Introduction to

Scala

Hamed Khashechi, Hossein Rimaz

K.N. Toosi University of Technology

Table of Contents

- Introduction
- Getting Started
- Features of Language
- Programming Environment
- Demo (Word Count with Spark, Simple Web App using Play, MongoDB, ReactiveMongo)
- Resources
- Q&A

Introduction

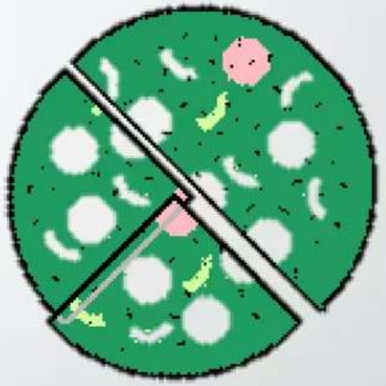
- History
- Why Scala
- Scala Ecosystem and Use Cases

History

The design of Scala started in 2001 at the (EPFL) by Martin Odersky. Odersky formerly worked on Generic Java, and javac, Sun's Java compiler.

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Martin decided take some of the ideas from functional programming and move them into the Java space. He has created a language called **Pizza**.



Pizza's initial distribution was in 1996, a year after Java came out. It was moderately successful in that it showed that one could implement functional language features on the JVM platform.

Then He got in contact with Gilad Bracha and David Stoutamire from the Sun core developer team.

They said, "We're really interested in the generics stuff you've been doing; let's do a new project that does just that.“

And that became GJ (Generic Java). So we developed GJ in 1997/98, and six years later it became the generics in Java 5, with some additions that we didn't do at the time.

In the generics design, there were a lot of very, very hard constraints.

The strongest constraint, the most difficult to cope with, was that it had to be fully backwards compatible with ungenerified Java.

That's why there were a number of fairly ugly things.

Like type eraser, raw types, unchecked warnings, … .

- Pizza (Odersky and Wadler 96) was another language on the JVM that added functional elements to Java:
  - algebraic datatypes and pattern matching
  - function values
  - generics

- Scala was more ambitious:
  - More innovation on the OOP side
  - More functional, e.g. immutable values, by-name parameters,
  - Better integration of functional/oop, e.g. case classes.
  - Not backwards compatible with Java

May 2011, **Odersky**

launched **Typesafe** Inc.

Currently named
Lightbend Inc.

Why Scala

"Scala is an acronym for

Scalable Language"

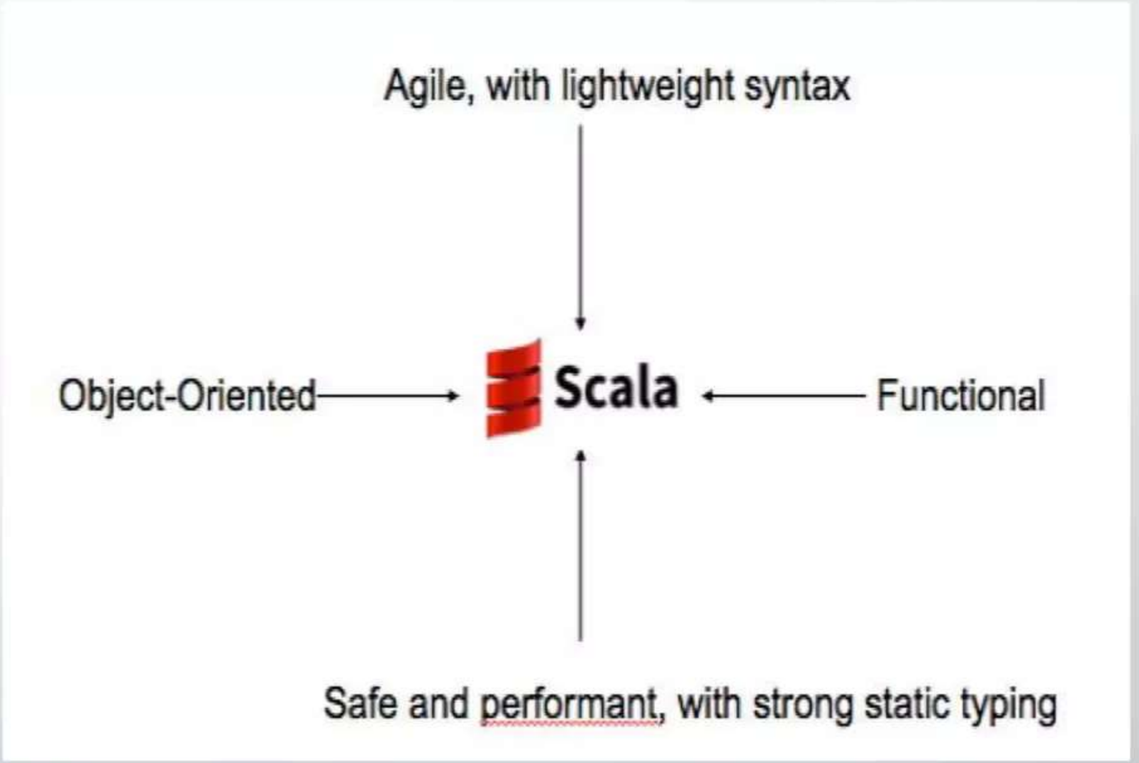
Is so named because it was designed to grow with the demands of its users.

Why a new Language?

The work on Scala was motivated by two **hypotheses**:

Hypothesis 1: A general-purpose language needs to be *scalable*; the same concepts should describe small as well as large parts.

Hypothesis 2: Scalability can be achieved by unifying and generalizing *functional* and *object-oriented* programming concepts.



If we were forced to name just one aspect of Scala that helps scalability, we'd pick its combination of object-oriented and functional programming.

What is Scala?

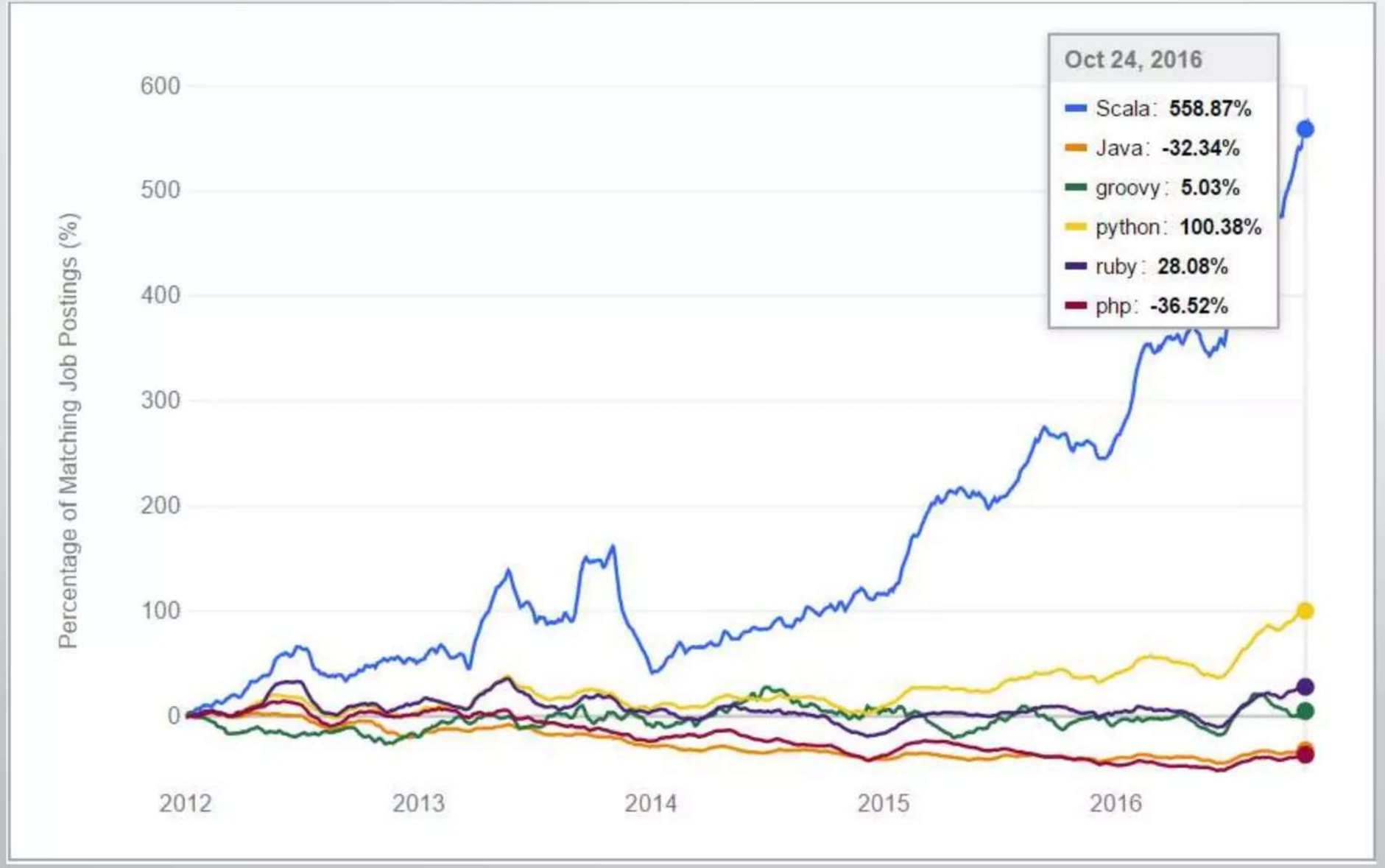
- Scala compiles to Byte Code in the JVM. Scala programs compile to JVM bytecodes. Their run-time performance is usually on par with Java programs.
- You can write a **.class** being java or scala code.
- Interoperability with Java, So you can easily use the Java Ecosystem.

Why Scala?

- Scala is concise. No boilerplate code! (semicolons, return, getter/setter, …) Fewer lines of code mean not only less typing, but also less effort at reading and understanding programs and fewer possibilities of defects
- Scala is high-level. OOP and FP let you write more complex programs.
- Scala is statically typed, verbosity is avoided through type inference so it look likes it's a dynamic language but it's not.

JVM Languages Timeline

Indeed Scala Job Trend



Even Apple needs Scala developer

Job Title	Job Function	Location	Posted
Maps – Sr. Software Engineer–Java / Big Data	Software Engineering	Santa Clara Valley	Aug. 10, 2016
Siri Data Engineer	Software Engineering	Santa Clara Valley	Dec. 8, 2016
Machine Learning Engineer, Maps Real–Time Traffic	Software Engineering	Santa Clara Valley	Sep. 20, 2016
Software Engineer – Java/Hadoop	Software Engineering	Santa Clara Valley	Jul. 6, 2016
Maps Data Engineer	Software Engineering	Santa Clara Valley	Dec. 16, 2016
Maps Sr. Software Engineer -- Big Data/Hadoop	Software Engineering	Santa Clara Valley	Nov. 8, 2016
iTunes Store Commerce Engineer – New York City	Software Engineering	New York City	Dec. 5, 2016
iAd – Sr. Data Engineer	Software Engineering	Santa Clara Valley	Jul. 2, 2016
Data Scientist, App Store Engineering R&D	Software Engineering	Santa Clara Valley	Mar. 4, 2016
App Store Engineering R&D, Data Engineer	Software Engineering	Santa Clara Valley	Apr. 27, 2016
Web Services Engineer – Cloud Services Localization	Software Engineering	Santa Clara Valley	May. 18, 2016

"If I were to pick a language to use today other than Java, it would be Scala."

James Gosling, the creator of the Java programming language

Scala Ecosystem and Use Cases

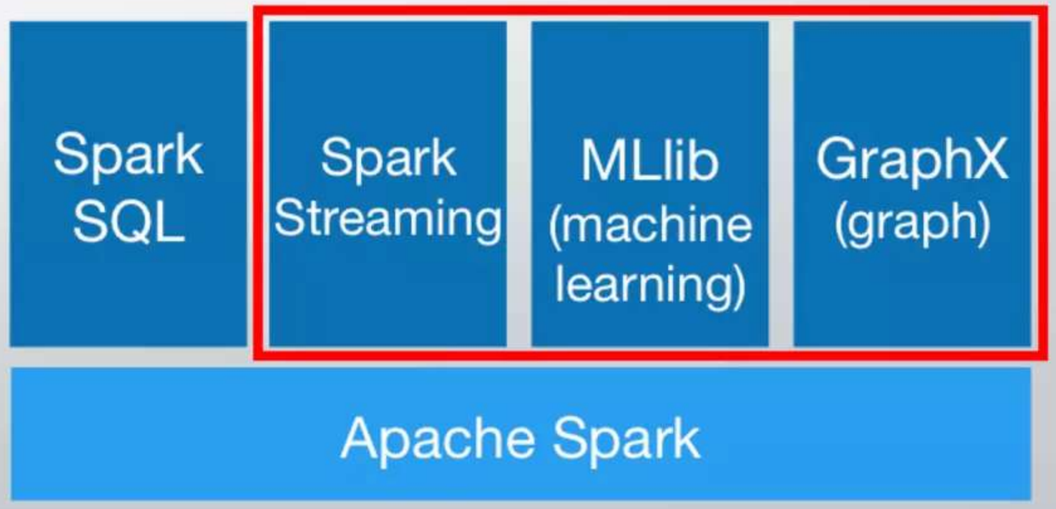
Scala Use Cases

- Big Data and Data Science
- Web Application Development, REST API Development
- Distributed System, Concurrency and Parallelism
- Scientific Computation like NLP, Numerical Computing and Data Visualization

Scala Use Cases

Big Data and Data Science

- Apache Spark is written in Scala. You can use Spark for machine learning, graph processing, streaming and generally as an in-memory distributed, fault-tolerant data processing engine.



Scala Use Cases

Big Data and Data Science

- Apache Kafka is written in Scala and Java. It provides a unified, high-throughput, low-latency platform for handling real-time data feeds.

Scala Use Cases

Big Data and Data Science

- Apache Samza is a distributed stream processing framework written in Java and Scala. It uses Apache Kafka for messaging, and Apache Hadoop YARN to provide fault tolerance, processor isolation, security, and resource management.

Scala Use Cases

Big Data and Data Science

- Apache Flink is a distributed streaming dataflow engine written in Java and Scala.

Scala Use Cases

Web Application Development, REST API Development

- Play is a stateless, asynchronous, and non-blocking framework that uses an underlying fork-join thread pool to do work stealing for network operations, and can leverage Akka for user level operations.

- Play Framework makes it easy to build web applications with Java & Scala.

Scala Use Cases

Web Application Development, REST API Development

- Lift claim that it is the most powerful, most secure web framework available today.
- It's Secure, Scalable, Modular, Designer friendly, and Developer centric.

Scala Use Cases

Web Application Development, REST API Development

- Spray is an open-source toolkit for building REST/HTTP-based integration layers on top of Scala and Akka.
- Being asynchronous, actor-based, fast, lightweight, modular and testable it's a great way to connect your Scala applications to the world.

Scala Use Cases

Web Application Development, REST API Development

- Scalatra is a simple, accessible and free web micro-framework.
- It combines the power of the JVM with the beauty and brevity of Scala, helping you quickly build high-performance web sites and APIs.

Scalatra

Scala Use Cases

Distributed System, Concurrency and Parallelism

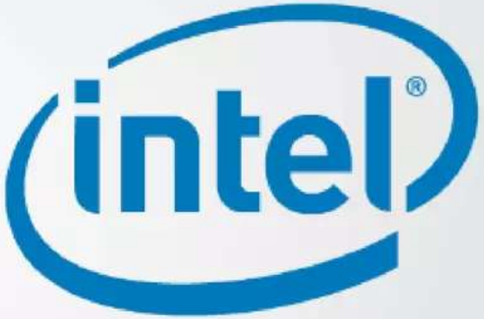
- Akka is open source middleware for building highly concurrent, distributed and fault-tolerant systems on the JVM.
- Akka is built with Scala, but offers both Scala and Java APIs to developers.
- At the heart of Akka is the concept of Actors, which provide an ideal model for thinking about highly concurrent and scalable systems.

Scala Use Cases

Scientific Computation

- Scala NLP library including Breeze, Epic, Puck.
- Wisp is a web-based plotting library.
- ADAM is a genomics processing engine and specialized file format built using Apache Avro, Apache Spark and Parquet.
- Scala-Chart is another library for creating and working with carts.
- Scalalab is an easy and efficient MATLAB-like scientific computing library in Scala

They all use Scala



https://www.lightbend.com/resources/case-studies-and-storie

"We've found that Scala has enabled us to deliver things faster with less code."

Graham Tackley from **TheGuardian.**

Features of Language

- Scala Language Design
- The Basics of Scala
- Functional Programming in Scala
- Object-Oriented Programming in Scala

Scala Language Design

- Scala Type System
- Everything is an Expression
- Lexical scope
- Scopes

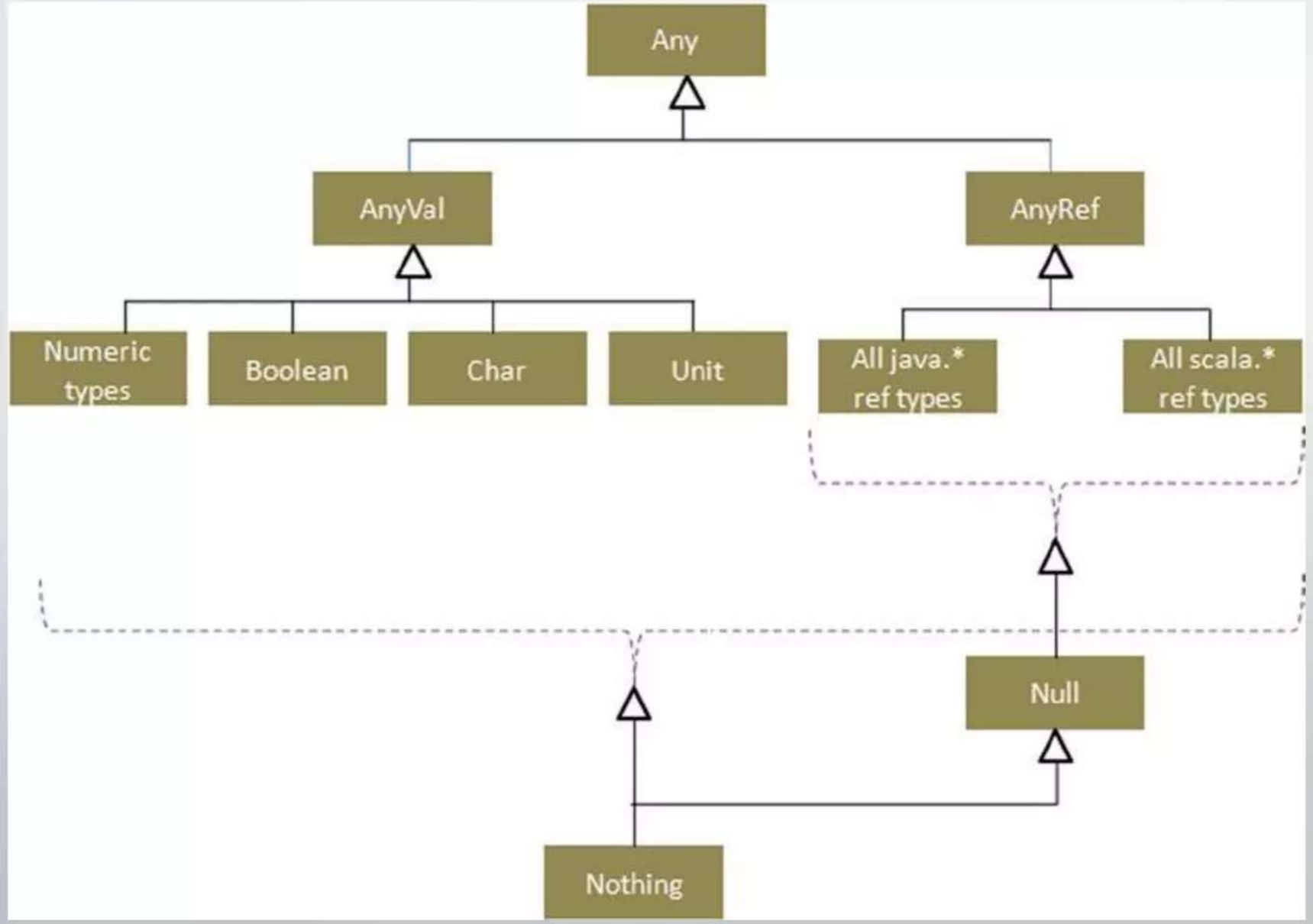
Scala Type System

Scala, unlike some of the other statically typed languages (C, Pascal, Rust, etc.), does not expect you to provide redundant type information. You don't have to specify a type in most cases, and you certainly don't have to repeat it.

Scala has a unified type system, enclosed by the type Any at the top of the hierarchy and the type Nothing at the

bottom of the hierarchy

Scala Type System



Scala Type System

- **Null:** Its a Trait.
- **null:** Its an instance of Null- Similar to Java null.
- **Nil:** Represents an empty List of anything of zero length. Its not that it refers to nothing but it refers to List which has no contents.
- **Nothing:** is a Trait. Its a subtype of everything. But not superclass of anything. There are no instances of Nothing.
- **None:** Used to represent a sensible return value. Just to avoid null pointer exception. Option has exactly 2 subclasses- Some and None. None signifies no result from the method.
- **Unit:** Type of method that doesn't return a value of anys sort.

Everything is An Expression

In programming language terminology, an "expression" is a combination of values and functions that are combined and interpreted by the compiler to create a new value, as opposed to a "statement" which is just a standalone unit of execution and doesn't return anything

```
def ifThenElseExpression(aBool:Boolean) = if aBool then 42 else 0
```

Lexical Scope (Static)

- **Scala using Lexical scoping like many other languages (why?)**
  - **Much easier for the user (programmer)**
  - **Much easier for the compiler to optimize**

Scopes

In a Scala program, an inner variable is said to shadow a like-named outer variable, because the outer variable becomes invisible in the inner scope.

$$Local\ Definitions \rightarrow Explicits\ imports \rightarrow Wild\ card\ imports \rightarrow Packages$$

The Basics of Language

- Data Types
- Casting
- Opperations
- If Expression
- Arrays & Tuples & Lists
- For & while Comprehension
- Functions
- Call By …
- Pattern Matching

Data Types

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

```
scala> var a:Int=2
a: Int = 2

scala> var a:Double=2.2
a: Double = 2.2

scala> var a:Float=2.2f
a: Float = 2.2

scala> var a:Byte=4
a: Byte = 4

scala> var a:Boolean=false
a: Boolean = false
```

Casting

- asInstanceOf

```
scala> var a:Char=(Char)955//Compile error
 <console>:1: error: ';' expected but integer literal found.
 var a:Char=(Char)955//Compile error
                  ^

scala> var a:Int=955
a: Int = 955

scala> var b=a.asInstanceOf[Char]
b: Char = λ
```

Operations

```
scala> var a=955
a: Int = 955

scala> import scala.math._
import scala.math._

scala> print(4-3/5.0)
//compile time they will change to primitive not object any more to run faster 3.4

scala> print(a+3)//a.add(3) add=>+
958

scala> println(math.sin(4*Pi/3).abs)
//1)you can import classes 2)=>math.abs(math.sin(4*Pi/3))
0.8660254037844385

scala> if(a==955) | print("lambda")
Lambda

scala> println("HOLY".>=("EVIL"))
true
```

If Expression

```
scala> var m="wood"
m: String = wood

scala> println(m.length())
4

scala> println(s"$m is brown")
wood is brown

scala> println(f"$m%5s")
 wood

scala> if(m=="wood"){//== is equivalent to equal
 | print("brown");
 | }else if(m.equals("grass")){
 | println("green");
 | }else{
 | print("i don't know")
 | }
 brown
```

Arrays

```
scala> var arr=Array(5,4,47,7,8,7)
 arr: Array[Int] = Array(5, 4, 47, 7, 8, 7)

scala> println(arr(1));println(arr(2));println(arr(3));
 4 47 7

scala> var array=new Array[String](3);
 array: Array[String] = Array(null, null, null)

scala> var at=Array(4,4.7,"Gasai Yuno")
 at: Array[Any] = Array(4, 4.7, Gasai Yuno)

scala> var matrix=Array.ofDim[Int](2,2)
 matrix: Array[Array[Int]] = Array(Array(0, 0), Array(0, 0))
```

Tuples

```
scala> var tuples=(2*5,"ten",10.0d,10.0f,10.0,(9,"NINE"))
 tuples: (Int, String, Double, Float, Double, (Int, String)) =
(10,ten,10.0,10.0,10.0,(9,NINE))

scala> print(tuples._1)
 10

scala> print(tuples._2)
 ten

scala> print(tuples._6._2)
 NINE
```

Lists

```
scala> var lst=List("b","c","d")
 lst: List[String] = List(b, c, d)

scala> print(lst.head)
 b

scala> print(lst.tail)
 List(c, d)

scala> var lst2="a"::lst
 lst2: List[String] = List(a, b, c, d)

scala> var l=1::2::3::4::Nil
 l: List[Int] = List(1, 2, 3, 4)

scala> var l2=l::List(5,6)
 l2: List[Any] = List(List(1, 2, 3, 4), 5, 6)

scala> var l3=l:::List(5,6)
 l3: List[Int] = List(1, 2, 3, 4, 5, 6)
```

While

- while (Boolean Expression) { Expression }
- do { Expression } while (Boolean Expression)

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

scala> while(i<10){
 | i+=1;
 | print(i+" ")
 | }
 1 2 3 4 5 6 7 8 9 10
```

Note : In functional , It is preferred to not use while and in general imperative style.

For Comprehension

- for (Iterations and Conditions)

```
val books = List("Beginning Scala", "Beginning Groovy",
"Beginning Java", "Scala in easy steps", "Scala in 24
hours")

 for (book<-books)
println(book)

var scalabooks = for{ book <-books if
book.contains("Scala") }yield book

for(book<-books if book.contains("Scala") )
println(book)
```

foreach

- ✓ foreach (Conditions)

```
scala> val list = List("Human","Dog","Cat")
 list: List[String] = List(Human, Dog, Cat)

scala> list.foreach((s : String) => println(s))
 Human
 Dog
 Cat

scala> list.foreach(s => println(s))
 Human
 Dog
 Cat
scala> list.foreach(println)
 Human
 Dog
 Cat
```

Foreach vs For Comprehension

- ✓ Foreach VS for each

```
scala> for (s<- list) println(s)//it is actually
calling foreach
 Human
 Dog
 Cat

scala> for(s<-list if s.length==3) println(s)
 Dog
 Cat

scala> list.foreach((s : String) => if(s.length()==3)
println(s))
 Dog
 Cat
```

Functions

```
scala> import scala.math._
 import scala.math._

scala> def add(x:Int,y:Int):Int = {
 | x+y//the last value if you dont specify return
 |
 | }
 add: (x: Int, y: Int)Int

scala> def add(x:Double,y:Double) = x+y
 add: (x: Double, y: Double)Double
```

Note: if you add "return" you need to specify return type else you are not obligated and one line function no need to bracket.

Functions

```
scala> def printAdd(x:Int,y:Int){// no equal mark no return
 | println(x+y)
 | }
 printAdd: (x: Int, y: Int)Unit

scala> def printAdd(x:Double,y:Double):Unit ={// Unit = void
 | println(x+y)
 | }
 printAdd: (x: Double, y: Double)Unit
```

Functions

```
scala> def distance1(x1:Double,y1:Double,x2:Double,y2:Double):Double={
 | def dif(x1:Double,x2:Double)={
 | pow(x2-x1,2)
 | }
 | return sqrt(dif(x1,x2)+dif(y1,y2))
 | }
 distance1: (x1: Double, y1: Double, x2: Double, y2: Double)Double

scala> def
distance2(x1:Double,y1:Double,x2:Double,y2:Double,dif:(Double,Double)=>Double):Double={
 | return sqrt(dif(x1,x2)+dif(y1,y2))
 | }
 distance2: (x1: Double, y1: Double, x2: Double, y2: Double, dif: (Double, Double) =>
Double)Double
```

Functions

```
scala> println(add(4.7,3.2))
 7.9

scala> println(distance1(0, 0, 3, 0))
 3.0

scala> var dif:(Double,Double)=>Double=(x:Double,y:Double)=>{pow(x-y,2)}
 dif: (Double, Double) => Double = <function2>

scala> println(distance2(0, 0, 3, 0, dif))//unanonymous
 3.0

scala> println(distance2(0, 0, 3, 0, (x:Double,y:Double)=>pow(x-y,2)))//anonymous
 3.0
```

Tail Recursion

```
import scala.annotation.tailrec

    // 1 - basic recursive factorial method
    def factorial(n: Int): Int = {
     if (n == 0) 1 else n * factorial(n-1)
    }

     // 2 - tail-recursive factorial method
    def factorial2(n: Long): Long = {
     @tailrec
     def factorialAccumulator(acc: Long, n: Long): Long = {
     if (n == 0) acc else factorialAccumulator(n*acc, n-1)
     }
     factorialAccumulator(1, n)
    }
```

Call By Name

```
def delayed(t:=> Long) = {
 println("Indelayed method")
 println("Param:"+t)
 t
}
```

Call By Ref

```
def notDelayed(t:Long) = {
 println("Innotdelayed method")
 println("Param:"+t)
 t
}
```

Note : In Java, all method invocations are call-by-reference or call-by-value (for primitive types) .

Scala gives you an additional mechanism for passing parameters to methods (and functions):

call-by-name, which passes a code block to the callee. Each time the callee accesses the parameter, the code block is executed and the value is calculated. Call-by-name allows you to pass parameters that might take a longtime to calculate but may not be used.

Pattern Matching

```
scala> def matchChecking(a:Any):Unit={
 | a match{
 | case a:Int => {
 | println("One");
 | "ali"*5;
 | }
 | case "two" | "One" => {
 | println("Two");
 | 2*8;
 | }
 | case a if a<4.7 =>{
 | println("four point seven");
 | 3.8*2;
 | }
 | case _ => {
 | println("Recognizing ...");
 | a;
 | }}}
 matchChecking: (a: Any)Unit

scala> matchChecking(1);
 One
```

Pattern Matching

```
scala> def simple_fun(list: List[Char]): List[Nothing] =
list match {
 | case x :: xs => {
 | println(x)
 | simple_fun(xs)
 | }
 | case _ => Nil
 | } simple_fun: (list: List[Char])List[Nothing]

scala> simple_fun(simplelist)
 a
 b
 c
 d
 res22: List[Nothing] = List()
```

Functional Programming in Scala

- **Lazy val & Eager Evaluation**
- **Currying**
- **Concurrency**
- **Closures**

Lazy val & Eager Evaluation

- ✓ **lazy val vs. val**

  The difference between them is, that a val is executed when it is defined whereas a lazy val is executed when it is accessed the first time.

  In contrast to a method (defined with def) a lazy val is executed once and then never again. This can be useful when an operation takes long time to complete and when it is not sure if it is later used. languages (like Scala) are strict by default, but lazy if explicitly specified for given variables or parameters.

Currying

Methods may define multiple parameter lists. When a method is called with a fewer number of parameter lists, then this will yield a function taking the missing parameter lists as its arguments.

```
scala> def nDividesM(m : Int)(n : Int) = (n % m == 0)
 nDividesM: (m: Int)(n: Int)Boolean

scala> val isEven = nDividesM(2)_
 isEven: Int => Boolean = <function1>

scala> println(isEven(4))
 true
```

Note : when you give only a subset of the parameters to the function, the result of the expression is a partially applied function.

Closures

A closure is a function, whose return value depends on the value of one or more variables declared outside this function.

```
scala> var votingAge = 18
 votingAge: Int = 18

scala> val isVotingAge = (age: Int) => age >= votingAge
 isVotingAge: Int => Boolean = <function1>

scala> isVotingAge(16)
 res2: Boolean = false
```

Concurrency

- **You have three options in concurrent programming:**
  - **Akka Actor Model**
  - **Thread**
  - **Apache Spark**

Actor

- Concurrency using threads is hard
  - Shared state – locks, race conditions, deadlocks
- Solution – message passing + no shared state
  - Inspired by Erlang language
    - Erlang used at Ericsson since 1987, open source since 1998
    - Facebook chat backend runs on Erlang

What is an Actor?

- **Actor is an object that receives messages**
- **Actor has a *mailbox* – queue of incoming messages**
- **Message send is by default *asynchronous***
  - **Sending a message to an actor immediately returns**

Object-Oriented Programming in Scala

- **Classes**
- **Objects**
- **Traits**
- **Case Classes**
- **Exceptions**

Classes

```
/** A Person class.
 * Constructor parameters become
 * public members of the class.*/
class Person(val name: String, var age: Int) {}

var p = new Person(“Peter", 21);
p.age += 1;
```

Objects

- Scala's way for "statics"
  - not quite – see next slide
  - (in Scala, there is no *static* keyword)
- "Companion object" for a class
  - = object with same name as the class

Objects

```
// we declare singleton object "Person"
// this is a companion object of class Person
object Person {
  def defaultName() = "nobody"
}
class Person(val name: String, var age: Int) {
  def getName() : String = name
}

// surprise, Person is really an object
val singleton = Person;
var p:Person=new Person(" ",0)
```

Case Classes

Implicitely override toString, equals, hashCode
- **take object's structure into account**
- **Usefull in pattern matching**

```
abstract class Expr
case class Number(n: Int) extends Expr
case class Sum(e1: Expr, e2: Expr) extends Expr

// true thanks to overriden equals
Sum(Number(1), Number(2)) ==
Sum(Number(1), Number(2))
```

Exceptions

```
object Main {
  def main(args: Array[String]) {
    try {
      val elems = args.map(Integer.parseInt)
      println("Sum is: " + elems.foldRight(0) (_ + _)) }
    catch {
      case e: NumberFormatException =>
        println("Usage: scala Main <n1> <n2> ... ")
    }
  }
}
```

Traits

Like Java interfaces

But can contain implementations and fields

```
trait Pet {
  var age: Int = 0
  def greet(): String = {
    return "Hi"
  }
}
```

Extending Traits

```
class Dog extends Pet {
  override def greet() = "Woof"
}

trait ExclamatoryGreeter extends Pet {
  override def greet() = super.greet() + " !"
}
```

Mixins traits

Traits can be "mixed in" at instation time

```
trait ExclamatoryGreeter extends Pet {
  override def greet() = super.greet() +
" !"
}
```

val pet = new Dog with ExclamatoryGreeter
println(pet.greet()) // Woof !

Scala has Multiple inheritance!!!

Solving the Diamond Problem

Scala's solution to the Diamond Problem is actually fairly simple: it considers the order in which traits are inherited. If there are multiple implementors of a given member, the implementation in the supertype that is furthest to the right (in the list of supertypes) "wins." Of course, the body of the class or trait doing the inheriting is further to the right than the entire list of supertypes, so it "wins" all conflicts, should it provide an overriding implementation for a member

Programming Environment

1. **Download & Install JDK**
2. **Download & Install Scala** http://www.scala-lang.org/download/.
3. **Add the installed software to your environment**
4. **Type scala in Console**
5. **Have fun ☺**

Note: You can download plugins in eclipse and intelij and code

Resources

- http://www.scala-lang.org/files/archive/spec/2.12/
- https://www.tutorialspoint.com/scala/
- http://stackoverflow.com/tags/scala/info
- Beginning Scala

Q & A

THANKS FOR YOUR ATTENTION