Scala for Java Programmers

```
val presentation = Map( “topic”     -> “Scala”,
                        “venue”     -> “OKCJUG”,
                        “date”      -> “2009-03-10”,
                        “presenter” -> “Mitch Blevins” )
```

What is Scala?

Scala is a general purpose programming language designed to express common programming patterns in a concise, elegant, and type-safe way. It smoothly integrates features of object-oriented and functional languages. It is also fully interoperable with Java.

http://www.scala-lang.org/

- Martin Odersky

What is Scala?

Scala is a general purpose programming language designed to express common programming patterns in a concise, elegant, and type-safe way. It smoothly integrates features of object-oriented and functional languages. It is also fully interoperable with Java.

http://www.scala-lang.org/

- Martin Odersky

I designed it!

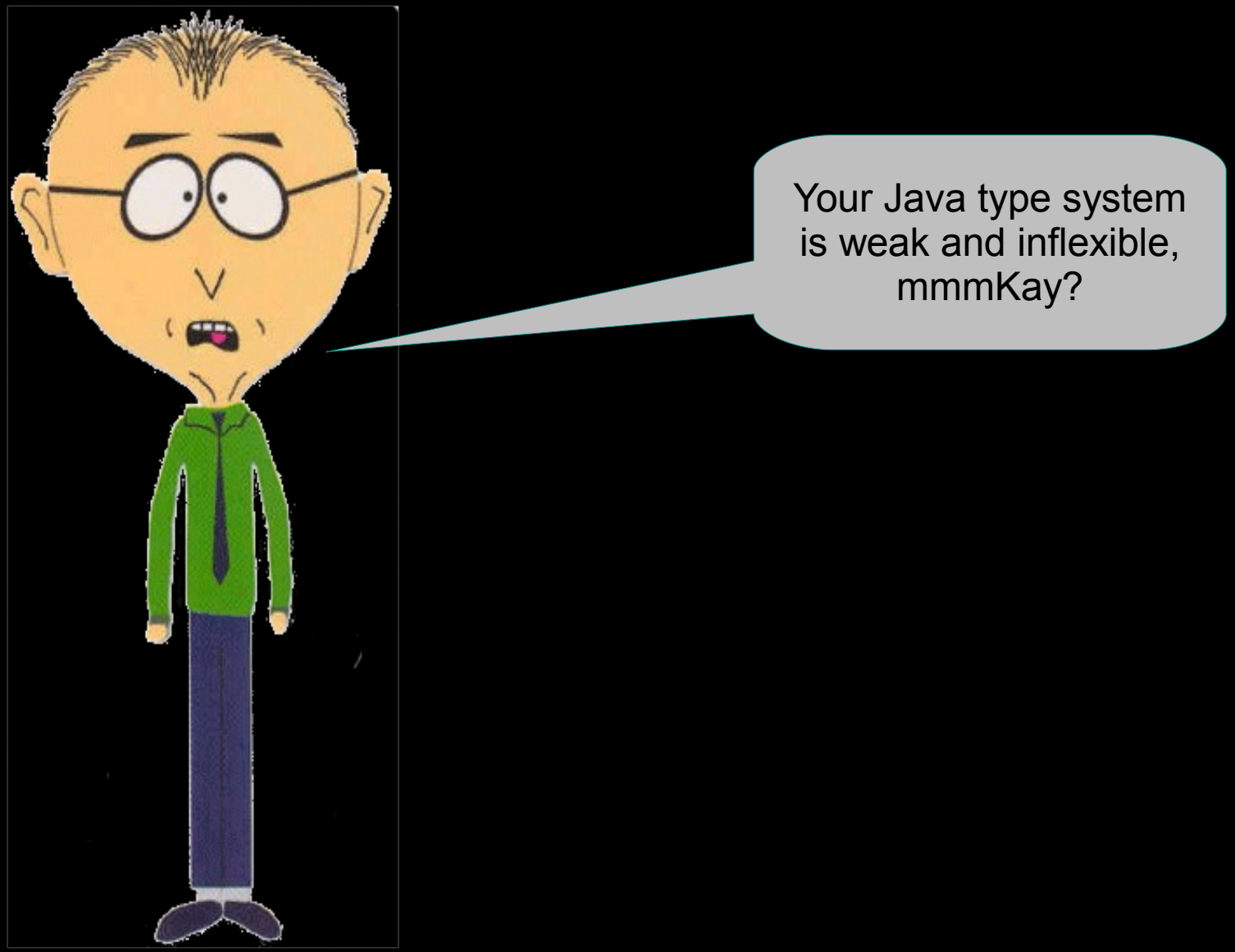
What else is Scala?

=> A better Java than Java
=> A great way to ease into learning functional programming
=> Static type systems gone wild!
=> Generics done right
=> An expressive language for DSLs
=> Because it runs on the JVM and interoperates with Java,
    possible to sneak into your projects at work
=> Concise syntax
=> Strong XML support for a web-based world
=> Closures!
=> Structural typing (a.k.a - Duck Typing)
=> Lazy evaluation

Java Interop

=> Scala compiles to Java bytecode
=> Scala can instantiate Java classes and call Java methods
=> Scala can extend Java classes
=> Scala can implement Java interfaces
=> Scala can use Java annotations
=> Scala can use Java libraries
=> Java can instantiate Scala classes and call Scala methods
=> Java can extend Scala classes
=> Java can implement Scala interfaces/traits
=> Java can use Scala libraries (with some caveats)
=> You can freely mix Java and Scala in the same project
(but not the same file)

More Motivation



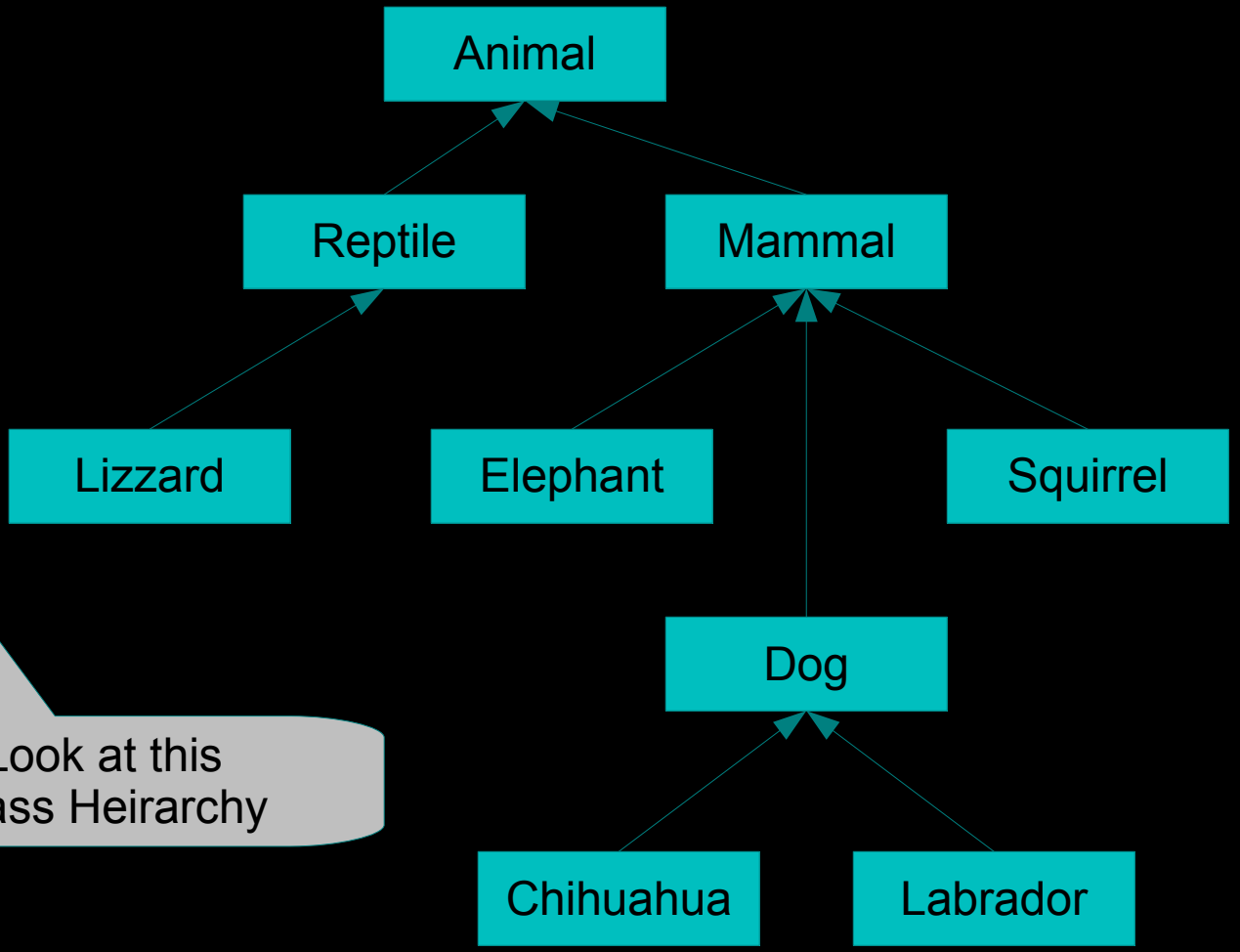
More Motivation

Your Java type system is weak and inflexible, mmmKay?

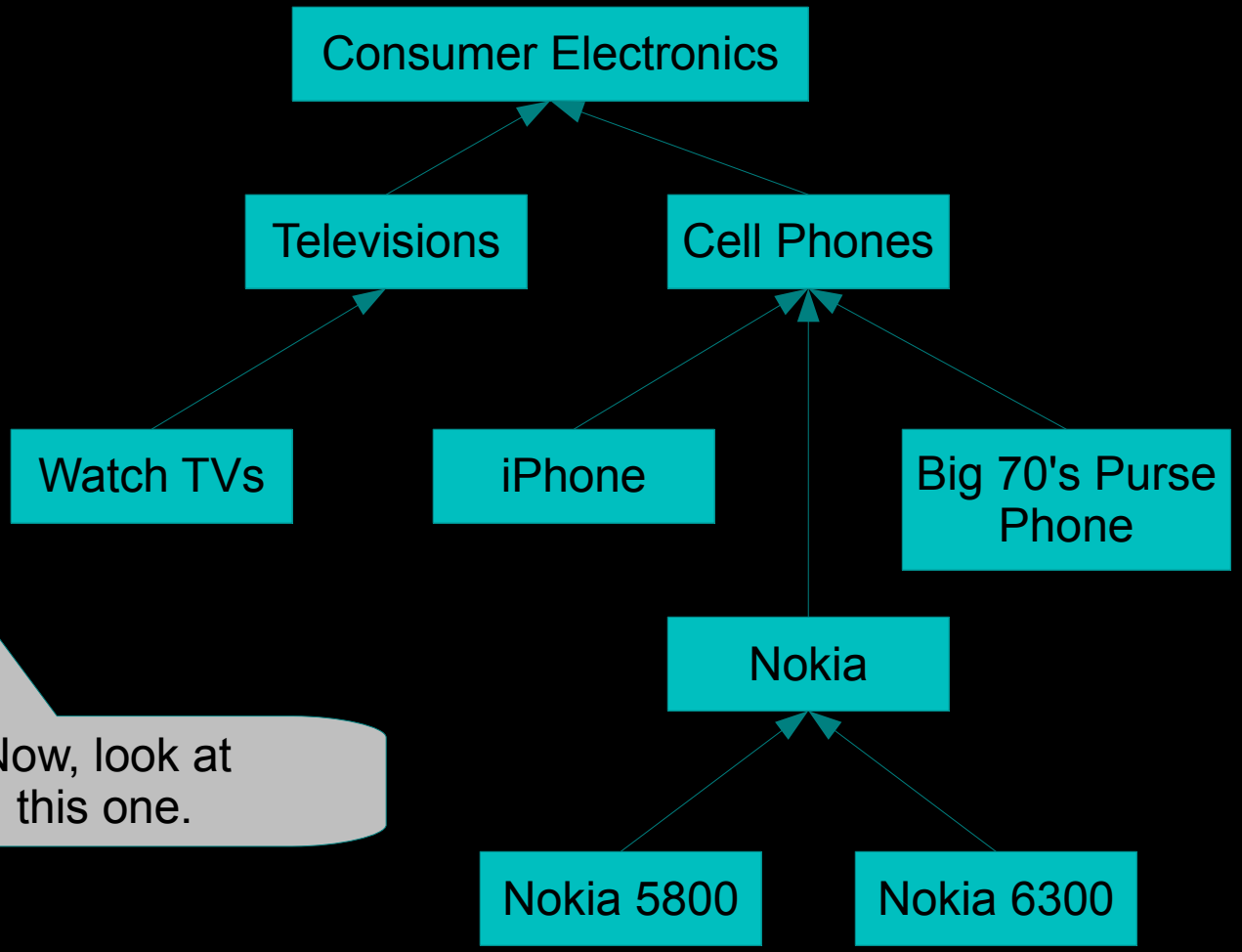
What!? Why?

More Motivation

Look at this
Class Heirarchy



More Motivation



More Motivation

Making a good domain heirarchy is hard, mmmKay?

For example, should I break down the cell phone heirarchy by vendor? By features? By color? By size?

And the animals I could break down by their taxonomical classification, or by whether they would be good pets, right?

This is called “the Tyranny of the Dominant Decomposition” by some smart guys.

Plus, will it blend?

More Motivation

Making a good domain heirarchy is hard, mmmKay?

For example, should I break down the cell phone heirarchy by vendor? By features? By color? By size?

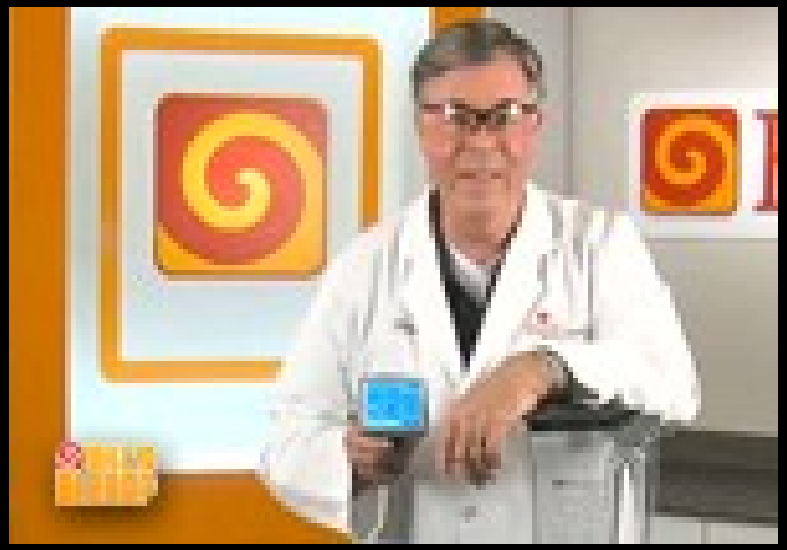
And the animals I could break down by their taxonomical classification, or by whether they would be good pets, right?

This is called “the Tyranny of the Dominant Decomposition” by some smart guys.

Plus, will it blend?

What?!

More Motivation



http://www.willitblend.com/

How do we represent blendability in these class heirarchies? It doesn't follow from the way we classified them.

More Motivation

Not Blendable: Elephants, Labradors, Televisions, and those big 70's purse phones

Blendable: Nokia 5800, Watch TVs, iPhone, Lizard, Squirrel, Chihuahua

So we need to attach blending code all throughout our heirarchies. This leads to Carpal Tunnel Syndrome, code duplication, and a maintenance nightmare.

mmmKay?

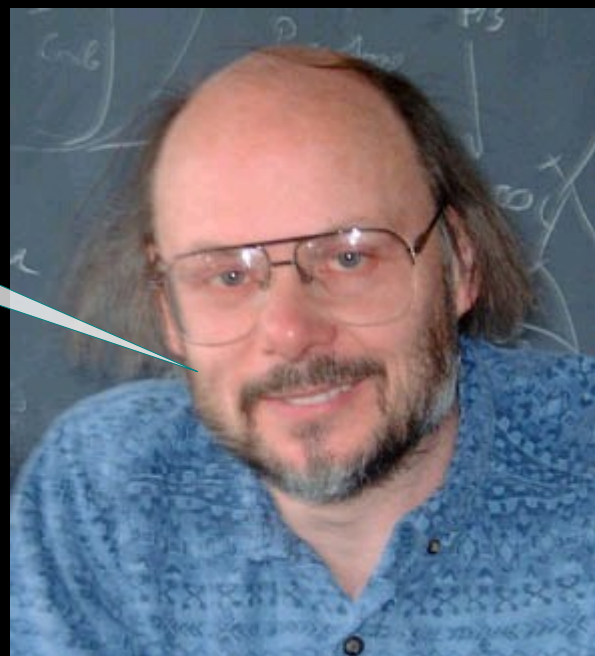
More Motivation

In Scala, you can have multiple, parallel, independant class heirarchies using traits (also called “Mixins”). They are like having interfaces that can implement methods.

More Motivation

In Scala, you can have multiple, parallel, independant class heirarchies using traits (also called “Mixins”). They are like having interfaces that can implement methods.

You can do that in C++

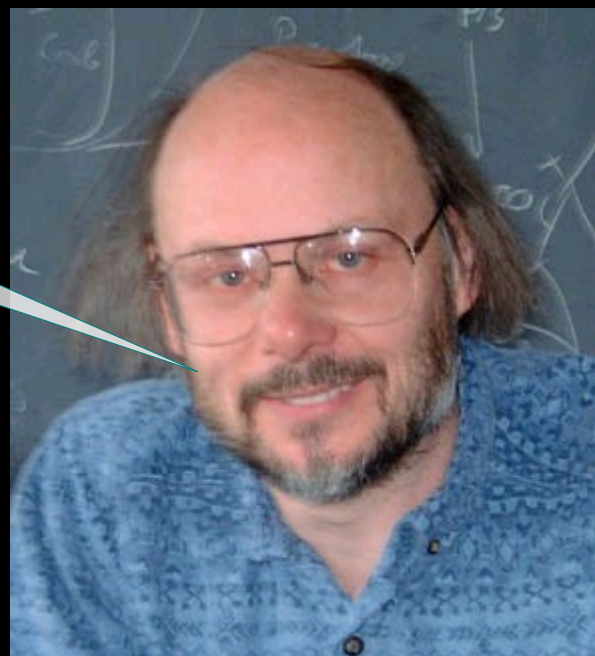


More Motivation

In Scala, you can have multiple, parallel, independant class heirarchies using traits (also called “Mixins”). They are like having interfaces that can implement methods.

You can do that in C++

Shuddup, mmmKay?



More Motivation

One of the reasons that Java did not implement multiple inheritance is because of the complete mess that it was in C++.

Other popular JVM languages decided to bypass the compile-time static type system altogether rather than deal with its problems

- Jython
- JRuby
- Clojure
- Groovy

In computer science, the term we use for this is “punting”, mmmKay?

More Motivation

Rather than running away from the type system, Scala embraced it and made it more powerful and flexible.

In addition to multiple inheritance, you get closures, implicit (but controllable) type conversions, type inferencing, duck typing, and generics with variance annotations.

More Motivation

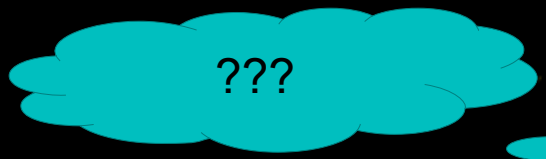
Rather than running away from the type system, Scala embraced it and made it more powerful and flexible.

In addition to multiple inheritance, you get closures, implicit (but controllable) type conversions, type inferencing, duck typing, and generics with variance annotations.

More Motivation

Rather than running away from the type system, Scala embraced it and made it more powerful and flexible.

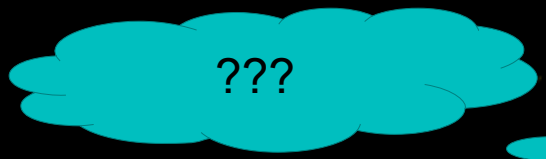
In addition to multiple inheritance, you get closures, implicit (but controllable) type conversions, type inferencing, duck typing, and generics with variance annotations.



More Motivation

Rather than running away from the type system, Scala embraced it and made it more powerful and flexible.

In addition to multiple inheritance, you get closures, implicit (but controllable) type conversions, type inferencing, duck typing, and generics with variance annotations.



Let's see some code, mmmKay?

Hello, World!

Hello, World!

```
package org.okcjug.hello

object HelloWorld extends Application {
  println(“Hello, World!”)
}
```

Hello, World!

```
package org.okcjug.hello

object HelloWorld extends Application {
  println(“Hello, World!”)
}
```

That's ugly!

Hello, World!

```
package org.okcjug.hello

object HelloWorld extends Application {
  println(“Hello, World!”)
}
```

That's ugly!

Make it more like Java.

Hello, World!

Hello, World!

```
package org.okcjug.hello

object HelloWorld {
  def main(args: Array[String]): Unit = {
    var greeting: String = “”;
    for (i: Int <- 0 until args.length) {
      greeting += (args(i) + “ “);
    }
    if (args.length > 0) greeting =
          greeting.substring(0, greeting.length - 1);
    println(greeting);
  }
}
```

Hello, World!

```
package org.okcjug.hello

object HelloWorld {
  def main(args: Array[String]): Unit = {
    var greeting: String = “”;
    for (i: Int <- 0 until args.length) {
      greeting += (args(i) + “ “);
    }
    if (args.length > 0) greeting =
          greeting.substring(0, greeting.length - 1);
    println(greeting);
  }
}
```

Don't need semi-colons

Hello, World!

```
package org.okcjug.hello

object HelloWorld {
  def main(args: Array[String]): Unit = {
    var greeting: String = “”
    for (i: Int <- 0 until args.length) {
      greeting += (args(i) + “ “)
    }
    if (args.length > 0) greeting =
          greeting.substring(0, greeting.length - 1)
    println(greeting)
  }
}
```

Hello, World!

```
package org.okcjug.hello

object HelloWorld {
  def main(args: Array[String]): Unit = {
    var greeting: String = “”
    for (i: Int <- 0 until args.length) {
      greeting += (args(i) + “ “)
    }
    if (args.length > 0) greeting =
          greeting.substring(0, greeting.length - 1)
    println(greeting)
  }
}
```

Types can be inferred

Hello, World!

```
package org.okcjug.hello

object HelloWorld {
  def main(args: Array[String]) = {
    var greeting = “”
    for (i <- 0 until args.length) {
      greeting += (args(i) + “ “)
    }
    if (args.length > 0) greeting =
          greeting.substring(0, greeting.length - 1)
    println(greeting)
  }
}
```

Hello, World!

```
package org.okcjug.hello

object HelloWorld {
  def main(args: Array[String]) = {
    var greeting = “”
    for (i <- 0 until args.length) {
      greeting += (args(i) + “ “)
    }
    if (args.length > 0) greeting =
          greeting.substring(0, greeting.length - 1)
    println(greeting)
  }
}
```

Replace looping code with built-in Array functions

Hello, World!

```
package org.okcjug.hello

object HelloWorld {
  def main(args: Array[String]) = {
    var greeting = “”
    greeting += args.mkString(“ “)
    println(greeting)
  }
}
```

Replace loooping code with built-in Array functions

Hello, World!

```
package org.okcjug.hello

object HelloWorld {
  def main(args: Array[String]) = {
    var greeting = “”
    greeting += args.mkString(“ “)
    println(greeting)
  }
}
```

Hello, World!

```
package org.okcjug.hello

object HelloWorld {
  def main(args: Array[String]) = {
    var greeting = “”
    greeting += args.mkString(“ “)
    println(greeting)
  }
}
```

No need for intermediate variables now

Hello, World!

```
package org.okcjug.hello

object HelloWorld {
  def main(args: Array[String]) = {
    println(args.mkString(“ “))
  }
}
```

Scala vs. Java

Scala vs. Java Files

Scala classes do not need to match the filename they are in. Can have multiple classes per file.

Java classes match the file names they are in

Scala vs. Java
Values

In Scala, every value is an object that you can call methods on.

```
// These two expressions are equivalent
1 + 8 / 4
1.add(8.divide(4))
// This is valid Scala
123.toString
```

Java has objects, primitives, operators, etc

Scala vs. Java
Method Nesting

```
// In Scala, classes and method definitions can be nested inside
// each other
class Dancer {
  def dance = {
    turn; step; step; turn; turn; step

    def turn = { ... }
    def step = { ... }
  }
}
```

Scala vs. Java
Types and Assignment

```
var s: String = “boo”

public String s = “boo”;
```

Scala vs. Java
Types and Assignment

```
private var s: String = “boo”

private String s = “boo”;
```

Scala vs. Java
Types and Assignment

```
protected var s: String = “boo”
// access modifiers can specify package names it applies to
protected[mypackage] var t: String = “blah”
```

```
protected String s = “boo”;
```

Scala vs. Java
Types and Assignment

```
val s: String = “boo”
```

```
public final String s = “boo”;
```

Scala vs. Java
Types and Assignment

```
val s: String = “boo”
```

```
public final String s = “boo”;
```

But, we know that the type can be inferred

Scala vs. Java
Types and Assignment

```
val s = “boo”
```

```
public final String s = “boo”;
```

Scala vs. Java
Classes

```
class Foo(bar: Bar) {
  // ...
}
```

```
public class Foo {

  public Foo(Bar bar) {
    // ...
  }

  // ...
}
```

Scala vs. Java
Classes

```
class Foo(bar: Bar) {
  // default constructor and method definitions
}
```

The body of the class *is* the default constructor

```
public class Foo {

  public Foo(Bar bar) {
    // contructor expressions here
  }

  // method definitions here
}
```

Scala vs. Java
Classes

```
class Foo(bar: Bar) {

  var password: String = “”
  bar.bend()

  def setPassword(p: String) = {
    password = p
  }

}
```

Initializations go in the body of the class. Methods are also defined here.

Scala vs. Java
Classes

```
class Foo(bar: Bar) {

  var password: String = “”
  bar.bend()

  def this(bar: Bar, pwd: String) {
    this(bar)
    password = pwd
  }

  def setPassword(p: String) = {
    password = p
  }

}
```

Secondary constructors (defined with “this”) must call the default constructor

Scala vs. Java
Overriding

```
// Scala has an override keyword
class Foo extends Bar {
  override def baz(p: String) = { ... }
}

// Java uses an annotation
class Foo extends bar {
  @Override public void baz(String p) { ... }
}
```

Scala vs. Java
static?

```
// Scala has no static.  Instead, use companion objects
object Foo {
  def x = { ... }
}

class Foo {
  def z = { ... }
}

// Java explicitly specifies static methods
class Foo {
  public static x() { ... }
  public z() { ... }
}
```

Scala vs. Java
Abstract Classes

```scala
/*  Methods do not need to be marked as abstract, as this
 *  is inferred from the lack of a method body
 */
abstract class Foo {
  def encrypt(x: String): String
}
```

```java
abstract class Foo {
  public abstract String encrypt(String x);
}
```

Scala vs. Java Interfaces/Traits

```
/*  Scala equivalent of an interface is an abstract trait with
 *  no method implementations
 */
abstract trait Foo {
  def encrypt(x: String): String
}

//  But traits can implement methods if desired
trait Bar {
  def unencrypt(s: String) = {
    rot13(s)
  }
}

interface Foo {
  public String encrypt(String x);
}
```

Scala vs. Java
Inheritance

```
// Zorg and Yog are traits, Bar can be either trait or class
class Foo extends Bar { ... }
class Gor extends Bar with Zorg with Yog { ... }
```

```
// Zorg and Yog are interfaces here
class Foo extends Bar { ... }
class Baz implements Zorg, Yog { ... }
class Gor extends Bar implements Zorg, Yog { ... }
```

Scala vs. Java
Packages

```
package org.okcjug.example
```

```
package org.okcjug.example;
```

Scala vs. Java
Packages

```
/*  Packages do not have to correspond to folders/directories
 *  But, it is recommended they do for your own sanity
 */
package org.okcjug.example
```

```
package org.okcjug.example;
```

Scala vs. Java
Imports

```
/*  Scala imports are not limited to the top of a source file
 *  They can occur inside classes, methods, or any other
 *  scope as needed.
 */
import org.okcjug.Wine           // single import
import org.okcjug._              // wildcard
import org.okcjug.{Wine,Soda}    // multiple imports
import org.okcjug.{Soda => Pop} // renaming import
```

```
import org.okcjug.Wine;  // single import
import org.okcjug.*;     // wildcard
```

Scala Idioms
Getters/Setters

```
// Don't bother with getters/setters
class Foo {
  var x = 1
  var y = 3
  var z = “foo”
}

val f = new Foo()
f.x = 3
f.y = f.x
// f.z = f.x  - compiler error
println(f.z)
```

Scala Idioms
Getters/Setters

```
// redefine a property to a method later if you need it
class Foo {
  var x = 1
  // def y = 3
  var z = “foo”
  private var yp = 3
  def y = { yp }
  def y_=(y: Int) { yp = y + 1 } // evil setter
}

val f = new Foo()
f.x = 3
f.y = f.x
// f.z = f.x  - compiler error
println(f.z)
```

Scala Idioms
Case Classes

```
// using a case class gives you a lot of function very compactly
case class Foo(x: Int, y: Int, z: String)

val f = Foo(3,3,”blah”)

/*
   => declarative constructor by implementing apply(...)
   => field accessors and setters
   => default toString(), equals() and hashcode() methods
   => pattern matching by implementing unapply(...)

 */
```

It's Time for a Code Break Basics

Scala vs. Java
Loops

```
// Java has loops
do {
  something();
} while (check == true)

// or this
for (Person p: bus.getPassengers()) {
  collectFare(p)
}
```

Scala vs. Java
Loops

```
// Scala has no such thing as loops
```

Scala vs. Java
Loops

```
// Scala has no such thing as loops
```

wait, wat?

Scala vs. Java
Loops

```
// Scala has no such thing as loops
// But, it can use closures to give you something that looks
// a lot like  a loop (but without 'break' or 'continue')
do {
  something()
} while (check == true)

// or
for (p <- bus.getPassengers) {
  collectFare(p)
}
// equivalent to
bus.getPassengers.foreach(p => collectFare(p))
// equivalent to
bus.getPassengers.foreach(collectFare)
```

Scala Idioms
Closures

```
// I can pass a String to a method like this
o.myMethod(“blah blah”)

// I can also pass a code block that evaluates to a String
o.myMethod( { if x == 1 “ONE” else “SOMETHING ELSE” } )

// I can pass a function to a method if it is defined to take it
def myMethod( x: (Int) => String )

myMethod( (z) => { z.toString } )
```

It's Time for a Code Break
Closures

Scala vs. Java Switch Statements

```
// Java can switch on ints or enums
switch (numberOfDogsOwned) {
  case 0:
    println(“Why no dogs?”);
    break;
  case 1:
    println(“Good choice”);
    break;
  case 2:
    println(“Crowded house”);
    break;
  default:
    println(“Call the pound”);
    break;
}
```

Scala vs. Java
Pattern Matching

```
// Scala uses the 'match' keyword
numberOfDogsOwned match {
  case 0 => println(“Why no dogs?”)
  case 1 => println(“Good choice”)
  case 2 => println(“Crowded house”)
  case _ => println(“Call the pound”)
}
```

Scala vs. Java
Pattern Matching

```
// Ever wanted to match on a String?
dog.name match {
  case “Fido” => println(“That's a good boy!”)
  case “Fifi” => println(“What a little dog”)
  case x: String => println(“You're dog is named ” + x)
  case _ => throw error(“Dog name is null”)
}
```

Scala vs. Java
Pattern Matching

```
// Or match on a class and the constructor parameters
dog match {
  case PitBull(x) => println(“Easy, boy... ” + x)
  case Poodle(“Fifi”) => println(“What a little dog named Fifi”)
  case x: Terrier(y) => { x.giveTreat; println(“Good boy, “ + y) }
  case Dog(x) => println(“You're dog is named ” + x)
}
```

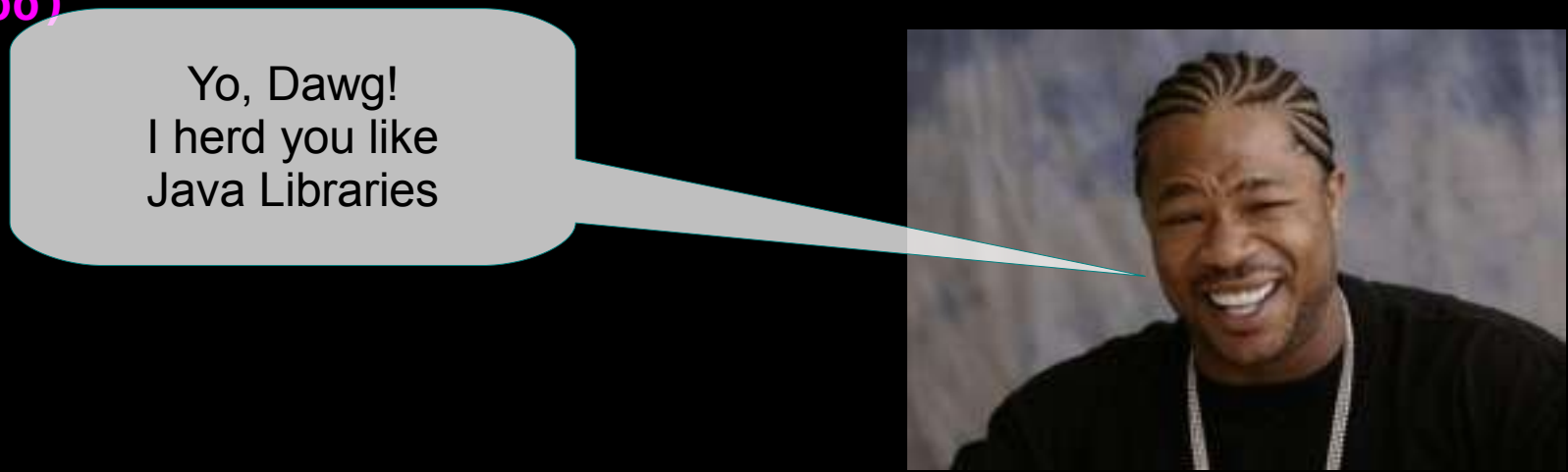
It's Time for a Code Break Patterns

Scala Idioms
Implicit Conversions

```
// If an implicit method like this is in scope
implicit def fooToBar(f: Foo) = { new Bar(f) }

// and a method that takes a Bar is defined
def myMethod(b: Bar) = { ... }

// I can pass it a Foo
myMethod(new Foo)
```

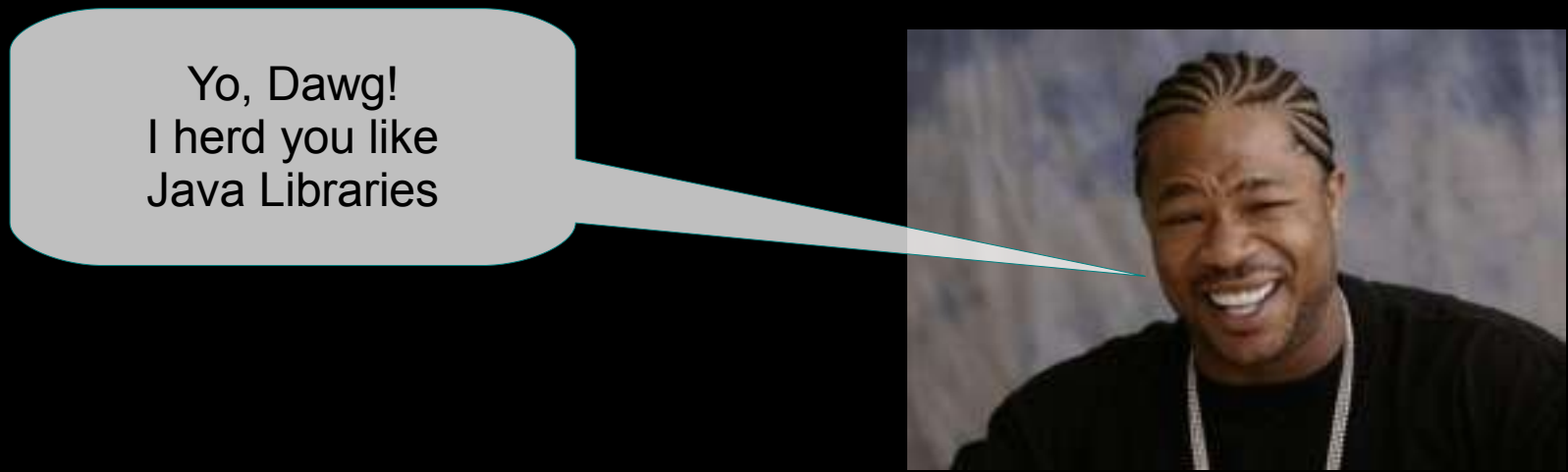
Scala Idioms
Implicit Conversions

```
// If an implicit method like this is in scope
implicit def fooToBar(f: Foo) = { new Bar(f) }

// and a method that takes a Bar is defined
def myMethod(b: Bar) = { ... }

// I can pass it a Foo
myMethod(new Foo)
```



It's Time for a Code Break
Implicit Conversions

(a.k.a - Pimp my Library)



Scala Idioms
For Comprehensions

```
// Another name for “fancy for statements”
// To participate an object must implement certain methods:
//   => map(), flatMap(), filter()
for (x <- xListOrSequence; y <- foo.getStuff if y.isValid) yield x

// Using the “yield” keyword makes a new sequence
// Or, you can just follow the for with an expression to evaluate
// it for everything in the list (think Java fors)
for (i <- 0 to 100) { println(i) }
```

Scala Idioms
XML Syntax

```
// You can put embed XML directly into Scala code.
// It gets compiled to an Elem() object
val doc = <html>
            <head>Who needs JSP?</head>
            <body/>
          </html>

// You can also embed scala code in the xml (with no limit on
// nesting)
val doc = <html><head>{ foo.title }</head></html>
```

It's Time for a Code Break
XML, For Comprehensions

Scala Idioms
Duck Typing

```
// You can define a method to take a Duck object
def pluck(d: Duck) = { ... }

// Or, if it looks like a Duck, quacks like a Duck ...
def pluck(d: { val feathers: Any; def quack(): Unit }) = { ... }
```

It's Time for a Movie Break
Duck Typing

Scala Idioms
Type Variances/View Bounds

```
// A List is covariant with its elements
// Or, a list of ducks is a sub-class of a list of animals
class List[+A] { ... }

// Sometimes you need bounds, or any super-class of Duck
def foo[T >: Duck](o: Option[T]) = { ... }

// Or, what if you want any type that can be implicitly converted
// to a Duck (view bounds)
def pluck[T <% Duck](d: Option[T]) = { ... }
```

Scala Idioms
Infix Types

```
// This
type Foo[Bar,Baz]

// is the same as this
type Bar Foo Baz

// Sometime syntactic sugar is fun
```

Do we have time for a confusing example?

Scala on your own
For further reference

=> The Scala website has tons of documentation
=> This is the best syntax overview I've seen:
    http://jim-mcbeath.blogspot.com/2008/09/scala-syntax-primer.html

Stuff I didn't cover:
=> Actors (concurrency model)
=> Lift web framework
=> Specs

Q&A => Either[Answers,BlankStares]