Scala: How to make best use of functions and objects

Philipp Haller
Lukas Rytz
Martin Odersky
EPFL

ACM Symposium on Applied Computing Tutorial

Scala

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Where it comes from

Scala has established itself as one of the main alternative languages on the JVM.

Prehistory:

1996 – 1997: Pizza
1998 – 2000: GJ, Java generics, javac
("*make Java better*")

Timeline:

2003 – 2006: The Scala "Experiment"
2006 – 2009: An industrial strength programming language
("*make a better Java*")


LinkedIn


twitter


foursquare


Novell.


EDF Trading


SONY
PICTURES


xerox


Office
DEPOT
Taking Care
of Business


SAIC
From Science to Solutions


ebay


naturenews


SIEMENS

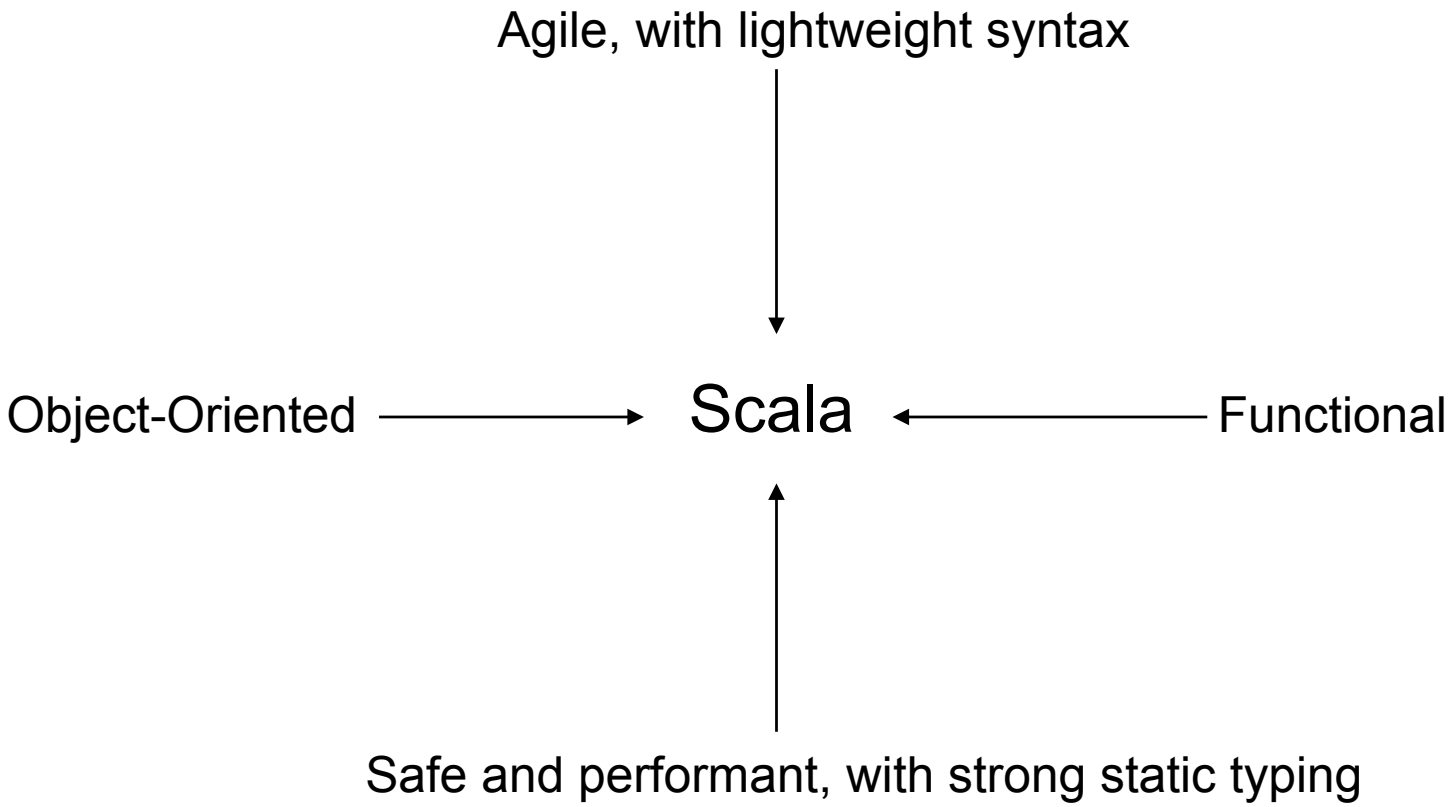

SAP


Day


Scala

Why Scala?

Scala is a Unifier



What others say:

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

- James Gosling, creator of Java

“Scala, it must be stated, is the current heir apparent to the Java throne. No other language on the JVM seems as capable of being a "replacement for Java" as Scala, and the momentum behind Scala is now unquestionable. While Scala is not a dynamic language, it has many of the characteristics of popular dynamic languages, through its rich and flexible type system, its sparse and clean syntax, and its marriage of functional and object paradigms.”

- Charles Nutter, creator of JRuby

“I can honestly say if someone had shown me the Programming in Scala book by Martin Odersky, Lex Spoon & Bill Venners back in 2003 I'd probably have never created Groovy.”

- James Strachan, creator of Groovy.

Let's see an example:

A class ...

... in Java:

```
public class Person {
  public final String name;
  public final int age;
  Person(String name, int age) {
      this.name = name;
      this.age = age;
  }
}
```

... in Scala:

```
class Person(val name: String,
             val age: Int) {}
```

Scala

... and its usage

... in Java:

```
import java.util.ArrayList;
...
Person[] people;
Person[] minors;
Person[] adults;
{  ArrayList<Person> minorsList = new ArrayList<Person>();
   ArrayList<Person> adultsList = new ArrayList<Person>();
   for (int i = 0; i < people.length; i++)
       (people[i].age < 18 ? minorsList : adultsList)
           .add(people[i]);
   minors = minorsList.toArray(people);
   adults = adultsList.toArray(people);
}
```

A function value

An infix method call

... in Scala:

```
val people: Array[Person]
val (minors, adults) = people partition (_.age < 18)
```

A simple pattern match

Scala

But there's more to it

Embedding Domain-Specific Languages

Scala's flexible syntax makes it easy to define

high-level APIs &

embedded DSLs

Examples:

- Scala actors (the core of Twitter's message queues)
- specs, ScalaCheck
- ScalaFX
- ScalaQuery

```
// asynchronous message send
actor ! message

// message receive
receive {
  case msgpat1 => action1
  …
  case msgpatn => actionn
}
```

scalac's plugin architecture makes it easy to typecheck DSLs and to enrich their semantics.

Scala

The Essence of Scala

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.

Why unify FP and OOP?

Both have complementary strengths for composition:

Functional programming:

Makes it easy to build interesting things from simple parts, using

- higher-order functions,
- algebraic types and pattern matching,
- parametric polymorphism.

Object-oriented programming:

Makes it easy to adapt and extend complex systems, using

- subtyping and inheritance,
- dynamic configurations,
- classes as partial abstractions.

Scala

- Scala is an object-oriented and functional language which is completely interoperable with Java. (the .NET version is currently under reconstruction.)
- It removes some of the more arcane constructs of these environments and adds instead:
  (1) a uniform object model,
  (2) pattern matching and higher-order functions,
  (3) novel ways to abstract and compose programs.

Scala is interoperable

Scala programs interoperate seamlessly with Java class libraries:

- Method calls
- Field accesses
- Class inheritance
- Interface implementation

all work as in Java.

Scala programs compile to JVM bytecodes.

Scala's syntax resembles [illegible], but th[illegible] differe[illegible].

```
object Example1 {
  def main(args: Array[String]) {
    val b = new StringBuilder()
    for (i ← 0 until args.length) {
      if (i > 0) b.append(" ")
      b.append(args(i).toUpperCase)
    }
    Console.println(b.toString)
  }
}
```

Array[String] instead of String[]

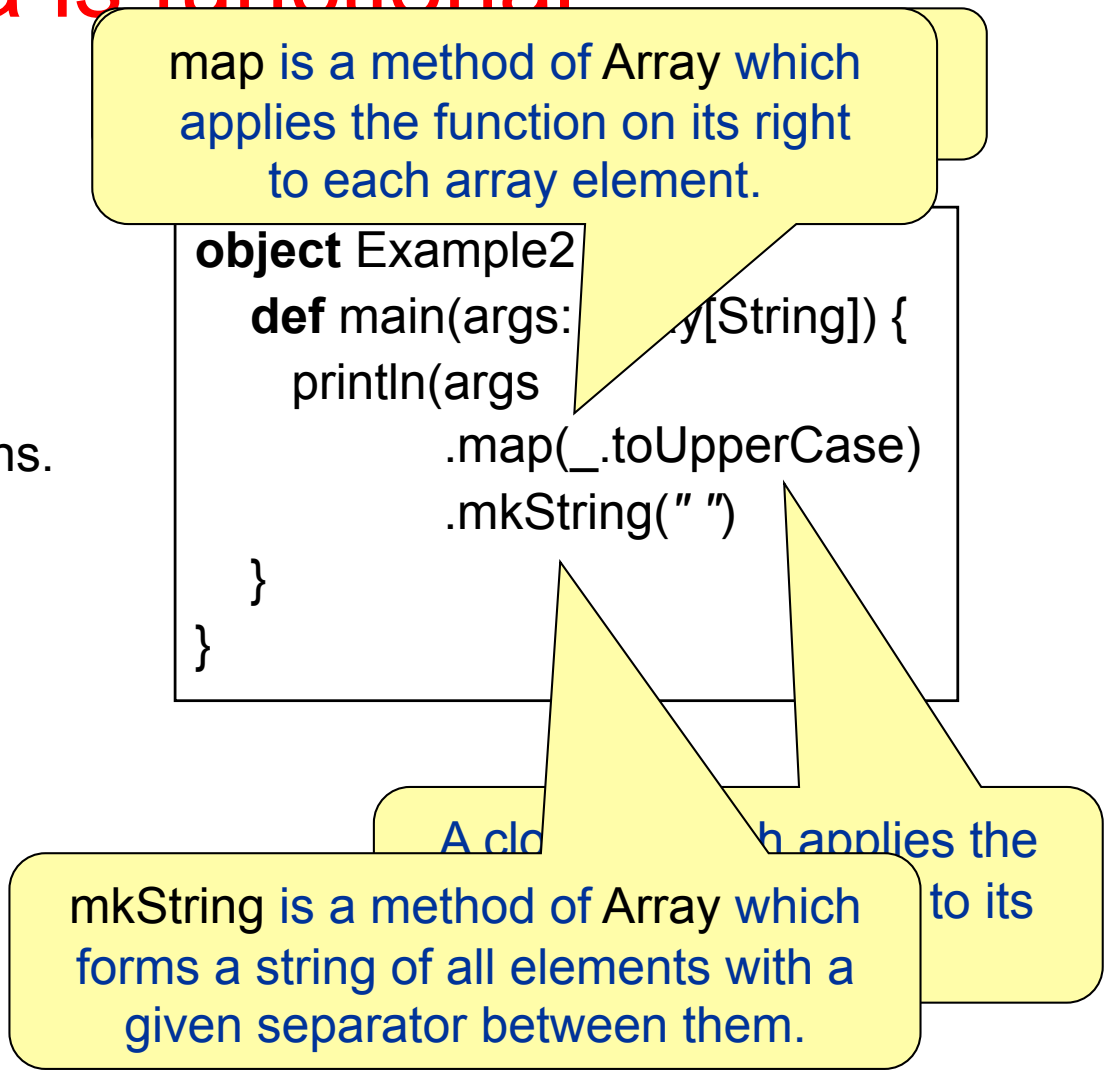
Scala's version of the extended **for** loop (use <- as an alias for ←)

Arrays are indexed args (i) instead of args[i]

Scala is functional

The last program can also be written in a completely different style:

- Treat arrays as instances of general sequence abstractions.
- Use higher-order functions instead of loops.



Scala is concise

Scala's syntax is lightweight and concise.

Contributors:

- semicolon inference,
- type inference,
- lightweight classes,
- extensible API's,
- closures as control abstractions.

```
var capital = Map( "US" → "Washington",
                   "France" → "paris",
                   "Japan" → "tokyo" )

capital += ( "Russia" → "Moskow" )

for ( (country, city) ← capital )
    capital += ( country → city.capitalize )

assert ( capital("Japan") == "Tokyo" )
```

Average reduction in LOC wrt Java: ≥ 2
due to concise syntax and better abstraction capabilities

```
***** Guy Steele:
      Scala led to a 4 times LOC reduction in the Fortress typechecker *****
```

Scala is precise

All code on the previous slide used library abstractions, not special syntax.

Mixin trait SynchronizedMap to make capital map thread-safe

grained control.

Elaborate static type system catches many errors early

Provide a default value: "?"

Specify map implementation: HashMap
Specify map type: String to String

```
import scala.colle          nutable._

val capital =
   new HashMap[String, String]
   with SynchronizedMap[String, String] {
      override def default(key: String) =
         "?"
   }

capital += ( "US" → "Washington",
             "France" → "Paris",
             "Japan" → "Tokyo" )

assert( capital("Russia") == "?" )
```

Big or small?

Every language design faces the tension whether it should be big or small:

- Big is good: expressive, easy to use.
- Small is good: elegant, easy to learn.

Can a language be both big and small?

Scala's approach: concentrate on abstraction and composition capabilities instead of basic language constructs.

Scala adds	Scala removes
+ a pure object system	- static members
+ operator overloading	- special treatment of primitive types
+ closures as control abstractions	- break, continue
+ mixin composition with traits	- special treatment of interfaces
+ abstract type members	- wildcards
+ pattern matching	

Scala is extensible

Guy Steele has formulated a benchmark for measuring language extensibility [Growing a Language, OOPSLA 98]:

> Can you add a type of *complex numbers* to the library and make it work as if it was a native number type?

Similar problems: Adding type BigInt, Decimal, Intervals, Polynomials...

```
scala> import Complex._
import Complex._

scala> val x = 1 + 1 * i
x: Complex = 1.0+1.0*i

scala> val y = x * i
y: Complex = -1.0+1.0*i

scala> val z = y + 1
z: Complex = 0.0+1.0*i
```

Implementing co

Infix operations are method calls:
a + b is the same as a.+(b)

Objects replace static class members

+ is an identifier; can be used as a method name

Class parameters instead of fields+ explicit constructor

```
t def double2complex(x: Double): Comple          omplex(x, 0)

}
class Complex(val re: Double, val im: Double) {
  def + (that: Complex): Complex = new Complex(this.re + that.re, this.im + that.im)
  def -  (that: Complex): Complex = new Complex(this.re - that.re, this.im - that.im)
  def *  (that: Complex): Complex = new Complex(this.re * that.re - this.im * that.im,
                                                this.re * that.im + this.im * that.re)
  def /  (that: Complex): Complex = {
    val denom = that.re * that.re + that.im * that.im
    new Complex((this.re * that.re + this.im * that.im) / denom,
                (this.im * that.re - this.re * that.im) / denom)
  }
  override def toString = re+(if (im < 0) "-"+(-im) else "+"+im)+"*I"
  ...
}
```

Implicit conversions for mixed arithmetic

Implicits are Poor Man's Type Classes

```
/** A "type class" */
class Ord[T] { def < (x: T, y: T): Boolean }

/** An "instance definition" */
implicit object intOrd extends Ord[Int] {
  def < (x: Int, y: Int) = x < y
}

/** Another instance definition */
implicit def listOrd[T](implicit tOrd: Ord[T]) = new Ord {
  def < (xs: List[T], ys: List[T]) = (xs, ys) match {
    case (_, Nil) => false
    case (Nil, _) => true
    case (x :: xs, y :: ts) => x < y || x == y && xs < ys
  }
}
```

The Bottom Line

When going from Java to Scala, expect at least a factor of 2 reduction in LOC.

But does it matter?
Doesn't Eclipse write these extra lines for me?

This does matter. Eye-tracking experiments* show that for program comprehension, average time spent per word of source code is constant.

So, roughly, half the code means half the time necessary to understand it.

*G. Dubochet. Computer Code as a Medium for Human Communication: Are Programming Languages Improving? In 21st Annual Psychology of Programming Interest Group Conference, pages 174-187, Limerick, Ireland, 2009.

Part 2: The Scala Design

The Scala design

Scala strives for the tightest possible integration of OOP and FP in a statically typed language.

This continues to have unexpected consequences.

Scala unifies

- algebraic data types with class hierarchies,
- functions with objects

This gives a nice & rather efficient formulation of Erlang style actors

ADTs are class hierarchies

Many functional languages have algebraic data types and pattern matching.

⇒

Concise and canonical manipulation of data structures.

Object-oriented programmers object:

- *ADTs are not extensible,*
- *ADTs violate the purity of the OO data model,*
- *Pattern matching breaks encapsulation,*
- *and it violates representation independence!*

Pattern matching in Scala

The **case** modifier of an object or class means you can pattern match on it

Here's a a set of definitions describing binary trees:

```
a    class Tree[T]

case object Empty extends Tree[Nothing]

case class Binary[T](elem: T, left: Tree[T], right: Tree[T])
        extends Tree[T]
```

And here's an inorder traversal of binary trees:

```
def inOrder [T] ( t: Tree[T] ): List[T] = t match {
 case Empty           => List()
 case Binary(e, l, r) => inOrder(l) ::: List(e) ::: inOrder(r)
}
```

This design keeps

- purity: all cases are classes or objects.
- extensibility: you can define more cases elsewhere.
- encapsulation: only parameters of case classes are revealed.
- representation independence using extractors [ECOOP 07].

Extractors

... are objects with unapply methods.
... similar to *active patterns* in F#

unapply is called implicitly for pattern matching

```
object Twice {
  def apply(x: Int) = x*2
  def unapply(z: Int): Option[Int] = if (z%2==0) Some(z/2) else None
}
val x = Twice(21)
x match {
  case Twice(y) => println(x+" is two times "+y)
  case _ => println("x is odd") }
}
```

Functions are objects

Scala is a functional language, in the sense that every function is a value.

If functions are values, and values are objects, it follows that functions themselves are objects.

The function type S => T is equivalent to scala.Function1[S, T] where Function1 is defined as follows :

```
trait Function1[-S, +T] {
    def apply(x: S): T
}
```

So functions are interpreted as objects with apply methods.

For example, the *anonymous successor* function
(x: Int) => x + 1 is expanded to

```
new Function1[Int, Int] {
    def apply(x: Int): Int =
        x + 1
}
```

Why should I care?

- Since (=>) is a class, it can be subclassed.
- So one can specialize the concept of a function.
- An obvious use is for arrays, which are mutable functions over integer ranges.
- Another bit of syntactic sugaring lets one write:

  a(i) = a(i) + 2 for

  a.update(i, a.apply(i) + 2)

```
class Array [T] ( length: Int )
   extends (Int => T)  {

   def length: Int = ...

   def apply(i: Int): A = ...

   def update(i: Int, x: A): unit = ...

   def elements: Iterator[A] = ...

   def exists(p: A => Boolean):Boolean
      = ...
}
```

Partial functions

- Another useful abstraction are partial functions.
- These are functions that are defined only in some part of their domain.
- What's more, one can inquire with the isDefinedAt method whether a partial function is defined for a given value.

```
trait PartialFunction[-A, +B]
extends (A => B) {
  def isDefinedAt(x: A): Boolean
}
```

- Scala treats blocks of pattern matching cases as instances of partial functions.
- This lets one write control structures that are not easily expressible otherwise.

Example: Erlang-style actors

- Two principal constructs (adopted from Erlang):
- Send (!) is asynchronous; messages are buffered in an actor's mailbox.
- receive picks the first message in the mailbox which matches any of the patterns $mspat_i$.
- If no pattern matches, the actor suspends.

```
// asynchronous message send
actor ! message

// message receive
receive {
    case msgpat1 => action1
    ...
    case msgpatn => actionn
}
```

A partial function of type
PartialFunction[*MessageType*, *ActionType*]

A simple actor

```
case class Data(b: Array[Byte])
case class GetSum(receiver: Actor)
val checkSumCalculator =
    actor {
        var sum = 0
        loop {
            receive {
                case Data(bytes) => sum += hash(bytes)
                case GetSum(receiver) => receiver ! sum
            }
        }
    }
```

Implementing receive

- Using partial functions, it is straightforward to implement receive:
- Here,

  self designates the currently executing actor,

  mailBox is its queue of pending messages, and

  extractFirst extracts first queue element matching given predicate.

```
def receive [A]
    (f: PartialFunction[Message, A]): A = {
    self.mailBox.extractFirst(f.isDefinedAt)
    match {
        case Some(msg) =>
            f(msg)
        case None =>
            self.wait(messageSent)
    }
}
```

Library or language?

- A possible objection to Scala's library-based approach is:

  Why define actors in a library when they exist already in purer, more optimized form in Erlang?

- First reason: interoperability
- Another reason: libraries are much easier to *extend* and *adapt* than languages.

Experience:

Initial versions of actors used one thread per actor

⇒ lack of speed and scalability

Later versions added a non-returning `receive' called react which makes actors *event-based.*

This gave great improvements in scalability.

New variants using delimited continuations are being explored (this ICFP).

Scala cheat sheet (1): Definitions

Scala method definitions:

```
def fun(x: Int): Int = {
    result
}

def fun = result
```

Scala variable definitions:

```
var x: int = expression
val x: String = expression
```

Java method definition:

```
int fun(int x) {
    return result
}
```

(no parameterless methods)

Java variable definitions:

```
int x = expression
final String x = expression
```

Scala cheat sheet (2): Expressions

Scala method calls:

```
    obj.meth(arg)
or: obj meth arg
```

Scala choice expressions:

```
if (cond) expr1 else expr2
```

```
expr match {
  case pat1 => expr1
  ....
  case patn => exprn
}
```

Java method call:

```
obj.meth(arg)
```
(no operator overloading)

Java choice expressions, stats:

```
cond ? expr1 : expr2   // expression
if (cond) return expr1; // statement
else return expr2;
```

```
switch (expr) {
  case pat1 : return expr1;
  ...
  case patn : return exprn ;
} // statement only
```

Scala cheat sheet (3): Objects and Classes

Scala Class and Object

```
class Sample(x: Int) {
  def instMeth(y: Int) = x + y
}

object Sample {
  def staticMeth(x: Int, y: Int) = x * y
}
```

Java Class with static

```
class Sample {
  final int x;
  Sample(int x) { this.x = x }

  int instMeth(int y) {
    return x + y;
  }

  static int staticMeth(int x, int y) {
    return x * y;
  }
}
```

Scala

Scala cheat sheet (4): Traits

Scala Trait

```
trait T {
  def abstractMeth(x: String): String

  def concreteMeth(x: String) =
    x+field

  var field = "!"
}
```

Scala mixin composition:

```
class C extends Super with T
```

Java Interface

```
interface T {
  String abstractMeth(String x)

  (no concrete methods)


  (no fields)
}
```

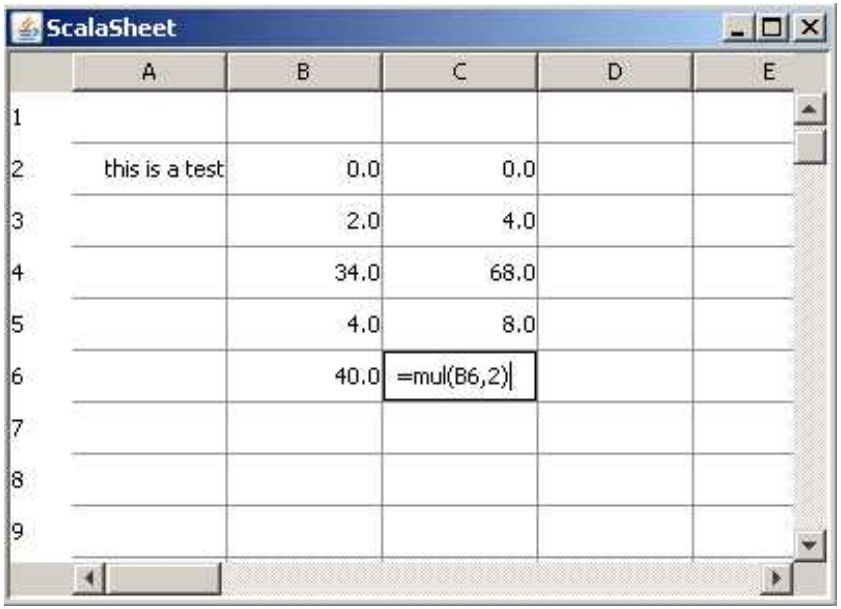
Java extension + implementation:

```
class C extends Super implements T
```

Part 3: Programming in Scala

Scala in serious use

- You'll see now how Scala's constructs play together in a realistic application.
- Task: Write a spreadsheet
- Start from scratch, don't use any parts which are not in the standard libraries
- You'll see that this can be done in under 200 lines of code.
- Nevertheless it demonstrates many aspects of scalability
- For comparison: Java demo: 850 LOC, MS Office 30Million LOC



Step 1: The main function

```
package scells
import swing._

object Main extends SimpleSwingApplication {
  def top = new MainFrame {
    title = "ScalaSheet"
    contents += new SpreadSheet(100, 26)
  }
}
```

- Advantage of objects over statics: objects can inherit.
- Hence, can hide low-level fiddling necessary to set up a swing application.

Property syntax; expands to r
rowHeight_=(25)

This calls in turn jtable.setRowHeight(25)

Step [illegible] SpreadSh[illegible]ass - view

```
class SpreadSheet[illegible]eight: Int[illegible] width: Int) extends ScrollPane {
  val cellModel = new Model(he[illegible]th)
  import cellModel.{cells, va[illegible]ged}
  val table = new [illegible]able(height, width) {
    rowHeight = 25
    autoResizeMode = Table.AutoResizeMode.Off
    showGrid = true
    gridColor = Color(150, 150, 150)

    def userData(row: Int, column: Int): String = {
      val v = this(row, column); if (v == null) "" else v.toString
    }

    override def render(isSelected: Boolean, hasFocus: Boolean, row: Int, column: Int) =
      if (hasFocus) new TextField(userData(row, column))
      else new Label(cells(row)(column).toString) { halign = Orientation.right }

    reactions += {
      case event.TableChanged(table, firstRow, lastRow, column) =>
        for (row <- firstRow to lastRow)
          cells(row)(column).formula =
            FormulaParsers.parse(userData(row, column))
      case ValueChanged(cell) =>
        markUpdated(cell.row, cell.column)
    }

    for (row <- cells; cell <- row) listenTo(cell)
  }

  val rowHeader = new ComponentList(0 until height map (_.toString)) {
    fixedCellWidth = 30
    fixedCellHeight = table.rowHeight
  }
  viewportView = table; rowHeaderView = rowHeader
}
```

Step 3: The SpreadSheet class - controller

```
class SpreadSheet(val height: Int, val width: Int) extends ScrollPane {
  val cellModel = new Model(height, width)
  import cellModel.{cells, valueChanged}

  val table = new Table(height, width) {
      rowHeight = 25
      autoResizeMode = Table.AutoResizeMode.Off
      showGrid = true
      gridColor = Color(150, 150, 150)

      def userData(row: Int, column: Int): String = {
        val v = this(row, column)
        if (v == null) "" else v.toString
      }

      override def render(isSelected: Boolean, hasFocus
        if (hasFocus) new TextField(userData(row, colum
        else new Label(cells(row)(column).toString) { halign = Right

    reactions += {
      case event.TableChanged(table, firstRow, lastRow,
        for (row <- firstRow to lastRow)
          cells(row)(column).formula = FormulaParsers.parse(userData(row, column))
      case ValueChanged(cell) =>
        markUpdated(cell.row, cell.column)
    }

    for (row <- cells; cell <- row) listenTo(cell)
  }


   val rowHeader = new ComponentList((0 until height) map (_.toString)) {
    fixedCellWidth = 30
    fixedCellHeight = table.rowHeight
  }
  viewportView = table; owHeaderView = rowHeader
}
```

`reactions` property defines component behavior with closures.

Events are objects, can pattern match on them.

Spreadsheet formulas

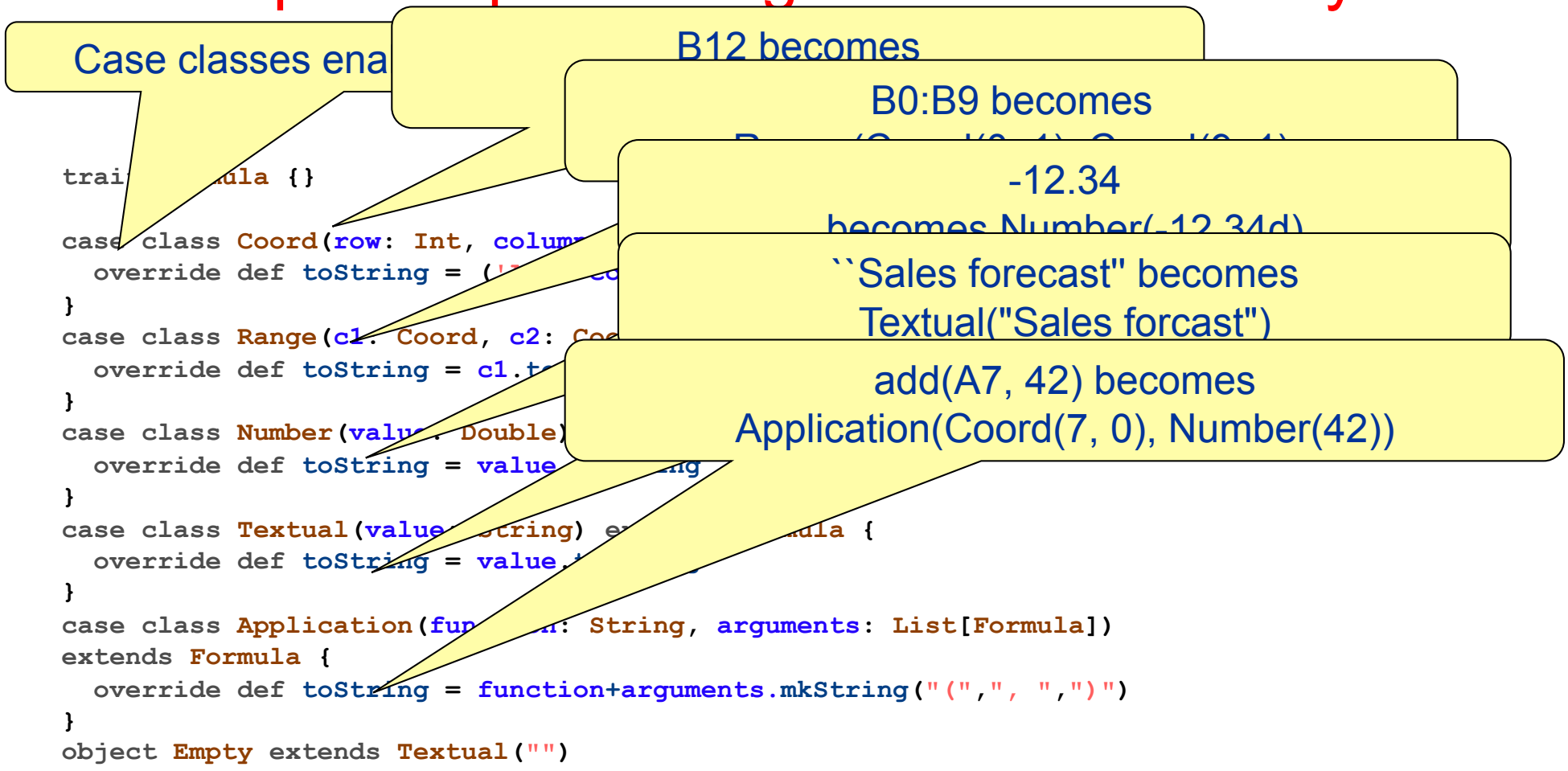
- We consider:

`-12.34`	Number
`text`	Text label
`=expr`	Formulas, consisting of
`B12`	Cell
`B12:C18`	Range of cells
`add(A7,A4)`	Binary operation
`sum(A12:A14,A16)`	Vararg operation
	(no infix operations such as `X+Y`)

- Formula expressions can nest, as in:

```
=sum(mul(A4, 2.0), B7:B15))
```

Step 4: Representing formulas internally




```
trai    ula {}

case class Coord(row: Int, colum
  override def toString = ('
}
case class Range(c1: Coord, c2: Coo
  override def toString = c1.to
}
case class Number(valu: Double)
  override def toString = value
}
case class Textual(value: String) ex    ula {
  override def toString = value.t
}
case class Application(fun    n: String, arguments: List[Formula])
extends Formula {
  override def toString = function+arguments.mkString("(",", ",")")
}
object Empty extends Textual("")
```

A grammar for formulas

```
number      =   -?\d+(\.\d*)
ident       =   [A-Za-z_]\w*
cell        =   [A-Za-Z]\d+
range       =   cell : cell
application =   ident
                ( expr (, expr)* )
expr        =   number
            |   cell
            |   range
            |   application
formula     =   = expr
textual     =   [^=].*
```

A grammar for formulas and their parsers

number	=	-?\d+(\.\d*)	`"""-?\d+(\.\d*)?""".r`
ident	=	[A-Za-z_]\w*	`"""[a-zA-Z_]\w*""".r`
cell	=	[A-Za-Z]\d+	`""" [A-Za-z]\d\d*""".r`
range	=	cell : cell	`cell~":"~cell`
application	=	ident (expr (, expr)*)	`ident~` `"("~repsep(expr, ",")~")"`
expr	=	number | cell | range | application	`number | cell | range | application`
formula	=	= expr	`"="~expr`
textual	=	[^=].*	`"""[^=].*""".r`

Step 5: Parsing formulas

```
object FormulaParsers
extends RegexParsers {

  def ident: Parser[String] =
    """[a-zA-Z_]\w*""".r
  def decimal: Parser[String] =
    """-?\d+(\.\d*)?""".r

  def cell: Parser[Coord] =
    """[A-Za-z]\d+""".r ^^ { s =>
      val column = s.charAt(0) - 'A'
      val row = s.substring(1).toInt
      Coord(row, column)
    }

  def range: Parser[Range] =
    cell~":"~cell ^^ {
      case c1~":"~c2 => Range(c1, c2)
    }

  def number: Parser[Number] =
    decimal ^^ (s => Number(s.toDouble))
```

```
  def application: Parser[Application] =
      ident~"("~repsep(expr, ",")~")" ^^ {
        case f~"("~ps~")" =>
       Application(f, ps)
      }

    def expr: Parser[Formula] =
      application | range | cell | number

    def textual: Parser[Textual] =
      """[^=].*""".r ^^ Textual

    def formula: Parser[Formula] =
      number | textual | "=" ~> expr

    def parse(input: String): Formula =
      parseAll(formula, input) match {
        case Success(e, _) => e
        case f: NoSuccess =>
                  Textual("["+f.msg+"]")
      }

}
```

This makes use of an *internal DSL*, much like the external Lex and Yacc.

Step 6

Evaluate by pattern matching on the kind of formula

```
trait Evaluator { this: Model =>
  val operations = new collection.mutable.HashMap[Double] => Double]
  def evaluate(e: Formula): Double = e
    case Number(v)                    =>
    case Textual(_)      => 0
    case Coord(row, column)           =>
    case Application(function, arguments) =>
      val argvals = arguments flatMap evalList
      operations(function)(argvals)
  }
  private def evalList(e: Formula): List[Double]          ch {
    case Range(_, _) => references(e) map (_.valu
    case _ => List(evaluate(e))
  }
  def references(e: Formula): List[Cell] = e match
    case Coord(row, column) => List(cel
    case Range(Coord(r1, c1), Coord(r2,
      for (row <- (r1 to r2).toList; column <- c1 to c2)
      yield cells(row)(column)
    case Application(function, arguments) => arguments flatMap references
    case => List()
  }
```

Scala's *Self-type* feature lets us assume the type of **this** in Evaluator is Model

But how does Evaluator know about **cells**?

Step 7: The spreadsheet Model class

```
class Model(val height: Int, val width: int) extends Evaluator with Arithmetic {

  class Cell(row: Int, column: Int) extends Publisher {
    private var v: Double = 0
    def value: Double = v
    def value_=(w: Double) {
      if (!(v == w || v.isNaN && w.isNaN)) {
        v = w
        publish(ValueChanged(this))
      }
    }
    private var e: Formula = Empty
    def formula: Formula = e
    def formula_=(e: Formula) {
      for (c <- references(formula)) deafTo(c)
      this.e = e
      for (c <- references(formula)) listenTo(c)
      value = evaluate(e)
    }
    reactions += {
      case ValueChanged(_) => value = evaluate(formula)
    }
  }

  case class ValueChanged(cell: Cell) extends event.Event

  val cells = Array.fromFunction(new Cell(_, _))(width, height)
}
```

Property definitions make interesting things happen when variables are set

Lessons learned

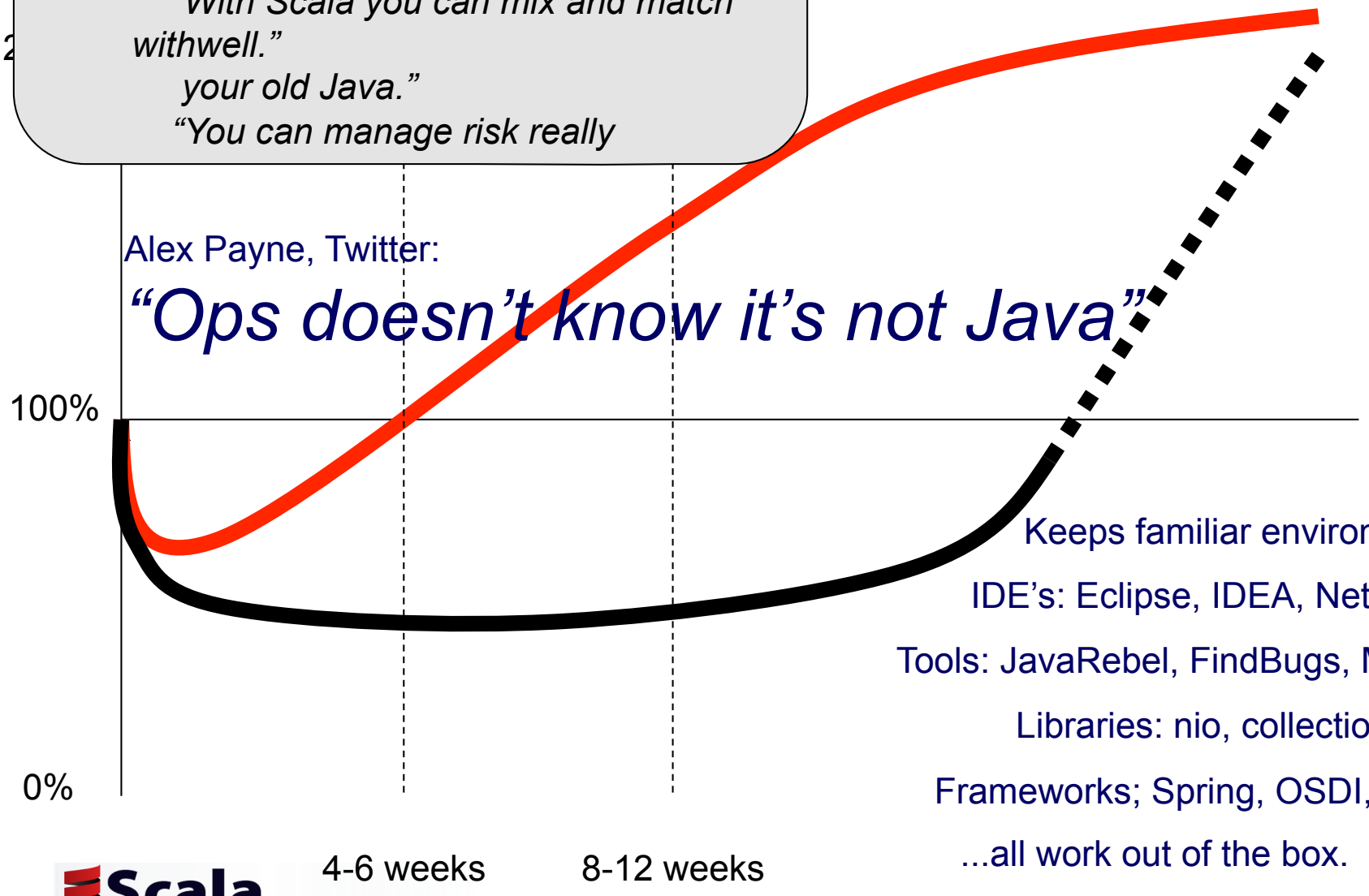
- DSL's can help keep software short and clear: Parser combinators, swing components and reactions.
- Internal DSLs have advantages over external ones.
- Mixin composition + self types let you write fully re-entrant complex systems without any statics.
- Application complexity can be reduced by the right language constructs.
- To ensure you always have the right constructs, you need a language that's extensible and scalable.

But how long will it take me to switch?

Curves

Alex McGuire, EDF, who replaced majority of 300K lines Java with Scala:

"Picking up Scala was really easy."
"Begin by writing Scala in Java style."
"With Scala you can mix and match withwell."
your old Java."
"You can manage risk really



Alex Payne, Twitter:

"Ops doesn't know it's not Java"

Keeps familiar environment:

IDE's: Eclipse, IDEA, Netbeans, ...

Tools: JavaRebel, FindBugs, Maven, ...

Libraries: nio, collections, FJ, ...

Frameworks; Spring, OSDI, J2EE, ...

...all work out of the box.

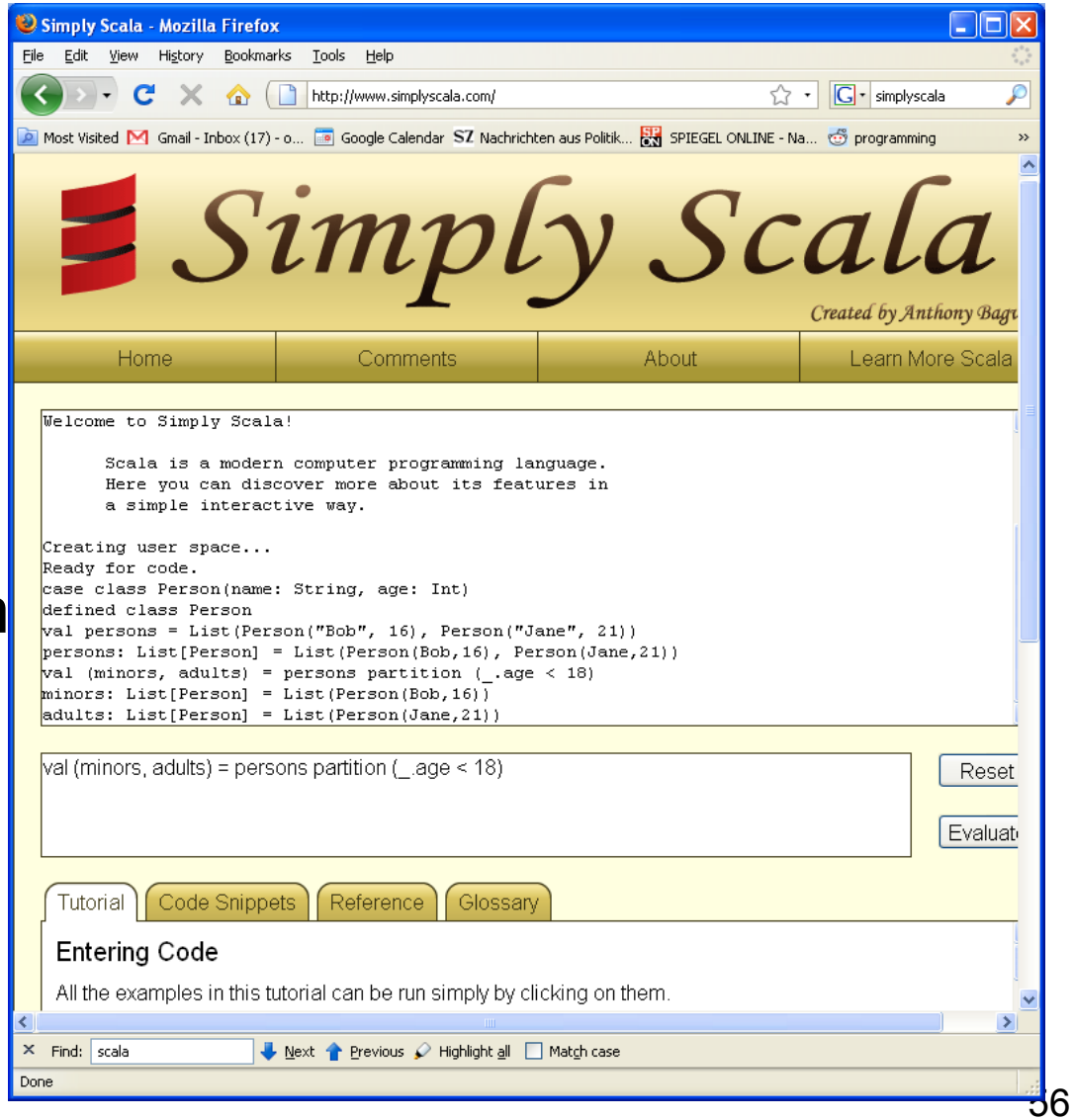
Scala

How to get started

100s of resources on the web.

Here are three great entry points:

- Simply Scala
- Scalazine @ artima.com
- Scala for Java refugees



How to find out more

Scala site: www.scala-lang.org

Six books last year

Soon to come

New release Scala 2.8, with

- named and default parameters,
- @specialized annotations for high performance numerical computations,
- improved IDE plugin support,
- and much more.

New version on .NET with Visual Studio integration

Long term focus: Concurrency & Parallelism

Our goal: establish Scala as the premier language for multicore programming.

Actors gave us a head start.

Actors as a library worked well because of Scala's flexible syntax and strong typing.

The same mechanisms can also be brought to bear in the development of other concurrency abstractions, such as:

- parallel collections,
- software transactional memory,
- stream processing.

Thank You