an alternative JVM language

GABRIEL-CALIN LAZAR


java?
scala?
what?
Scala
Java
Just give a try...

coursera

COURSES UNIVERSITY

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Functional Programming Principles in Scala

Martin Odersky

Learn about functional programming, and how it can be effectively combined with object-oriented programming. Gain practice in writing clean functional code, using the Scala programming language.

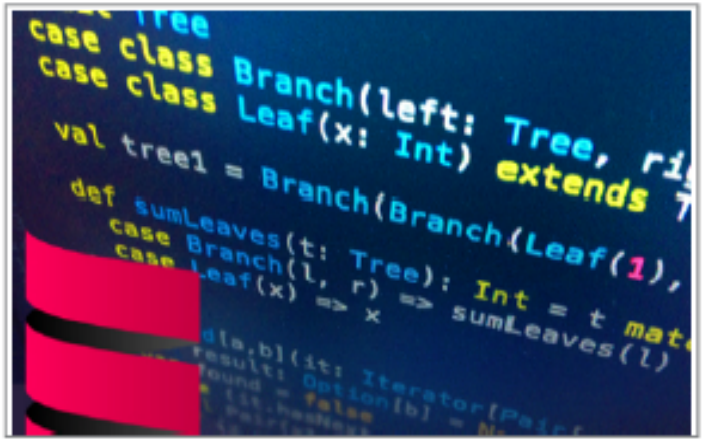
JUNE 03, 2013

Statement of Accomplishment

WITH DISTINCTION

GABRIEL-CALIN LAZAR

HAS SUCCESSFULLY COMPLETED THE ECOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE'S ONLINE OFFERING OF



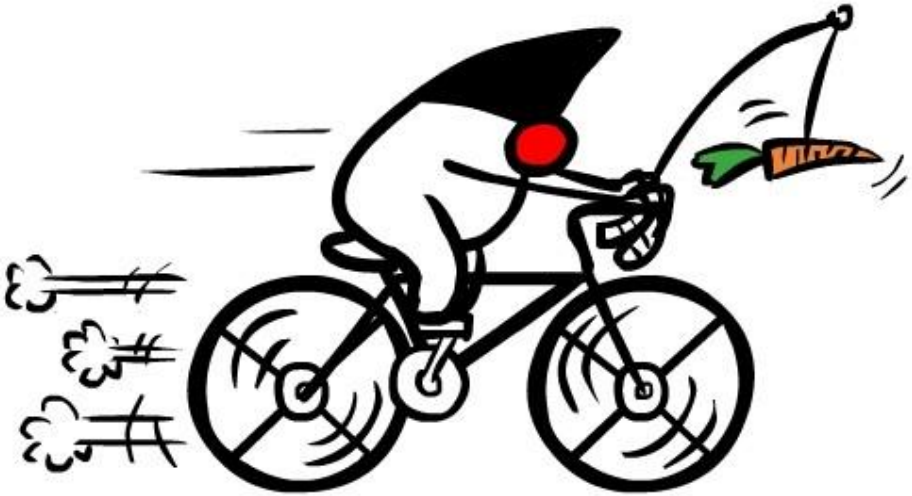
Functional Programming Principles in Scala

This advanced undergraduate programming course covers the principles of functional programming using Scala, including the use of functions as values, recursion, immutability, pattern matching, higher-order functions and collections, and lazy evaluation.


Lets get
Going!
™

good design decisions

continuous improvement

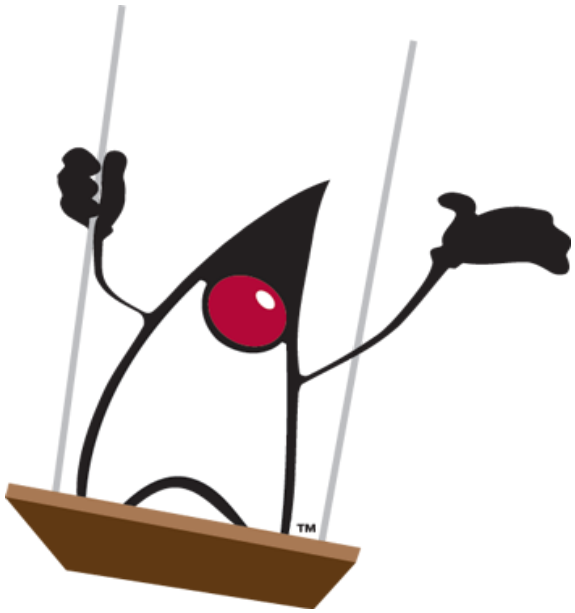


JVM rocks


Scala
Jython
Groovy
JRuby
Clojure
ceylon
Kotlin

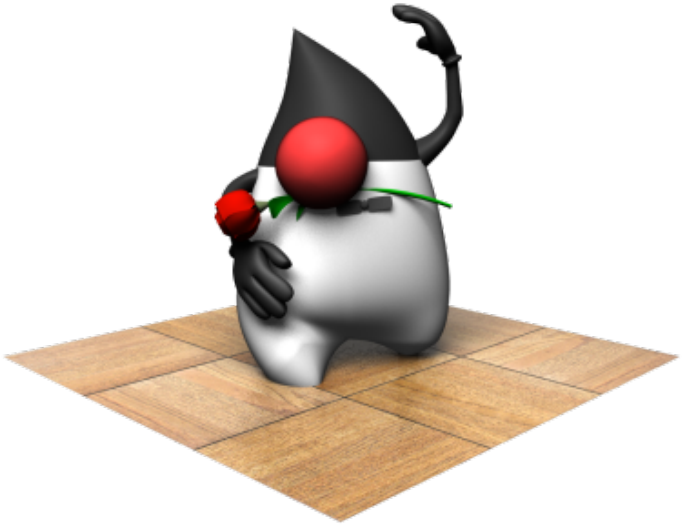
future

power



fun

skills



love

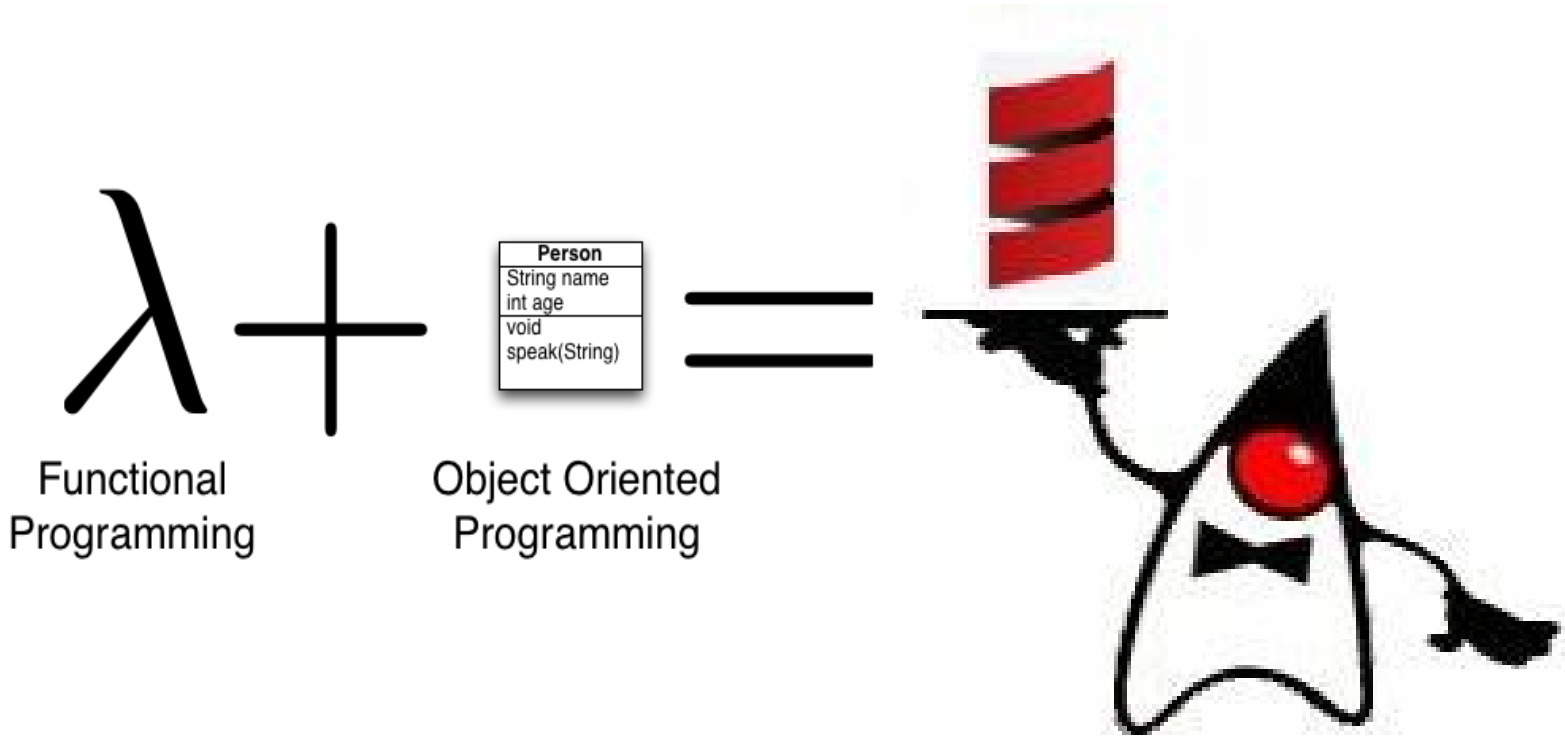
The future is already here.
***It's just not evenly distributed*.**

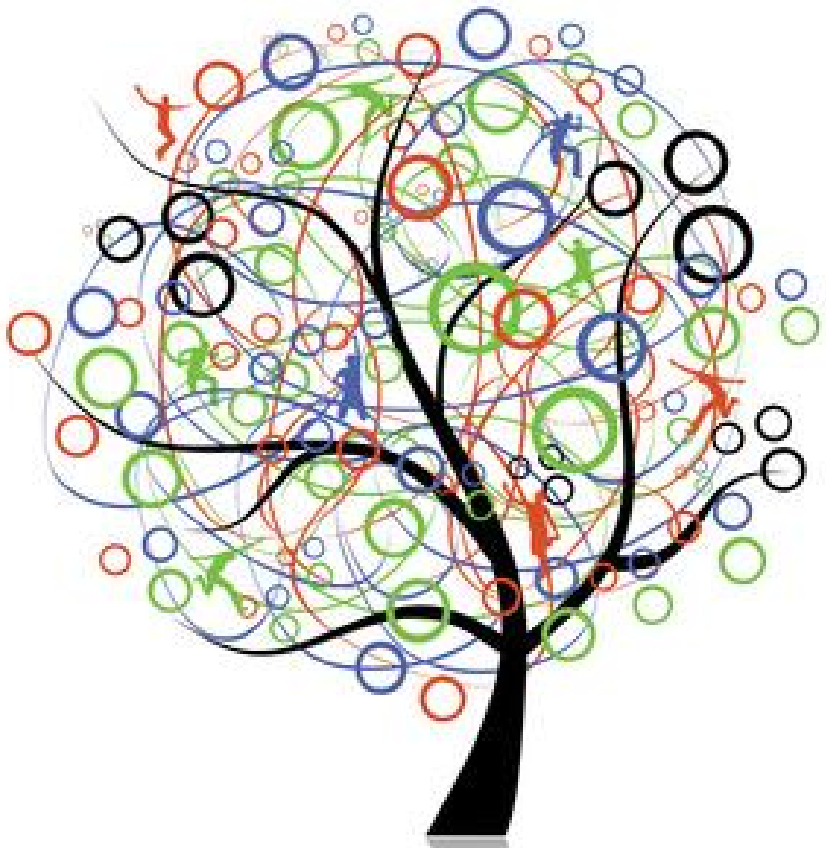
William Gibson


Scala
Language
Java
Programming
Code
Libraries
programmers
Compiler
functional
object-oriented
features
applications
Type
Types
Download
Example
classes
programmer
used
make
supports
Guides
Eclipse
Talks
Tools
smoothly
also
safe
problem
concise
every
use
Version
Scala's
new
Started

Martin Odersky

I have always tried to make Scala
a very powerful but at the same beautifully simple language,
by trying to find unifications of formerly disparate concepts.


Person
String name
int age
void
speak(String)
Functional
Programming
Object Oriented
Programming

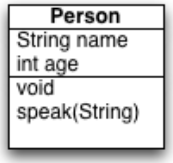


Scala

designed to be extended and adapted

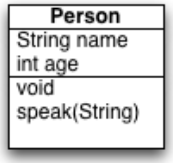
fast to first product, scalable afterwards

Functional
Programming



Object Oriented
Programming

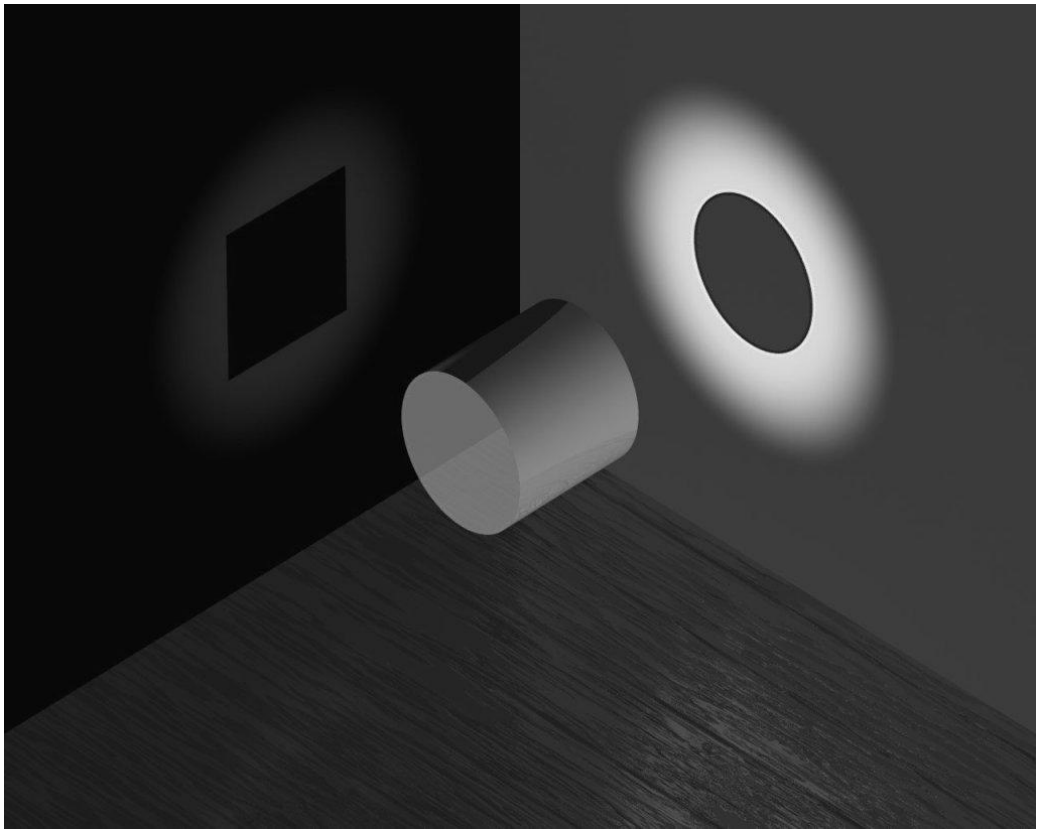
+

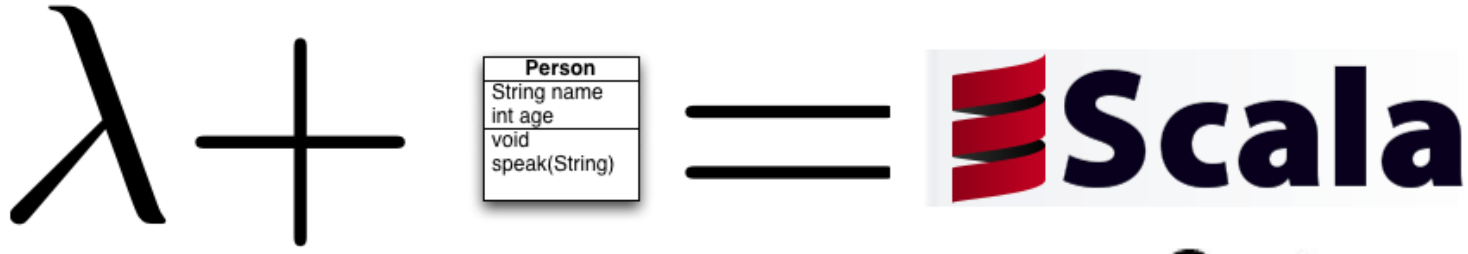


=

Functional Programming

Object Oriented Programming





Functional Programming

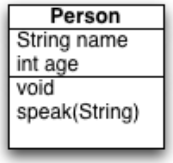
Object Oriented Programming

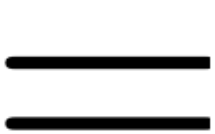
Functional Programming offers a very valuable *layer of abstraction*:

A generic control structure (Higher Order Function) that implements a reusable flow

```
val namesTopGrades = students.filter(_.grade >= 9).map(_.name)
```

A Function used by the generic control structure to customize a computation





Functional Programming

Object Oriented Programming

Imperative Java

```
public List<String> getShortCityNames(List<String>cities) {
    List<String> shortCityNames = new ArrayList<String>();
    for (String city : cities) {
       if (city.length() < 8) {
          shortCityNames.add(city);
       }
    }
    return shortCityNames;
}
```

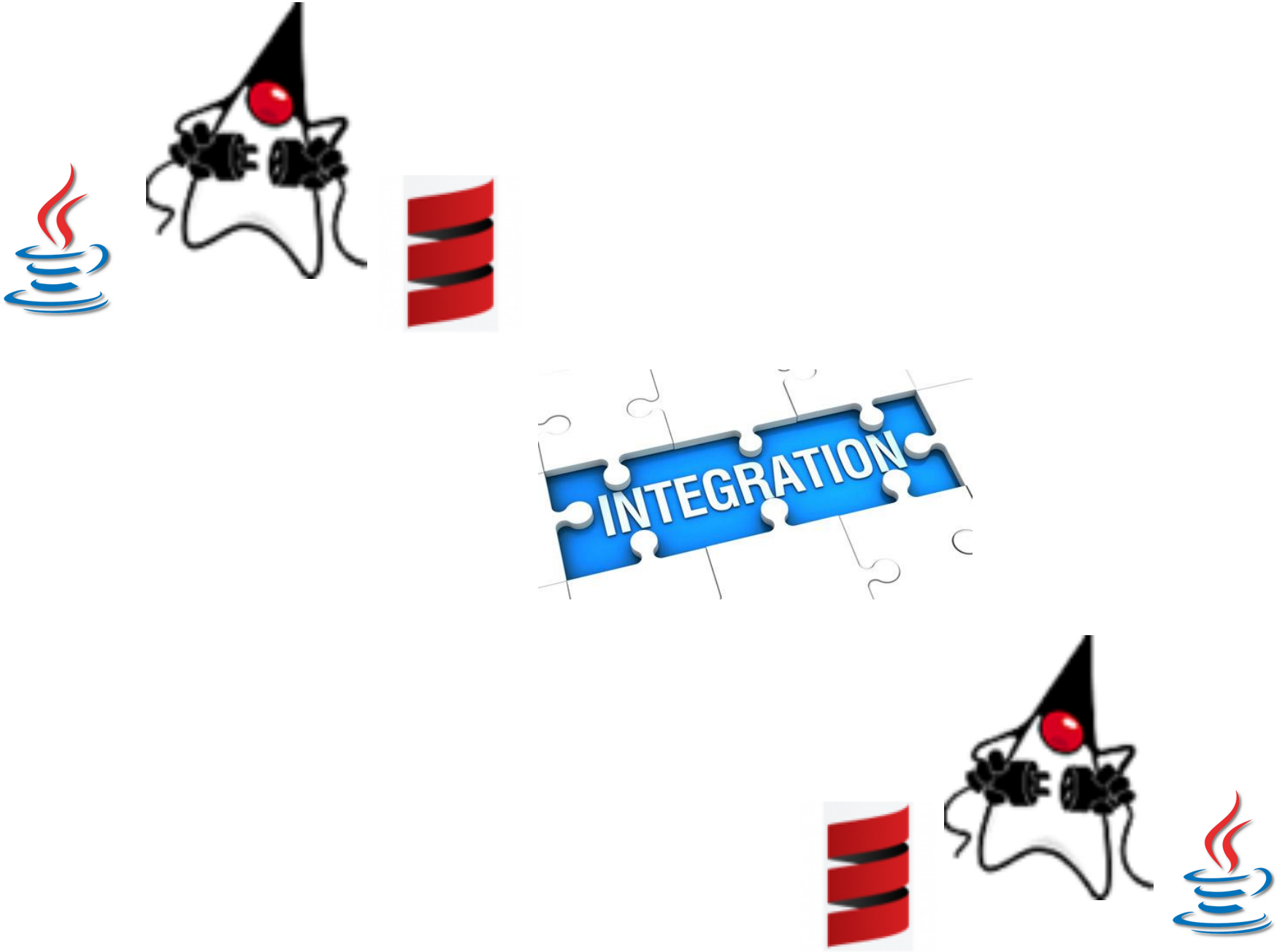
Functional Scala

```
val shortCities = cities.filter { city: String => city.length < 8 }
val shortCities = cities.filter { city => city.length < 8 }
val shortCities = cities.filter { _.length < 8 }
```


```
//total salary for all employees in accounting
val totalSalaryForAccountingDept = employees
                                   .filter(_.dept == "Accounting")
                                   .map(_.salary)
                                   .sum
```

NO


ALL IN
Scala
≠
Java


INTEGRATION

```scala
class CrazyScalaTechnology {
  def doWebThreeDotOhStuff(msg:String) {
    println(msg)
  }
}
```

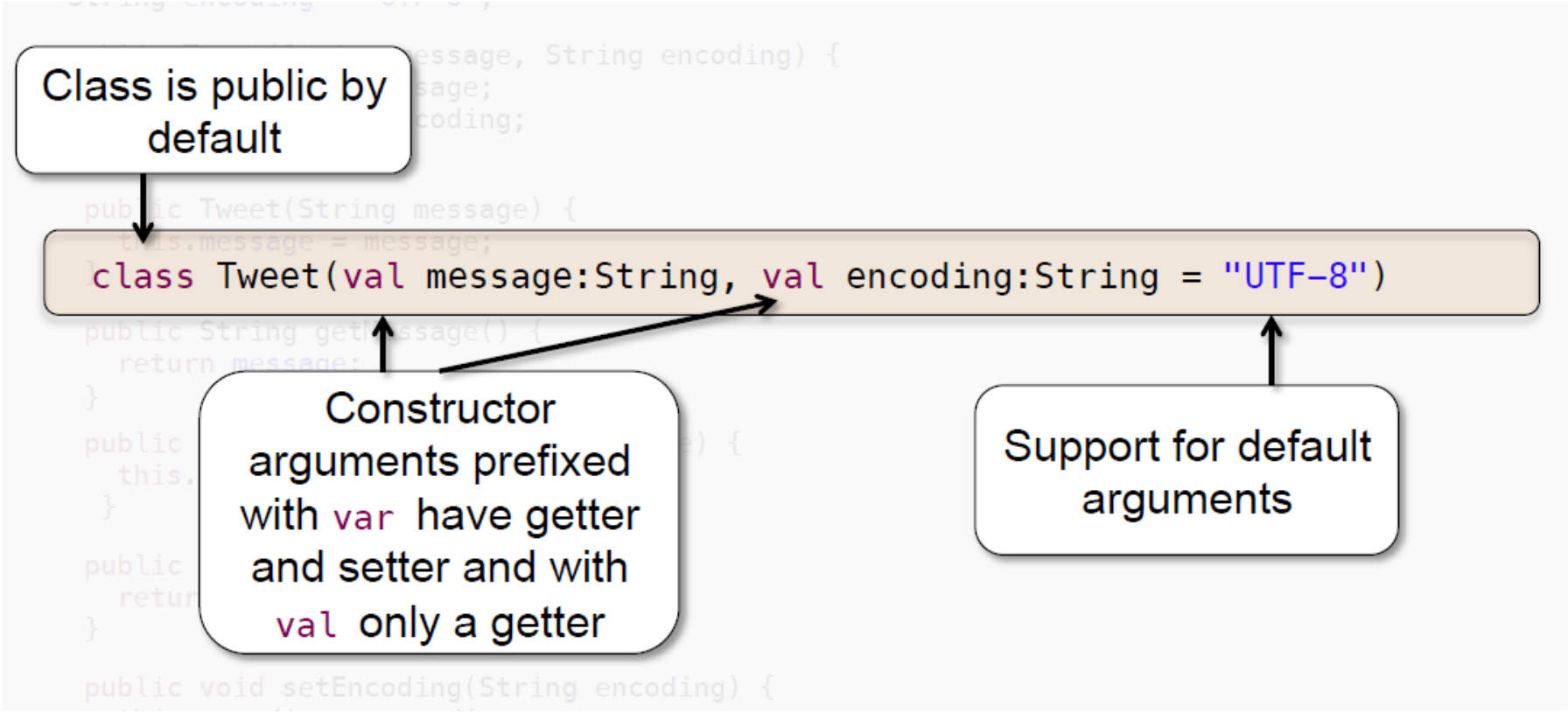
Scala

```java
public class JavaPojo {
   public static void main(String[] args) {
      new CrazyScalaTechnology().doWebThreeDotOhStuff("Hello world!");
   }
}
```

Java


```scala
import javax.servlet.http.HttpServlet
import javax.servlet.http.HttpServletRequest
import javax.servlet.http.HttpServletResponse

class ScalaServlet extends HttpServlet {

  override def doGet(request: HttpServletRequest,
                     response: HttpServletResponse) {

      response.getWriter().println("Hello World!")
  }

}
```


I Can't Believe
It's Not
Java™
!


Class is public by default
class Tweet(val message:String, val encoding:String = "UTF-8")
Constructor arguments prefixed with var have getter and setter and with val only a getter
Support for default arguments

```java
public class Circle {
 double radius;

 public Circle(double radius) {
    this.radius = radius;
 }

 void paint () { ... }

 boolean gt (Circle other) { ... }

 //getter & setter

 static Circle fromDiameter(double d) {
    return new Circle(d/2);
 }

}
```

```
class Circle(val radius:Double) {

 def paint

 def > (ot                                { ... }

}
```

Constructor arguments are fields when prefixed with `var` or `val`

```
object Circle {
 def fromDiameter(diameter:Double) =
    new Circle(diameter / 2)
}
```

```java
public class Circle {
 double radius;

 public Circle(double radius) {
    this.radius = radius;
 }

 void paint () { ... }

 boolean gt (Circle other) { ... }


 static Circle fromDiameter(double d) {
    return new Circle(d/2);
 }

}
```

Scala has no primitives. Everything is an Object.

```scala
class Circle(val radius:Double) {

 def paint:Unit = { ... }

 def > (other:Circle):Boolean = { ... }
}
```

```scala
object Circle {
 def fromDiameter(d:Double):Circle =
    new Circle(d / 2)
}
```

```java
public class Circle {
 double radius;

 public Circle(double radius) {
    this.radius = radius;
 }

 void paint () { ... }

 boolean gt (Circle other) { ... }


 static Circle fromDiameter(double d) {
    return new Circle(d/2);
 }

}
```

Every method has a return type. If the method is void the return type is **Unit**.

```
class Circle(val radius:Double) {

 def paint:Unit = { ... }

 def > (other:Circle):Boolean = { ... }
}
```

```
object Circle {
 def fromDiameter(d:Double):Circle =
    new Circle(d / 2)
}
```

```java
public class Circle {
 double radius;

 public Circle(double radius) {
    this.radius = radius;
 }

 void paint () { ... }

 boolean gt (Circle other) { ... }


 static Circle fromDiameter(double d) {
    return new Circle(d/2);
 }

}
```

Scala has no reserved operators. Every operator (*, /, +,-, >, < etc.) is a method. Symbolic operators are allowed (but discouraged).

```scala
class Circle(val radius:Double) {

 def paint:Unit = { ... }

 def > (other:Circle):Boolean = { ... }
}
```

```scala
object Circle {
 def fromDiameter(d:Double):Circle =
    new Circle(d / 2)
}
```

```java
public class Circle {
 double radius;

 public Circle(double radius) {
    this.radius = radius;
 }

 void paint () { ... }

 boolean gt (Circle other) { ... }


 static Circle fromDiameter(double d) {
    return new Circle(d/2);
 }

}
```

Scala has no statics. Scala supports objects, which are singletons. Usage:
`Circle.fromDiameter(2.0)`

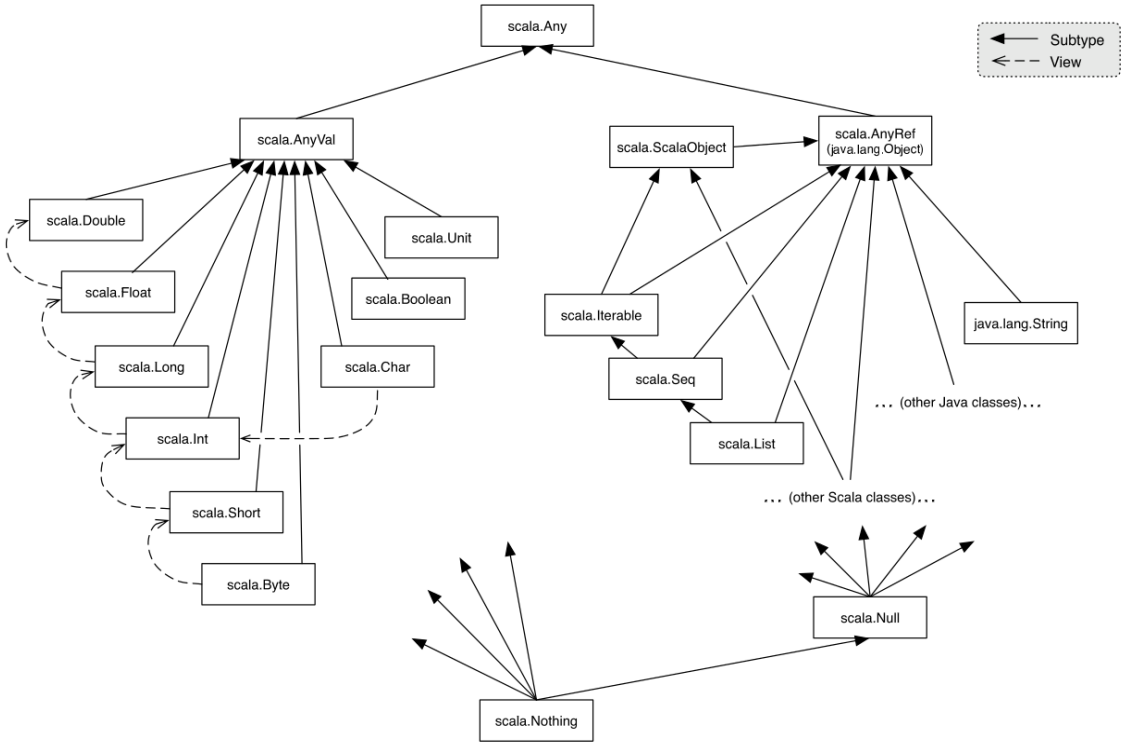
```scala
class Circle(val radius:Double) {

 def paint:Unit = { ... }

 def > (other:Circle):Boolean = { ... }
}
```

```scala
object Circle {
 def fromDiameter(d:Double):Circle =
    new Circle(d / 2)
}
```


MYTHBUSTERS
EXPOSING THE MYTHS
THAT MAKE US MISERABLE

I'm not against types, but
I don't know of any type systems that aren't a complete pain, so
***I still like dynamic typing*.**

Alan Kay



```java
Circle c1 = new Circle(2);
Circle c2 = new Circle(4);

List<Circle> circles = new ArrayList<>();
circles.add(c1);
circles.add(c2);

c1.gt(c2);
```

```
val c1 = new Circle(2)
val c2 = new Circle(4)

val circles = List(c1, c2)



c1 > c2
```

Types are optional in Scala.
…but you can declare them:

```
val c1:Circle = new Circle(2)
val circles:List[Circle] = List[Circle](c1,c2)
```

```
Circle c1 = new Circle(2);
Circle c2 = new Circle(4);

List<Circle> circles = new ArrayList<>();
circles.add(c1);
circles.add(c2);

c1.gt(c2);
```

Java 7 supports Type inference only for generics with the 'diamond' operator

```
val c1 = new Circle(2)
val c2 = new Circle(4)

val circles = List(c1, c2)

c1 > c2
```

Semicolon is inferred

Scala supports the ‘infix’ notation, mainly for operator-like syntax.
In the ends it’s a simple method call:
`c1.>(c2)`

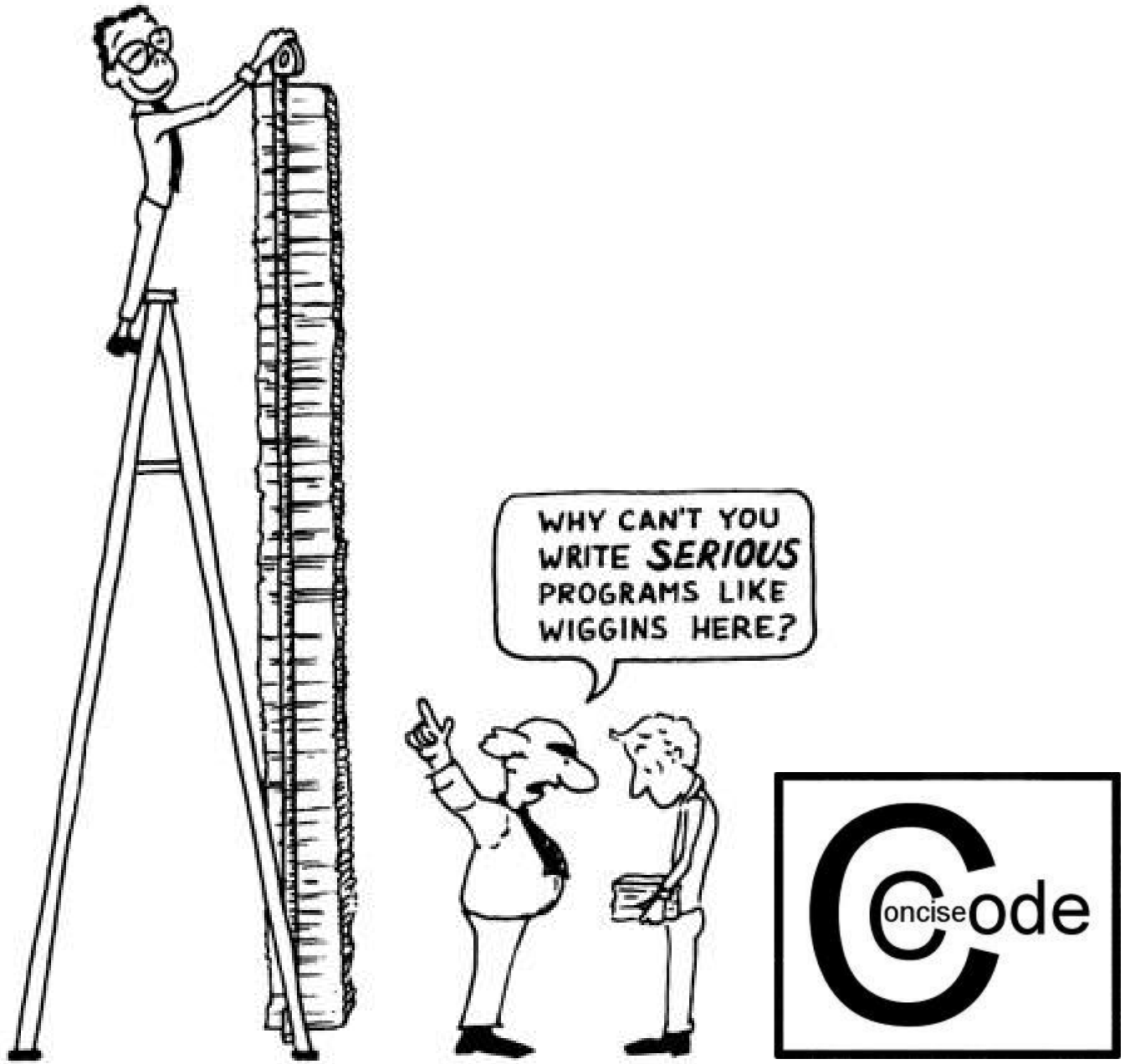
```
Circle c1 = new Circ
Circle c2 = new Circ

List<Circle> circles = new ArrayList<>();
circles.add(c1);
circles.add(c2);

c1.gt(c2);
```

```
ew Circle(2)
ew Circle(4)

es = List(c1, c2)

c1 > c2
```

Feels like a

dynamic scripting language

…but it's not


WHY CAN'T YOU WRITE SERIOUS PROGRAMS LIKE WIGGINS HERE?
Concise Code

```
val squareFunction = { x:Int => x * x }
```

Is just shorthand for

```
val squareFunction = new Function1[Int, Int]() {
  def apply(x:Int) = x * x
}
```

Java 8 Lambda Expressions and Scala Functions seem identical.

Higher order functions reveal the difference:

Java 8

```
interface Collection<T> {

  public Collection<T> filter(Predicate<T> p);

  public <R> Collection<R> map(Mapper<T, R> m);
}
```

```
interface Predicate<T> {
  public boolean op(T t);
}
```

```
interface Mapper<T, R> {
  public R map(T t);
}
```

in Java 8 a functional interface (with one method) must be declared for *every* higher order function.

Scala

```
trait Seq[T] {

  def filter(f:T => Boolean):Seq[T]

  def map[R](m:T => R):Seq[R]
}
```

In Scala no functional interface is needed because functions are 'first class citizens'

Using Lambda's/Functions as Objects reveal the difference too:

A Lambda always needs to be assigned to a **matching Functional Interface** in Java 8.

Java 8

```
Predicate<Integer> evenFilter = (Integer i) -> i % 2 == 0;
Predicate<Integer> evenFilter = NumUtils::isEven;

numbers.filter(evenFilter);
```

Due to Scala's Type Inference combined with Functions as 'First Class Citizens' approach this is not necessary:

Scala

```
val evenFilter = (i:Int) => i % 2 == 0

//evenFilter has type: Int => Boolean
numbers.filter(evenFilter)
```

Scala's Functions are composable:

```
val add = (i:Int) => (j:Int) => i + j

val multiply = (x:Int) => (y:Int) => x * y

val prettyPrint = (i:Int) => s"the result is $i"

val combi = add(5) compose multiply(2) andThen prettyPrint

//combi has type: Int => String
combi(10)
> the result is 25
```

compose and andThen are methods of the Scala Function object

```scala
def parseBoolean(a: Any):Boolean = {
  val stringTruthies = Set("true", "1", "yes")
  a match {
    case b:Boolean => b
    case i:Int => i == 1
    case s:String => stringTruthies.contains(s)
  }
}
```

```scala
case class Town(city:String, state:String)
case class Person(first:String, last:String, hometown:Town)

def getTownsThatHaveSmithsInVirginia(people:List[Person]) = {
  people.collect { person =>
      person match {
         case Person(_, "Smith", Town(town, "Virginia")) => town
         case _ =>
      }
  }
}
```

```
def getPerson(id:String):Option[Person] = {
   if (id == "person123") {
      Some(new Person("Bob"))
   }
   else {
      None
   }
}

//get the value, or use a default
val name = getPerson("person123").getOrElse("Name unknown")
```

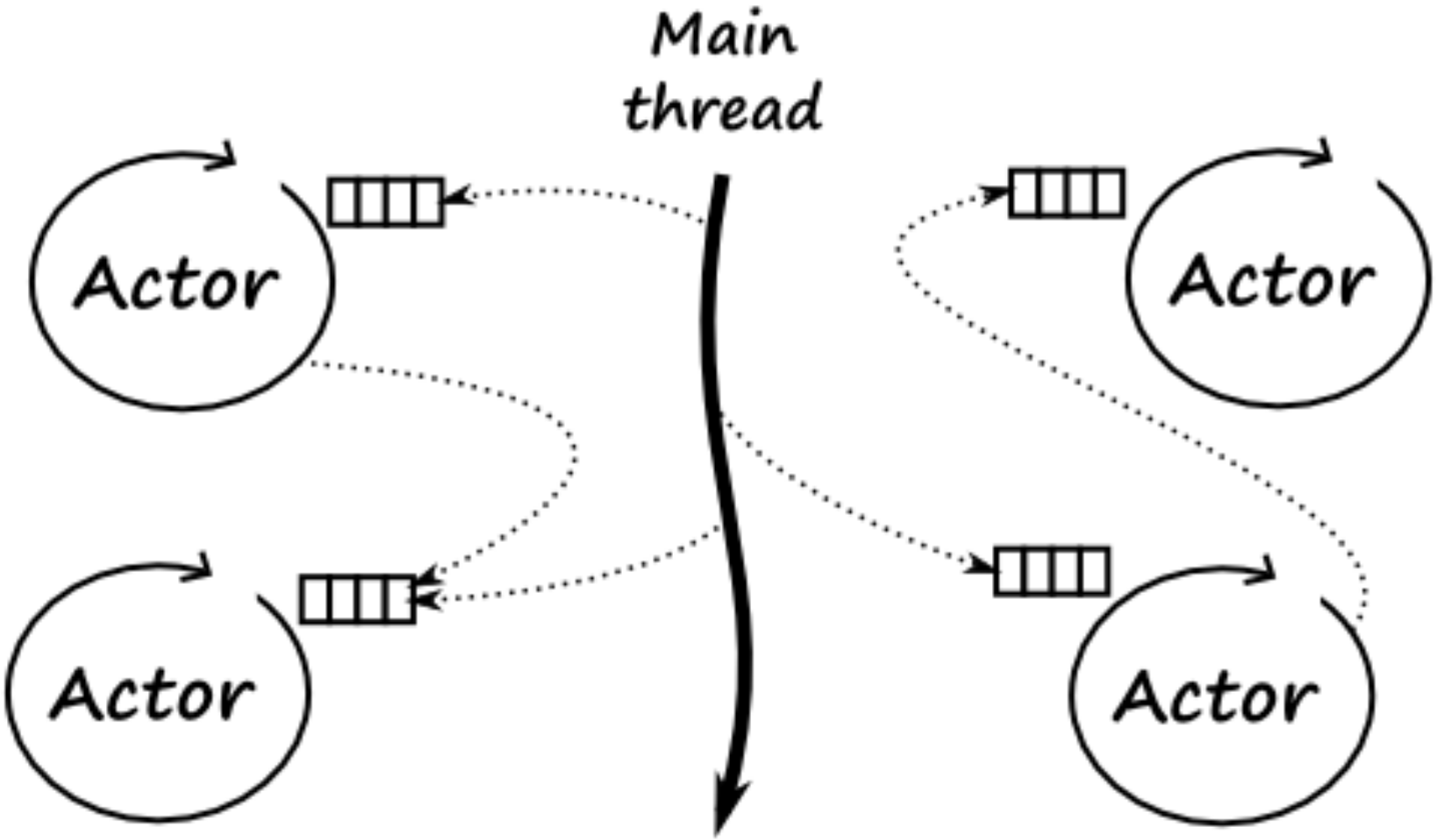
```
def transfer(fromAccount:Account, toAccount:Account, amount:MonetaryAmount) {

   transaction {
      fromAccount.debit(amount)
      toAccount.credit(amount)
   }

}

def transaction(transactionCode: => Unit)
```

```
  def sendEmail(
      to : List[String],
      from : String,
      subject : String = "No subject",
      cc : List[String] = Nil,
      bcc : List[String] = Nil,
      attachments : List[File] = Nil
  )


sendEmail(List("a@a.com"), "b@b.com")

sendEmail(to = List("a@a.com"),
          from = "b@b.com",
          subject ="Status Check!")

sendEmail(to = List("a@a.com"),
          from = "b@b.com",
          subject ="Status Check!",
          attachments = List(new File("coolpic.jpg")))
```

```
val cities =  <cities>
                  <city><name>{ city.name }</name></city>
                   ....
              </cities>

cities match {
      case <cities>{ cities @ _* }</cities> =>
         for (city <- cities) println("City:" + (city \ "name").text)
}
```

```
class Foo {
    private volatile Bar bar = null;

    public Bar getBar() {
        if (bar == null) {
            synchronized (this) {
                if (bar == null) {
                    bar = new Bar();
                }
            }
        }
        return bar;
    }
}
```


```
class Foo {
 lazy val bar = new Bar
}
```


Main
thread
Actor
Actor
Actor
Actor

akka

```
case object PingMessage
case object PongMessage
case object StartMessage
case object StopMessage
```

```
class Pong extends Actor {
  def receive = {
    case PingMessage =>
      println("  pong")
      sender ! PongMessage
    case StopMessage =>
      println("pong stopped")
      context.stop(self)
  }
}
```

```
class Ping(pong: ActorRef) extends Actor {
  var count = 0
  def incrementAndPrint {
    count += 1
    println("ping")
  }
  def receive = {
    case StartMessage =>
      incrementAndPrint
      pong ! PingMessage
    case PongMessage =>
      incrementAndPrint
      if (count > 99) {
        sender ! StopMessage
        println("ping stopped")
        context.stop(self)
      } else {
        sender ! PingMessage
      }
  }
}
```

```
object PingPong extends App {
  val system = ActorSystem("PingPong")
  val pong = system.actorOf(Props[Pong], name = "pong")
  val ping = system.actorOf(Props(new Ping(pong)), name = "ping")
  ping ! StartMessage
}
```

akka

```
case object PingMessage
case object PongMessage
case object StartMessage
case object StopMessage
```

```
class Pong extends Actor {
  def receive = {
    case PingMessage =>
      println("  pong")
      sender ! PongMessage
    case StopMessage =>
      println("pong stopped")
      context.stop(self)
  }
}
```

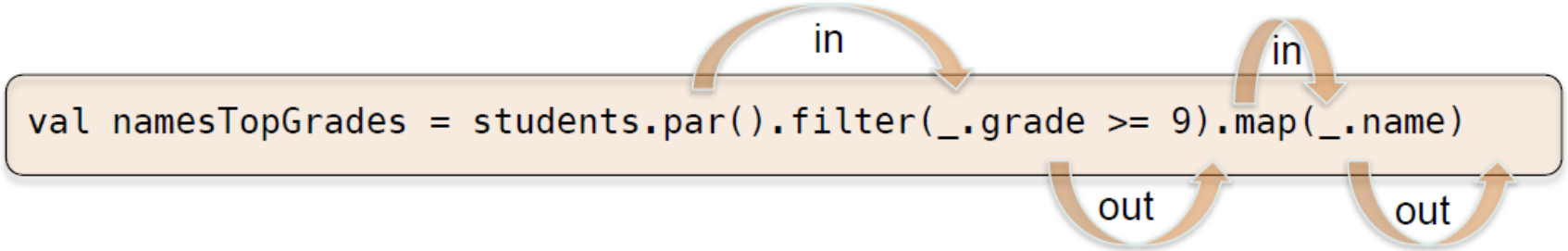
```
class Ping(pong: ActorRef) extends Actor {
  var count = 0
  def incrementAndPrint {
    count += 1
    println("ping")
  }
  def receive = {
    case StartMessage =>
      incrementAndPrint
      pong ! PingMessage
    case PongMessage =>
      incrementAndPrint
      if (count > 99) {
        sender ! StopMessage
        println("ping stopped")
        context.stop(self)
      } else {
        sender ! PingMessage
      }
  }
}
```

```
object PingPong extends App {
  val system = ActorSystem("PingPong")
  val pong = system.actorOf(Props[Pong], name = "pong")
  val ping = system.actorOf(Props(new Ping(pong)), name = "ping")
  ping ! StartMessage
}
```


SIDE
EFFECTS

To understand *functional programming* the programmer has to think differently:

Think in **transforming input to output**
(like pipes in unix)



… instead of: *manipulating state*

Classes should be immutable
unless there's a very good reason
to make them mutable.

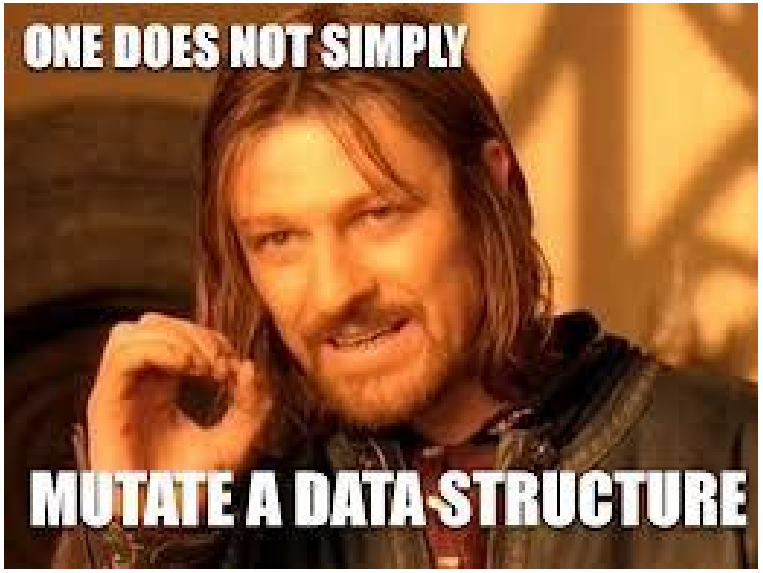
If a class cannot be made immutable,
limit its mutability as much as possible.

Joshua Bloch

Classes should be immutable
unless there's a very good reason
to make them mutable.

If a class cannot be made immutable,
limit its mutability as much as possible.

Joshua Bloch




eclipse

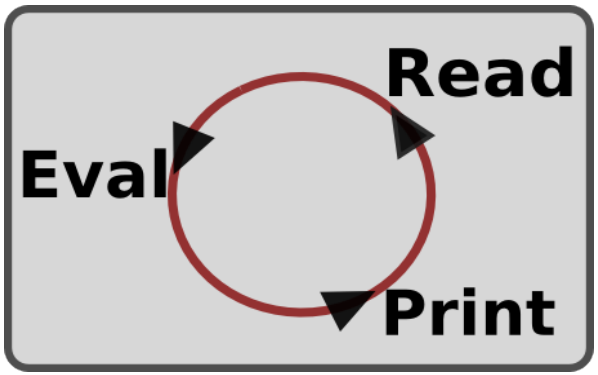

spring

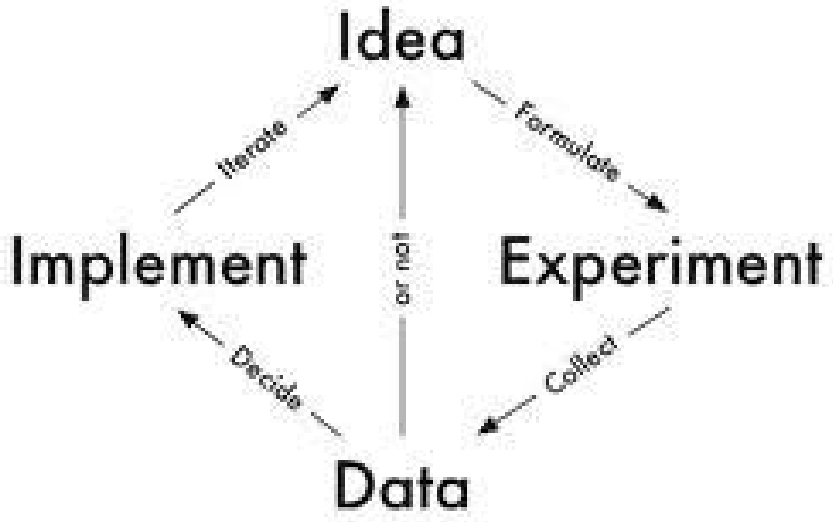

IntelliJIDEA

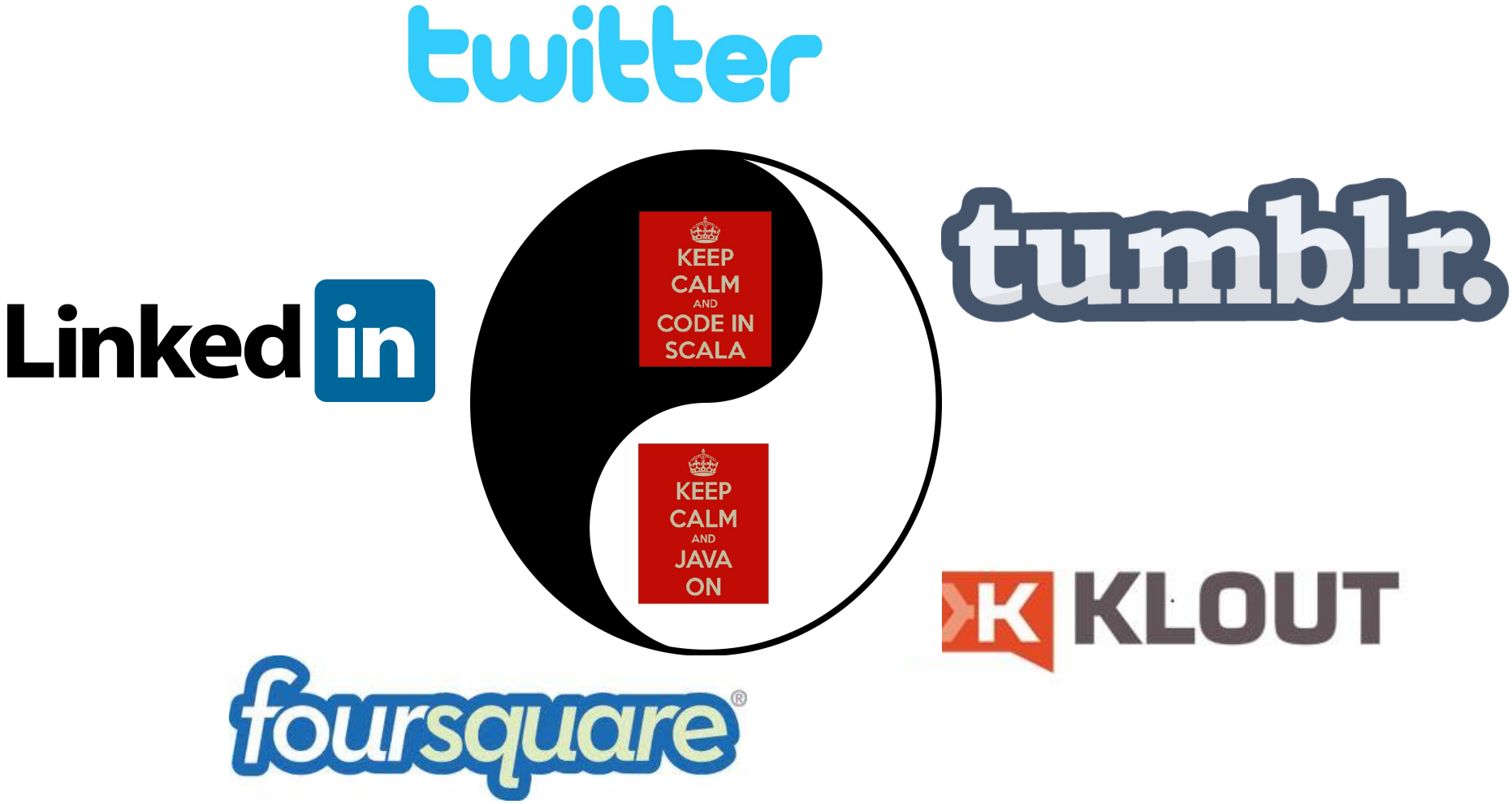

HIBERNATE

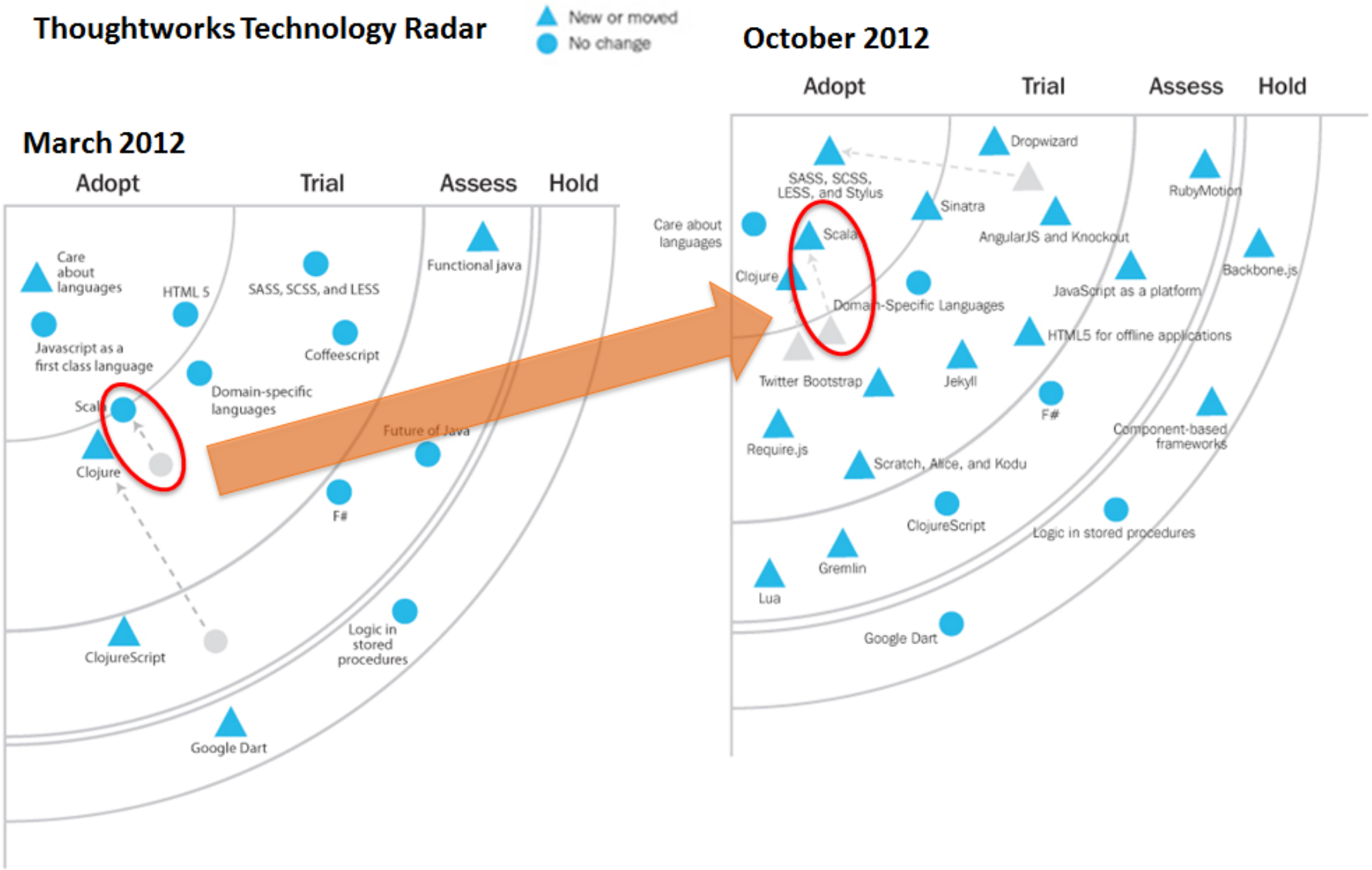

NetBeans


maven


Read
Eval
Print

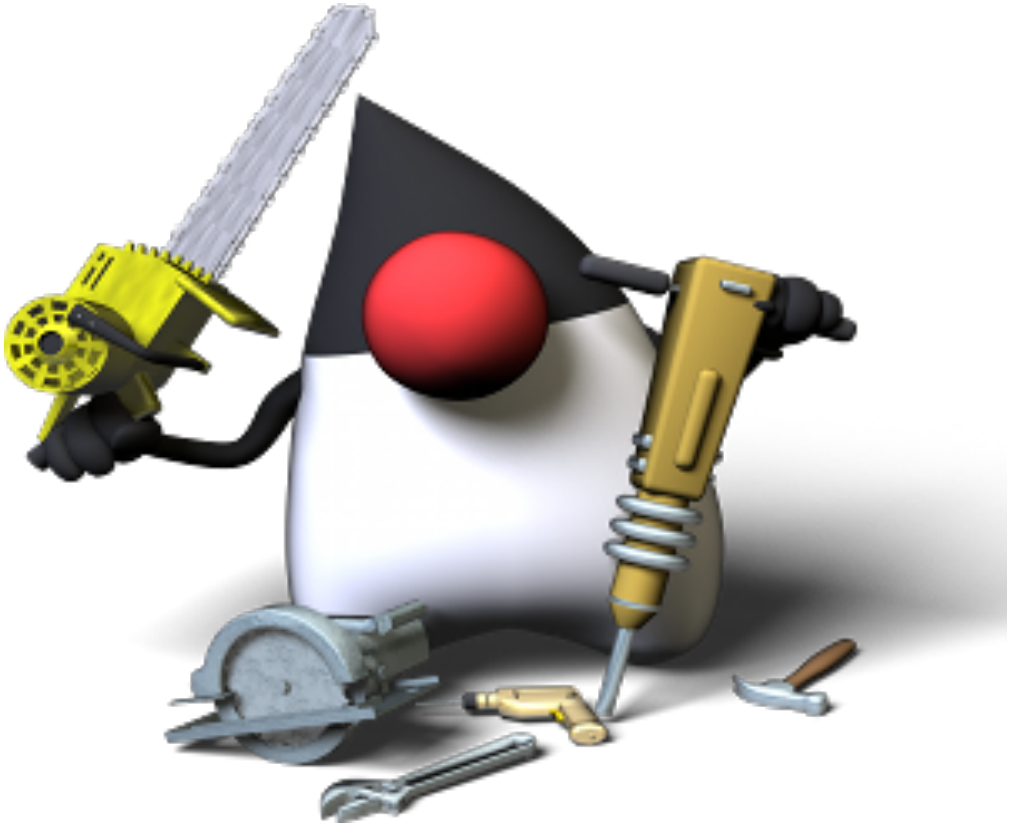

Idea
Iterate
Formulate
Implement
run
Experiment
Decide
Collect
Data


twitter
LinkedIn
KEEP CALM AND CODE IN SCALA
KEEP CALM AND JAVA ON
tumblr.
KLOUT
foursquare


Thoughtworks Technology Radar
New or moved
No change
October 2012
Adopt
Trial
Assess
Hold
March 2012
Adopt
Trial
Assess
Hold
Care about languages
HTML 5
Javascript as a first class language
Scala
Clojure
Domain-specific languages
SASS, SCSS, and LESS
Coffeescript
F#
Functional java
Future of Java
ClojureScript
Logic in stored procedures
Google Dart
Care about languages
SASS, SCSS, LESS, and Stylus
Scala
Clojure
Domain-Specific Languages
Twitter Bootstrap
Require.js
Lua
Gremlin
Google Dart
Dropwizard
Sinatra
AngularJS and Knockout
Jekyll
Scratch, Alice, and Kodu
ClojureScript
RubyMotion
Backbone.js
JavaScript as a platform
HTML5 for offline applications
F#
Component-based frameworks
Logic in stored procedures

Question :
Which programming language would you use now on top of JVM, except Java ?

James Gosling:
Scala



Remember what tools are for. They are for solving problems, not finding problems to solve.

Terry Wall

Typesafe

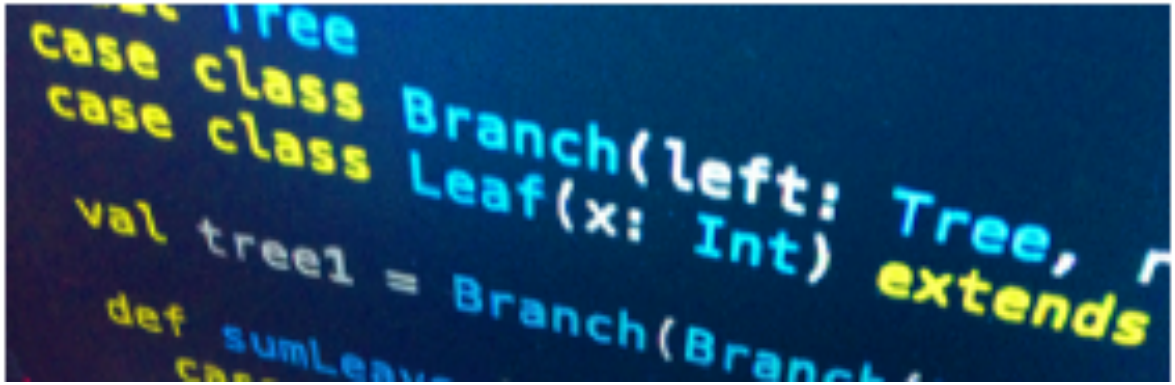
http://www.typesafe.com

Typesafe is strategically positioned to provide innovative solutions with its modern Scala and Akka-based software stack and developer tools for the next wave of multicore, parallel and cloud applications.

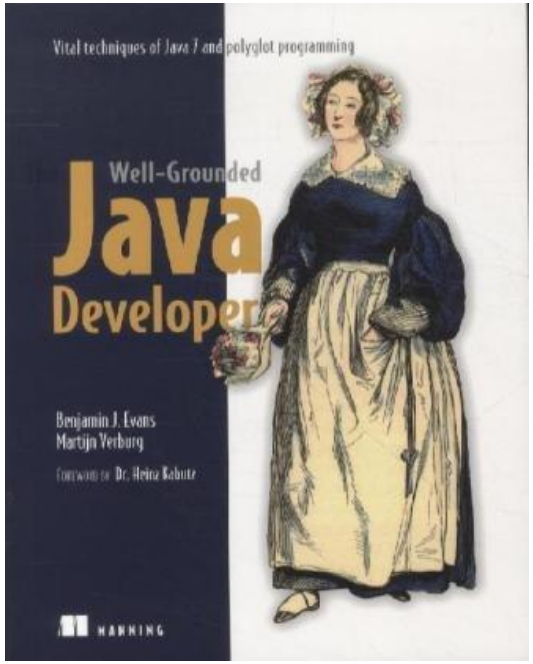
Rod Johnson

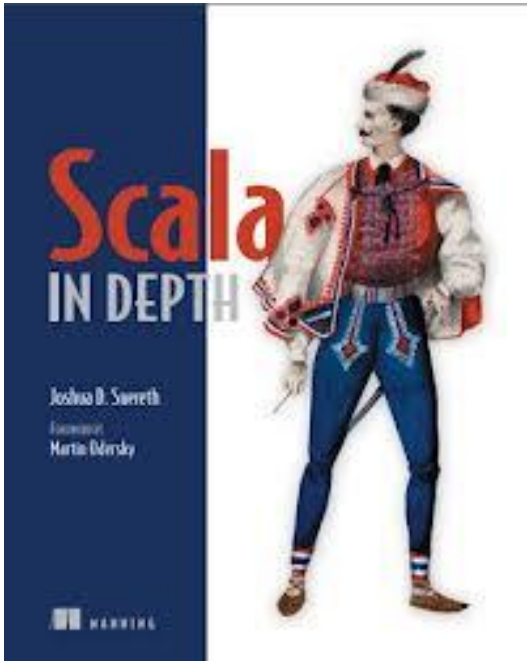
Functional Programming Principles in Scala

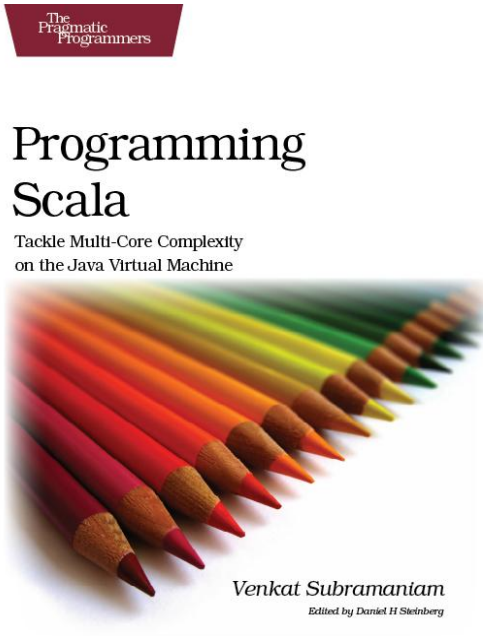
coursera

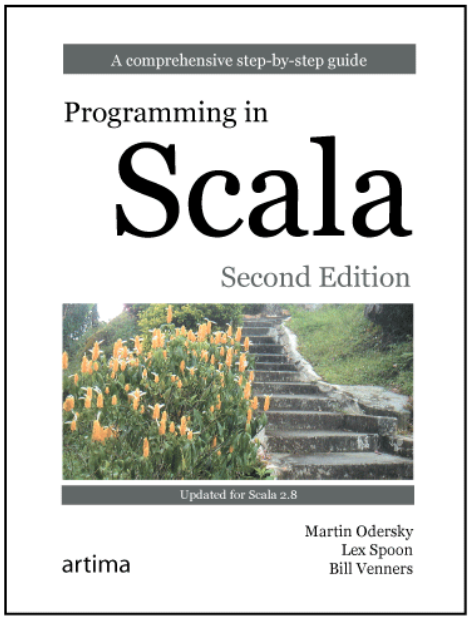


https://www.coursera.org/course/progfun


Vital techniques of Java 7 and polyglot programming
Well-Grounded
Java
Developer
Benjamin J. Evans
Martijn Verburg
MANNING


Scala
IN DEPTH
Joshua D. Suereth
Martin Odersky


The Pragmatic Programmers
Programming
Scala
Tackle Multi-Core Complexity
on the Java Virtual Machine
Venkat Subramaniam
Edited by Daniel H Steinberg


A comprehensive step-by-step guide
Programming in
Scala
Second Edition
Updated for Scala 2.8
Martin Odersky
Lex Spoon
Bill Venners
artima

The Scala Programming Language
http://www.scala-lang.org
http://www.scala-lang.org/docu/files/ScalaTutorial.pdf
http://www.scala-lang.org/docu/files/ScalaByExample.pdf
http://www.scala-lang.org/docu/files/ScalaOverview.pdf
http://www.scala-lang.org/node/960

Scala School
http://twitter.github.io/scala_school
Effective Scala
http://twitter.github.io/effectivescala

Scala - A Scalable Language
http://www.youtube.com/watch?v=zqFryHC018k
Scala Versus Java
http://www.youtube.com/watch?v=PKc5IwHG68k
Why Scala? ...by a hilarious Indian guy
http://www.youtube.com/watch?v=LH75sJAR0hc
Working Hard to Keep It Simple
http://www.youtube.com/watch?v=3jg1AheF4n0

Introduction to Scala for Java Programmers
http://www.slideshare.net/adamrabung/introduction-to-scala-for-java-programmers
Scala Through the Eyes of Java (8)
http://parleys.com/play/5148922b0364bc17fc56c890/chapter1/about
Scala or Java? Exploring myths and facts
http://www.infoq.com/articles/scala-java-myths-facts

Java 8 to the rescue!?
http://2013.flatmap.no/vraalsen.html
Economies of Scala
http://2013.flatmap.no/simen.html

an alternative JVM language

Scala