 50"
```

Scala Basics

- Pattern matching with Case class
  - Extracting value out of case class to use

```
abstract class Shape
case class Rectangle(h:Int, w:Int) extends Shape
case class Circle(r:Int) extends Shape

def area(s:Shape) = s match {
  case Rectangle(h,w) => h * w
  case Circle(r) => r * r * 3.14
}

println(area(Rectangle(4,5)))

println(area(Circle(5)))
```

Scala Basics

- Working with Array
  - Mutable flat data structure

```
val myArray = Array(1,2,3,4);

myArray(0)                        // 1
myArray(0) = myArray(1) + 1;
myArray(0)                        // 3


myArray.foreach(a => print(a + " "))
myArray.foreach(println)
```

Scala Basics

- Working with List
  - Immutable and recursive data structure
  - Designed for functional programming style

```
val l = List(1,2,3,4);
l.foreach(println)

println(l.head)  // 1
println(l.tail)  // List(2,3,4)
println(l.last)  // 4
println(l.init)  // List(1,2,3)

val table: List[List[Int]] = List (
       List(1,0,0),
       List(0,1,0),
       List(0,0,1)
)
```

Scala Basics

- Working with List

```
val list = List(2,3,4);

// cons operator – prepend a new element to the beginning
val m = 1::list

// appending
val n = list :+ 5

// to find out whether a list is empty or not
println("empty list? " + m.isEmpty)

// take the first n elements
list.take(2) // List(2,3)

// drop the first n elements
list.drop(2) // List(4)
```

Scala Basics

- High-order methods
  - Transforming every element in a list in some way
  - Verifying whether a certain property holds
  - Extracting elements that satisfy certain condition
  - Combining elements in a list using some operator
- `map, flatMap, foreach, filter, partition, find, takeWhile, dropWhile, span, forall, exists`

Scala Basics

Operation	Description
`map(f)`	Apply function f to each element and return a new list
`reduce(f)`	Reduce the elements using associative binary operator
`flatMap(f)`	Apply function f to each element and return the concatenation of all function result
`foreach`	Apply the procedure to each element
`filter(p)`	Return list of elements which p(x) is true
`partition(p)`	Return a pair of lists – one for p(x) == true, and p(x) == false
`contains(e)`	Whether list contains the given element
`takeWhile(p)`	Take the longest prefix that satisfies p(x) == true
`dropWhile(p)`	Remove the longest prefix that satisfies p(x) == true
`forall(p)`	Return true if all elements satisfies p(x) == true
`exists(p)`	Return true if one of the elements satisfies p(x) == true

Scala Basics

```
val n = List(1,2,3,4)
val s = List("LNKD", "GOOG", "AAPL")
val p = List(265.69, 511.78, 108.49)

var product = 1;
n.foreach(product *= _)  // 24

n.filter(_ % 2 != 0)     // List(1,3)
n.partition(_ % 2 != 0)  // (List(1,3), List(2,4))

n.find(_ % 2 != 0)       // Some(1)
n.find(_ < 0)            // None

p.takeWhile(_ > 200.00)  // List(265.69, 511.78)
p.dropWhile(_ > 200.00)  // List(108.49)

val p2 = List(265.69, 50.11, 511.78, 108.49)
p2.span(_ > 200.00)      // (List(265.69),List(50.11, 511.78,108.49))
```

Scala Basics

```
val n = List(1,2,3,4)
val s = List("LNKD", "GOOG", "AAPL")


n.map(_ + 1)             // List(2,3,4,5)
s.flatMap(_.toList)      // List(L,N,K,D,G,O,O,G,A,A,P,L)

n.reduce((a,b) => { a + b} )  ==> 10

n.contains(3)  ==> true
```

Scala Basics

- Pattern matching with List

```
val n = List(1,2,3,4)
val s = List("LNKD", "GOOG", "AAPL")

def sum(xs: List[Int]) : Int = xs match {
  case Nil => 0
  case x :: ys => x + sum(ys)
}

val dups = List(1,2,3,4,6,3,2,7,9,4)

// challenge
def removeDups(xs : List[int]) : List[Int] = xs match {
}
```

Scala Basics

- Interoperability with Java
  - Classes from java.lang package imported by default
  - More flexible import statement

```
import java.util.{Date, Locale}
import java.text.DateFormat
import java.text.DateFormat._

object ScalaJava {
    def main(args: Array[String]) {
        val now = new Date
        val df = getDateInstance(LONG, Locale.FRANCE)
        println (df format now)
    }
}
```

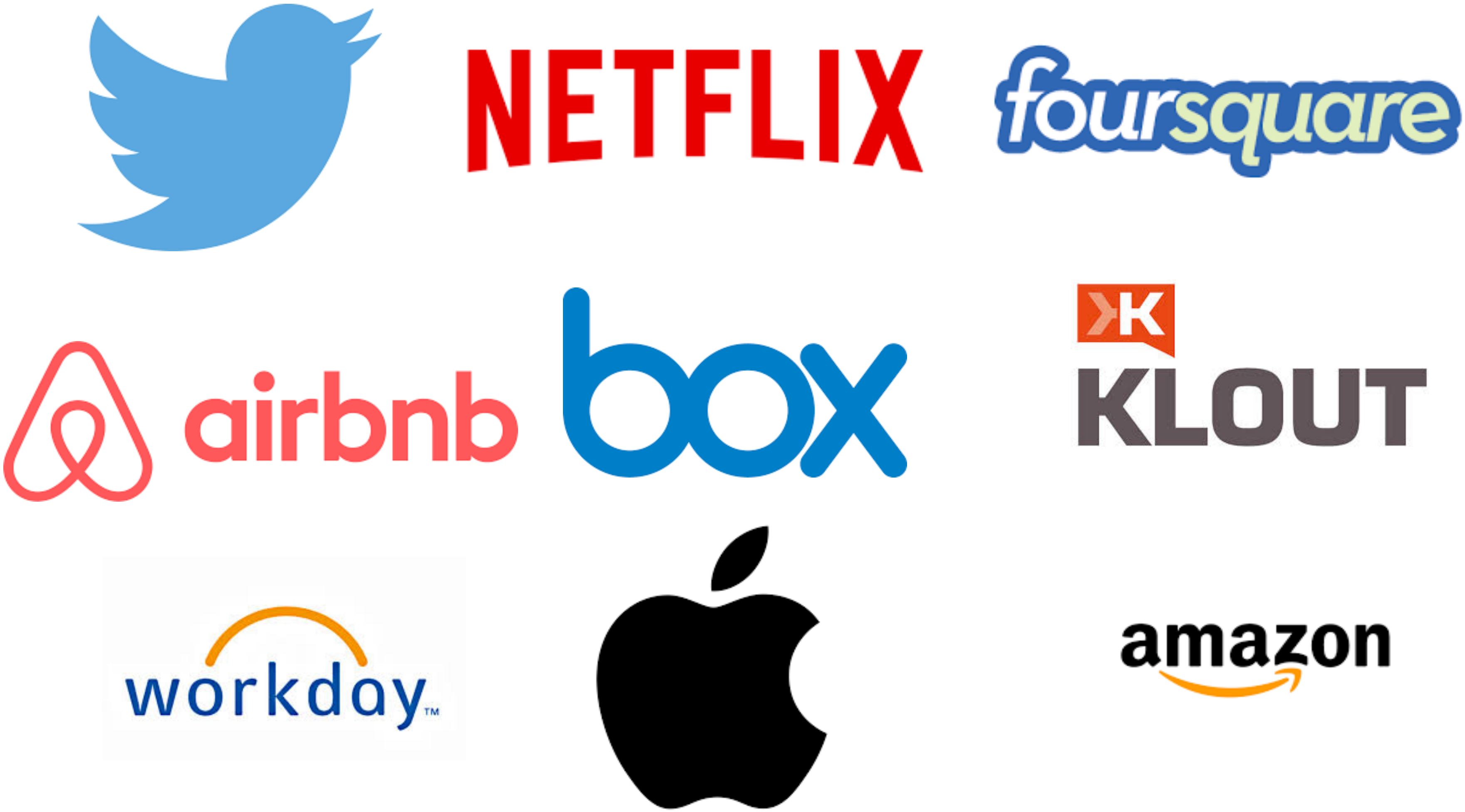
Scala Basics

- Traits
  - Unit of code reuse
  - Similar to Java interfaces
  - Similar to Java abstract classes w/o constructor
    - Can maintain state
  - Multiple inheritance w/o the dangers and limitations
  - A class can mix in any number of traits

Scala Ecosystem

- Play framework
  - Modern web application framework
  - Developer friendly and scalable web applications
- Akka
  - Toolkit and runtime for highly concurrent, distributed, fault tolerant applications
- Slick frameworks
  - Functional relational mapping
  - Entities and queries are statically checked at compile-time

Scala Adopters


NETFLIX
foursquare
airbnb
box
K
KLOUT
workday™
amazon