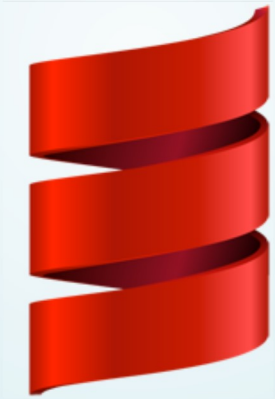
SCALA 101

OBJECT ORIENTED AND **FUN**CTIONAL



SO?

1. What is scala?
2. Who made/maintains/uses scala?
3. Where was it developed?
4. Why
    - was it made?
    - you should use it?
5. How do I get started?

BUT FIRST!

WHO AM I?

- Darren Gibson
- OSU Grad
- Software Engineer @

- Professional .Net developer.
- Member of Tulsa Java and Tulsa Web Devs
- **Scala Enthusiast**

WHERE CAN I BE FOUND?

- github.com/zarthross
- twitter.com/zarthross
- techlahoma.slack.com/team/zarthross

WHAT IS SCALA?


```
class Point(xc: Int, yc: Int) {
    var x: Int = xc
    var y: Int = yc
    def move(dx: Int, dy: Int) {
        x = x + dx
        y = y + dy
    }
    override def toString(): String = "(" + x + ", " + y + ")";
}
```

PARADIGM

IMPERATIVE

```
def fib( n : Int ) : Int = {
  var a = 0
  var b = 1
  var i = 0

  while( i < n ) {
    val c = a + b
    a = b
    b = c
    i = i + 1
  }
  return a
}
```

PARADIGM

FUNCTIONAL

```
def fib2( n : Int) : Int = {
    @scala.annotation.tailrec // Optional
  def fib_tail( n: Int, a:Int, b:Int): Int = n match {
    case 0 => a
    case _ => fib_tail( n-1, b, a+b )
  }
  return fib_tail( n, 0, 1)
}
```

`fib_tail` becomes a `while` loop in the bytecode!

TYPING DISCIPLINE

```
val foo: Int = 10 // Statically Checked!
val bar: String = 10 /* This FAILS */ // Strongly Typed
val bleh = 10 // Types can be infered!

// Structural Typing!
// (probably shouldn't do this...)
// (Uses JVM reflection so ends up being slow)
def quacker(duck: { def quack: String }) =  println(duck.quack)

quacker(new { def quack = "QUACK!" })

class Goose { def quack = "HONK!" }
quacker(new Goose())
```

PLATFORMS

- Java Virtual Machine (Scala)
  - Windows/Linux/Mac
    - x86(-64), Arm and more
  - Android
  - iOS via BugVM
  - .Net via IKVM
- Javascript (Scala.js)
  - NodeJS
  - Any JS compatible browser
- Bare Metal (Scala-native)
  - Any LLVM Target

WHEN

FIRST APPEARED

JANUARY 20, 2004

LAST STABLE RELEASE

2.11.8 MARCH 8, 2016

WHO?

WHO USES SCALA?

And MANY More

http://www.lightbend.com/resources/case-studies-and-stories

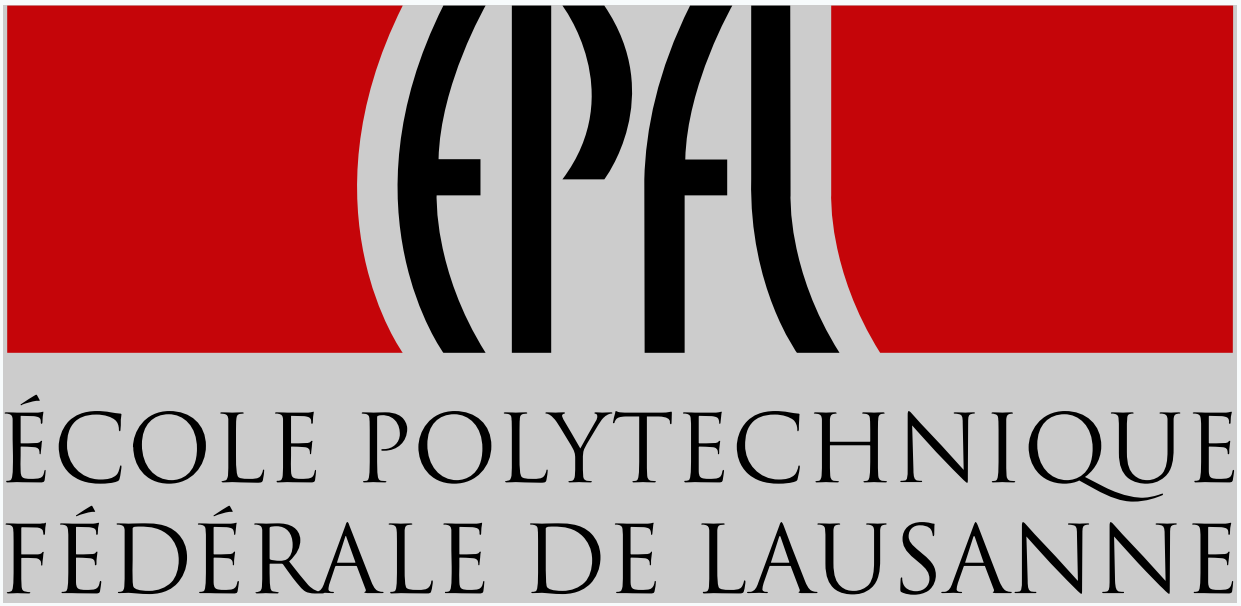
WHO MADE SCALA

MARTIN ODERSKY

Created an early javac compiler and java generics and much more.

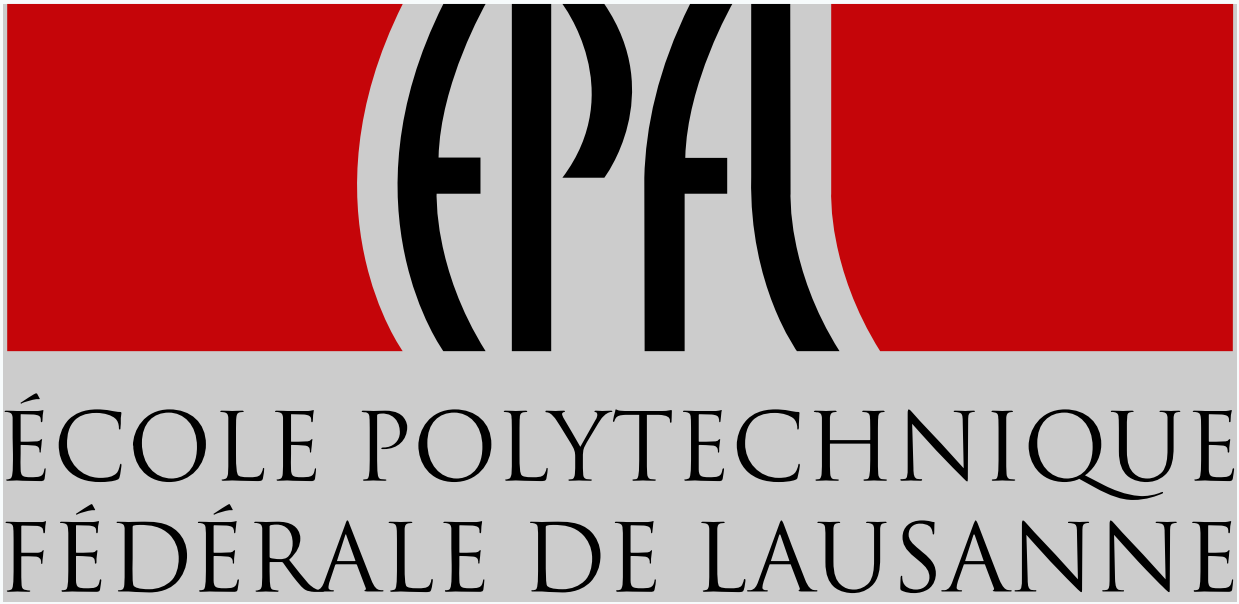
WHERE WAS SCALA DEVELOPED?

Programming Methods Laboratory of



WHO MAINTAINS SCALA?

AND



WHY?

QUESTION?

CAN YOU MATCH THE SIZE OF THE GRAMMAR TO THE LANGUAGE?

1. C# - 1210
2. C++ - 1748
3. Haskell 98 - 416
4. Java 7 - 528
5. Scala 2.10 - 302

source: https://groups.google.com/forum/#!topic/scala-debate/fE8w9pi-hbQ post by Martin Odersky

SO, WHY WAS IT MADE?

- To make a **SCA**LABLE **LA**NGUAGE
- To Combine Object Oriented and Functional Paradigms
- Make Concurrent/Parallal Programming Easier
- Java SUUUUCKs
- JVM is OK
- Lots of good libraries...
- but, really.... Java SUUUUCKs

WHY SHOULD YOU USE IT?

All the previous reasons and more!

- The Syntax
- The Libraries
- The not JAVA

HOW DO I GET STARTED?

- Tools/IDEs
- Syntax
  - Variables
  - Expressions
  - Functions/Lambdas
  - Class/Object/Package
  - Interface/Traits
  - Types (Generics)
  - Pattern Matching
- If we have time
  - Option Monad
  - Implicits

TOOLS/IDES

- IDE's
  - Eclipse
    - Idea Intellij <- Recommended
    - Netbeans
- Build scripts
  - Maven
  - Apache Buildr
  - Gradle
  - SBT (Simple/Scala Build Tool) <- Recommended
- Scala-SDK
  - scala/scalac
- scala-js-fiddle.com

HELLO WORLD!

HELLO WORLD!

JAVA

```
class HelloWorld {
     public static void main(String[] args) {
          System.out.println("Hello, world!");
     }
}
```

HELLO WORLD!

SCALA

```
object HelloWorld {
    def main(args: Array[String]): Unit = {
      println("Hello, world!")
    }
}
```

JUST KIDDING.

WE CAN DO BETTER

HELLO WORLD!

SCALA

```
object HelloWorld extends App {
    println("Hello, world!")
}
```

VARIABLES

VARIABLES

- Semicolons are optional
- Most variables types can be infered
- `val` vs `var`

QUICK ASIDE

Java and Scala share primitive types

- `Int`
- `Float`
- `Double`
- `String`
- (etc)

VARIABLES (EXAMPLES)

JAVA

```java
int foo = 5;
String bar = "asdf";

final double bob = 20.1;
final String jo = "JO";
```

SCALA

```scala
var foo: Int = 5;
var bar: String = "asdf"

val bob = 20.1   // Type Infered
val jo: String = "JO"
```

VARIABLES (CONT)

```
val t: (Int,Int,String) = (1,3,"TEST")
// Really its Tuple3[Int,Int,String]

//Tuple destructuring bind??? (similar to pythons)
val (x,y,z) = t

x == t._1
y == t._2 //etc

// Useful for writing maps since its just a Tuple2
val bar: (Int,String) = 1 -> "s"
```

EXPRESSIONS

EXPRESSIONS

IF

```
var bar = 10

val foo = if(bar < 10) "less" else "more"
//foo: String = more

println(foo)
```

EXPRESSIONS

FOR

```
var bar = 10

val foo = for (i <- Range(0,10)) yield i * bar
// foo: IndexedSeq[Int] = Vector(0, 10, 20, 30, 40, 50, 60, 70, 80, 90)
```

EXPRESSIONS

MATCH

A better `switch`

```
val x:Any = 9
val y = x match {
    case 1 => "one"
    case 2 => "two"
    case x: Int if x % 2 == 0 => "even"
    case "foobar" => "wat?"
    case _ => "other"
}
//y: String = other
```

EXPRESSIONS

BLOCK

```
val fooBar: Int = {
    val f = 100 * 500
    f * 10000
}
// fooBar: Int = 500000000
```

EXPRESSIONS

TRY

```
val userInput = "19"
val n: Int = try {
  5 / userInput.toInt
} catch {
    case a: ArithmeticException => 0
    case n: NumberFormatException => throw n
    case _: Throwable =>
            println("PROBLEM! Doing nothing");
            -1
} finally {
    println("Finally!")
}
```

FUNCTIONS

FUNCTIONS

INFERED RETURN TYPES

```
def doubler(i: Int): Int = {
    return i * 2
}

def triplier(i: Int) = {
    i * 3 // 'return' is optional and type is infered
}
```

FUNCTIONS

CURLY BRACES ARE OPTIONAL (FOR ONE LINERS)

```
def triplier(i: Int) = i * 3

val foo: Int = triplier(10)
```

FUNCTIONS

NESTED

```
def foo(f1: Double, f2: Double): String = {
    def bar(a: Double) = s"BAR! $a / $f1 = ${ a / f1}"

    bar(f2)
}

val r = foo(5, 11)
// r: String = BAR! 11.0 / 5.0 = 2.2
```

FUNCTIONS

CURRYING

```
def curry(i: Int)(j: Int) = i + j

var c = curry(1) _

c(2)
```

FUNCTIONS

GENERICS

```
def logIt[T](t: T): T = {
    println(s"${System.currentTimeMillis()}: $t")
    t
}

logIt("asdf")
// 1470315417220: asdf
// res2: String = asdf

logIt(102.123)
// 1470315422909: 102.123
// res3: Double = 102.123
```

FUNCTIONS

CALL BY NAME

```
def timer[T](block: => T): (T, Long) = {
    val startTime = System.nanoTime
    val result = block
    val endTime = System.nanoTime
    (result, endTime - startTime)
}

timer(Math.pow(100, 100))
//res10: (Double, Long) = (1.0E200,12513)

timer {
    println("Sleeping")
    Thread.sleep(10000)
    println("Sleeped")
}
//Sleeping
//Sleeped
```

FUNCTIONS

LAMBDA SYNTAX

```
def t1(a: Int) = a * 10

val t2: Int => Int = (a) => a * 10

val t3 = (a: Int) => a * 10
```

FUNCTIONS

LAMBDA PT2

```
var l = List(1,2,3,4)

l.reduce((x,y) => x+y)

l.map(_ + 2)
//> res1: List[Int] = List(3, 4, 5, 6)

l reduce { (x,y) => x + y }
//> res2: Int = 10
```

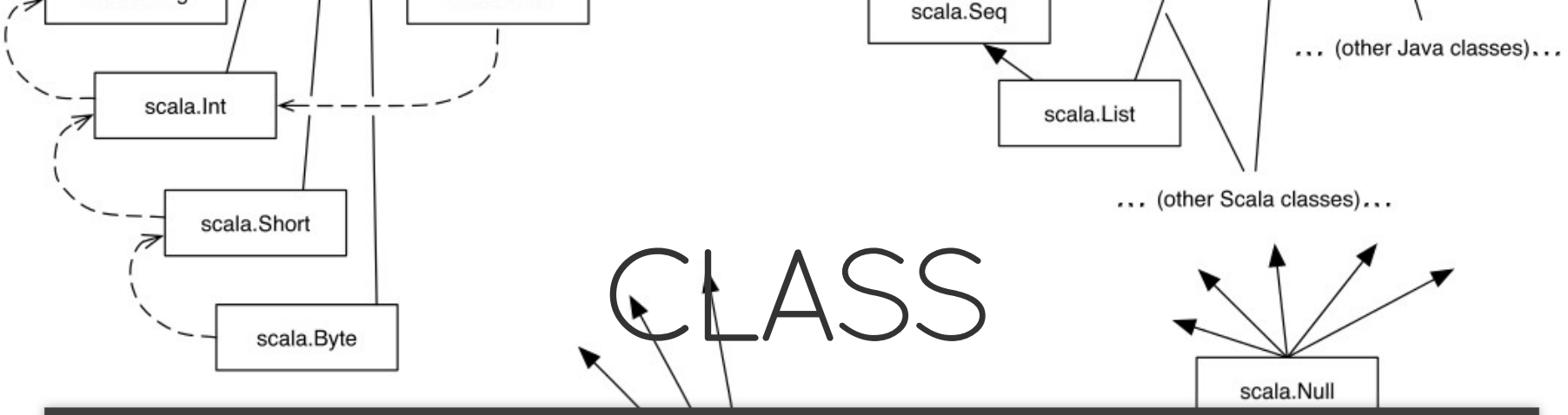
FUNCTIONS

VAR-ARGS

```
def sum(vals: Int*): Int = vals.reduce(_ + _)

sum(1,2,3,4)
//res4: Int = 10
```

OBJECTS



CLASS

```
// Primary constructor is function body
class Car(val model: String, private var spd: Double) {
  def this(model: String) = this(model,0) // Secondary constructor

  def isMoving = speed != 0  // Accessor

    println("Is this car moving?", isMoving)  // Yup, constructor

    private var desc = s"Car: $model"
    override def toString(): String = s"$desc $speed mph"

    def speed = spd           // Accessor
    def speed_=(v: Double) {  // Mutator
      spd = v;
    }
}
```

CLASS INHERITANCE

```
class Vehicle(val model: String)

class Truck(val model: String, towCapcity: Int)
    extends Vehicle(model)
// Child class must call parent primary constructor
// No multiple inheritance with classes
```

'STATIC'

Just Kidding, no such thing

Only singletons!

SINGLETONS

```
class Point(val x: Int, val y: Int)

object Point { // 'Companion` Object
    private var foo: Int = 10 // meh, its a static var
    def add(a: Point, b: Point): Point = //Static(ish) method!
        new Point(a.x + b.x, a.y + b.y)

    def apply(x: Int, y: Int) = new Point(x,y) //Factory
    def unapply(t: Point): Option[(Int, Int)] = Some((t.x, t.y))
}

val i = Point(10,30) // Point.apply
Point.add(i, i)
i match {
    case Point(x,y) => println(s"($x, $y)") // Point.unapply
}
```

CASE CLASS

```
case class Point(x: Int, y: Int) {// x, y immutable by default
    def add(a: Point) = Point(x + a.x, y + a.y)
}
val i1 = Point(10,30)
val i2 = i1.copy(y = i1.y + 10)

i1 add i2

i1 match {
    case Point(x,y) => println(s"($x, $y)")
}

i1 == Point(10,30) // true
i1 == Point(1, 3)  // False
```

PACKAGE

```scala
// com/garden/apples/package.scala
package com.garden

package object apples extends RedApples with GreenApples {
    val redApples = List(red1, red2)
    val greenApples = List(green1, green2)
}

// com/garden/apples/RedApples.scala
package com.garden.apples
trait RedApples { val red1, red2 = "red" }

// com/garden/apples/GreenApples.scala
package com.garden.apples
trait GreenApples { val green1, green2 = "green" }
```

TYPES

- Metaprogramming
- Same features as java + a few more.
- Still experiences erasure.
  - Workaround using implicit ‘Manifest’

TYPES

- Variance
  - `T'` is subclass of `T`
  - covariant `[+T]`: `C[T']` is a subclass of `C[T]`
  - contravariant `[-T]`: `C[T]` is a subclass of `C[T']`
  - invariant `[T]`: `C[T]` and `C[T']` are not related

TYPES

- Type Bounds
  - based on subtype relationships
  - for instance: `[T <: AnyVal]`
- View bounds
  - types that are 'viewable' as another type.
- Much More (structural types, etc)

TRAITS

TRAITS

(AKA MIXINS)

```
trait Automobile {           // Like Interfaces, BUT
    def Type: String = "Auto" // Can have default implementations
    val hasWheels = true      // Allows var/vals
}
trait Sporty { this: Automobile => // Sporty must be an Automobile
    def MaxSpeed: Long
}
class Car extends Automobile with Sporty {
    override def Type = "Car"
    def MaxSpeed: Long = 100
}

class Foo extends Sporty {} // Does not compile
// <console>:9: error: illegal inheritance;
// self-type Foo does not conform to Sporty's selftype
//  Sporty with Automobile
```

TRAIT

LINEARIZATION

```
trait A { def common = "A" }

trait B extends A { override def common = "B" }
trait C extends A { override def common = "C" }

class D1 extends B with C
class D2 extends C with B

(new D1).common == "C"
(new D2).common == "B"
```

source: https://ktoso.github.io/scala-types-of-types/

TRAIT

LINEARIZATION EXPLAINED

```
class D1 extends B with C

// start with D1:
B with C with <D1>

// expand all the types until you reach Any for all of them:
(Any with AnyRef with A with B) with (Any with AnyRef with A with C) with

// remove duplicates by removing "already seen" types,
// when moving left-to-right:
(Any with AnyRef with A with B) with (                              C) with

// write the resulting type nicely:
Any with AnyRef with A with B with C with <D1>
```

source: https://ktoso.github.io/scala-types-of-types/

ADT

ABSTRACT DATA TYPES

BASIC PATTERN MATCHING

```
def matchTest(x: Int): String = x match {
  case 1 => "one"
  case 2 => "two"
  case _ => "many"
}
println(matchTest(3))
```

ADTS

PUTTING IT ALL TOGETHER

```
abstract class Term
case class Val(v: Int) extends Term
case class Plus(left: Term, right: Term) extends Term
case class Times(left: Term, right: Term) extends Term
case class Pow(x: Term, n: Term) extends Term
```

ADTS

PRINT

```
def printTerm(term: Term): String = {
  term match {
    case Val(n) => n.toString
    case Plus(l,r) => s"(${printTerm(l)} + ${printTerm(r)})"
    case Times(l,r) => s"(${printTerm(l)} * ${printTerm(r)})"
    case Pow(x,n) => s"(${printTerm(x)} ^ ${printTerm(n)})"
  }
}

val p = Plus(Val(1), Val(5))
val t = Times(Val(3), p)
val powAdt = Pow(t, Val(3))

val result = printTerm(powAdt)
println(result)
// ((3 * (1 + 5)) ^ 3)
```

ADTS

EXECUTE

```
def executeTerm(term: Term): Int = {
  term match {
    case Val(n) => n
    case Plus(l,r) => executeTerm(l) + executeTerm(r)
    case Times(l,r) => executeTerm(l) * executeTerm(r)
    case Pow(_ ,Val(n)) if n == 0 => 1 // Base case for Pow recursion
    case Pow(x@Val(_), n@Val(nv)) =>  // If x and n are simple
      executeTerm(Times(x, Pow(x, Plus(n, Val(-1)))))
    case Pow(x, n) => // Simplify x and n and try again
      executeTerm(Pow(Val(executeTerm(x)),Val(executeTerm(n))))
  }
}

val r = executeTerm(powAdt)
println(r)
// 5832
```

RESOURCES

- coursera.org/specializations/scala
- scala-js-fiddle.com
- docs.scala-lang.org
- scala-exercises.org
- twitter.github.io/scala_school
- twitter.github.io/effectivescala
- Programming in Scala Book:
  - artima.com/pins1ed
- scalatutorials.com/tour
- scala-tour.com
- ktoso.github.io/scala-types-of-types