Programming – III

Object-oriented Programming SCS2104

Kasun De Zoysa

Why a new language?

- Goal was to create a language with better support for component software
- Two hypotheses:
  - Programming language for component software should be scalable
    - The same concepts describe small and large parts
    - Rather than adding lots of primitives, focus is on *abstraction*, *composition*, and *decomposition*
  - Language that unifies OOP and functional programming can provide scalable support for components
- Adoption is key for testing this hypothesis
  - Scala interoperates with Java and .NET

Java

- What's wrong with Java?
  - Not designed for highly concurrent programs
    - The original Thread model was just *wrong* (it's been fixed)
    - Java 5+ helps by including `java.util.concurrent`
  - Verbose
    - Too much of `Thing thing = new Thing();`
    - Too much "boilerplate," for example, getters and setters

- What's right with Java?
  - Very popular
  - Object oriented (mostly), which is important for large projects
  - Strong typing (more on this later)
  - The fine large library of classes
  - **The JVM!** Platform independent, highly optimized

Scala is like Java, except when it isn't

- Java is a *good* language, and Scala is a lot like it
- For each difference, there is a *reason*--none of the changes are "just to be different"
- Scala and Java are (almost) completely interoperable
  - Call Java from Scala? No problem!
  - Call Scala from Java? Some restrictions, but mostly OK.
  - Scala compiles to .class files (a *lot* of them!), and can be run with either the scala command or the java command
- To understand Scala, it helps to understand the reasons for the changes, and what it is Scala is trying to accomplish

Where it comes from

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

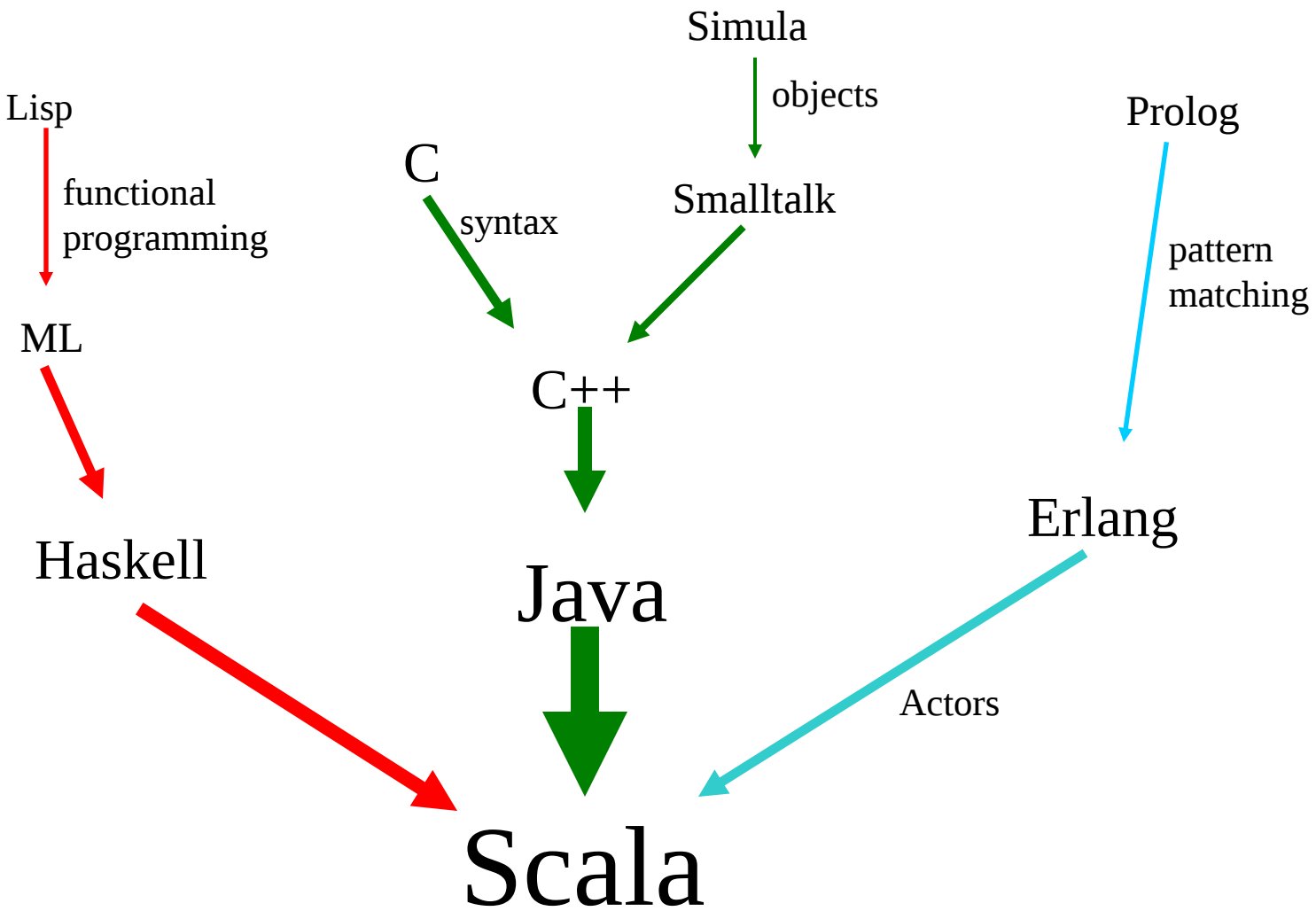
Prehistory:

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

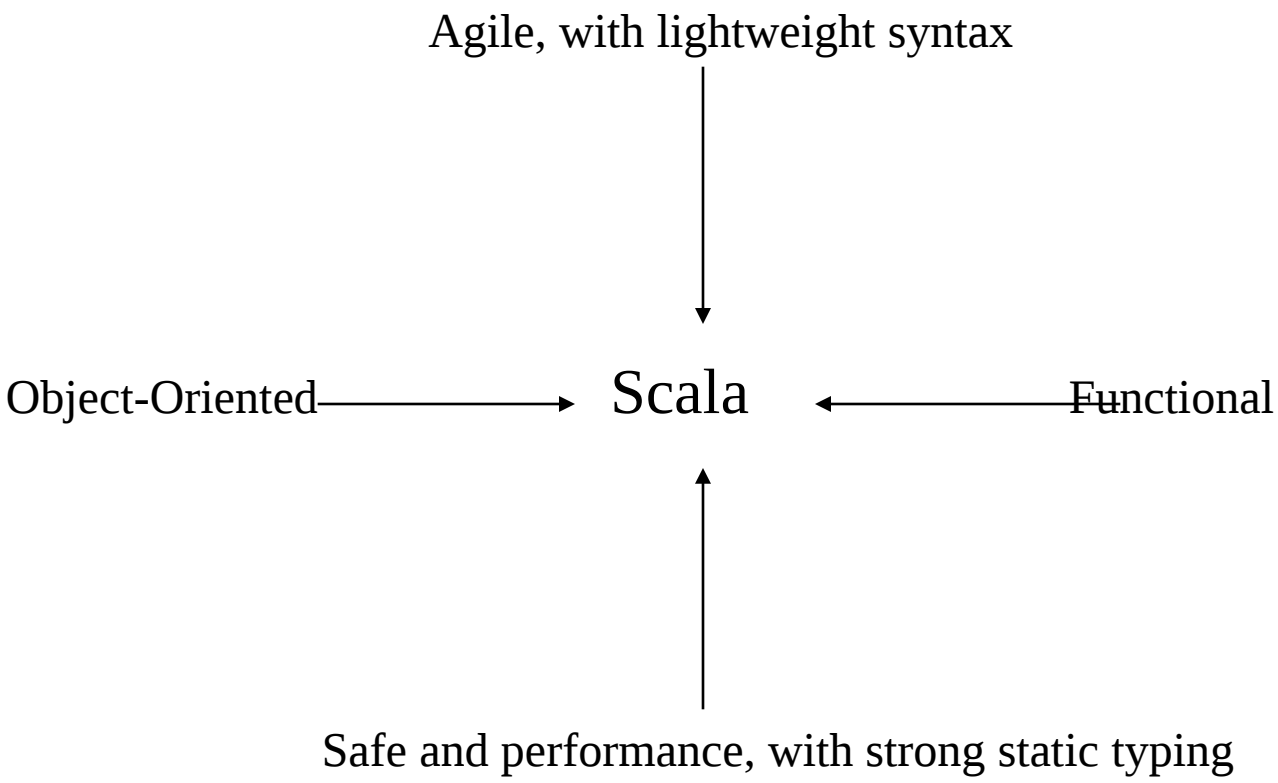
Timeline:

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

Genealogy


Simula
objects
Lisp
functional
programming
C
syntax
Smalltalk
Prolog
pattern
matching
ML
C++
Haskell
Java
Erlang
Actors
Scala

Scala is a Unifier




LinkedIn


twitter


Novell.


foursquare


EDF Trading


xerox


SONY
PICTURES


Office
DEPOT
Taking Care
of Business


SAIC
From Science to Solutions


ebay


naturenews


SIEMENS


SAP


Day

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) {}
```

... 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

Features of Scala

- Scala is both functional and object-oriented
  - every value is an object
  - every function is a value--including methods
- Scala is statically typed
  - includes a local type inference system:

in Java 1.5:

```
Pair p = new Pair<Integer, String>(1, "Scala");
```

in Scala:

```
val p = new MyPair(1, "scala");
```

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.

Functional languages

- The best-known functional languages are ML, OCaml, and Haskell
- Functional languages are regarded as:
  - "Ivory tower languages," used only by academics (mostly but not entirely true)
  - Difficult to learn (mostly true)
  - The solution to all concurrent programming problems everywhere (exaggerated, but not entirely wrong)
- Scala is an "impure" functional language--you can program functionally, but it isn't forced upon you

Scala as a functional language

- The hope--*my* hope, anyway--is that Scala will let people "sneak up" on functional programming (FP), and gradually learn to use it
  - This is how C++ introduced Object-Oriented programming
- Even a little bit of functional programming makes some things a lot easier
- Meanwhile, Scala has plenty of other attractions

- FP really is a different way of thinking about programming, and not easy to master...
- ...but...
- Most people that master it, never want to go back

More features

- Supports lightweight syntax for anonymous functions, higher-order functions, nested functions, currying
- ML-style pattern matching
- Integration with XML
  - can write XML directly in Scala program
  - can convert XML DTD into Scala class definitions
- Support for regular expression patterns

Consistency is good

- In Java, every value is an object--unless it's a primitive
  - Numbers and booleans are primitives for reasons of efficiency, so we have to treat them differently (you can't "talk" to a primitive)
- In Scala, all values are objects. Period.
  - The compiler turns them into primitives, so no efficiency is lost (behind the scenes, there are objects like RichInt)

- Java has *operators* (+, <, ...) and *methods*, with different syntax
- In Scala, operators are just methods, and in many cases you can use either syntax

Type safety is good

- Java is statically typed--a variable has a type, and can hold *only* values of that type
  - You must specify the type of every variable
  - Type errors are caught by the compiler, not at runtime--this is a big win
  - However, it leads to a lot of typing (pun intended)
- Languages like Ruby and Python don't make you declare types
  - Easier (and more fun) to write programs
  - Less fun to debug, especially if you have even slightly complicated types
- Scala is *also* statically typed, but it uses type inferencing--that is, it figures out the types, so you don't have to
  - The good news: Less typing, more fun, type errors caught by the compiler
  - The bad news: More kinds of error messages to get familiar with

null in Scala

- In Java, any method that is *supposed* to return an object *could* return null
  - Here are your options:
    - Always check for null
    - Always put your method calls inside a try...catch
    - Make sure the method can't possibly return null
    - Ignore the problem and depend on luck
      - http://www.youtube.com/watch?v=u0-oinyjsk0

- Yes, Scala has null--but only so that it can talk to Java
- In Scala, if a method *could* return "nothing," write it to return an Option object, which is either Some(***theObject***) or None
  - This forces you to use a match statement--but only when one is really needed!

Referential transparency

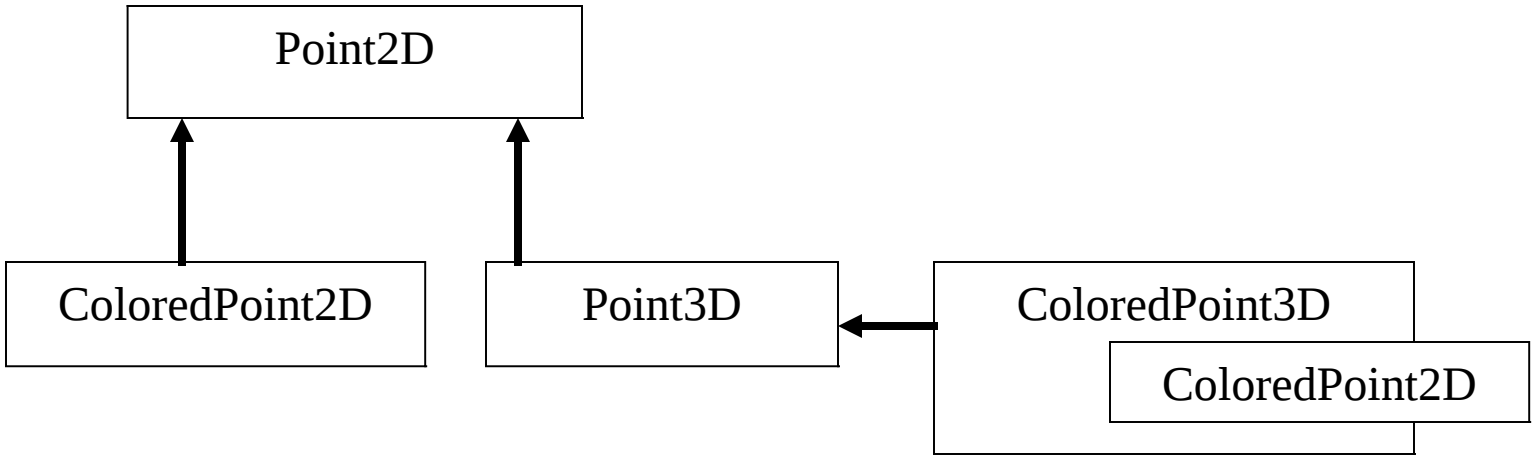
- In Scala, variables are really functions
  - Huh?
- In Java, if `age` is a public field of `Person`, you can say:
  `david.age = david.age + 1;`
  but if `age` is accessed via methods, you would say:
  `david.setAge(david.getAge() + 1);`
- In Scala, if `age` is a public field of `Person`, you can say:
  `david.age = david.age + 1;`
  but if `Person` defines methods `age` and `age_=`, you would say:
  `david.age = david.age + 1;`
- In other words, if you want to access a piece of data in Scala, you don't have to know whether it is computed by a method or held in a simple variable
  - This is the principle of uniform access
  - Scala *won't let you* use parentheses when you call a function with no parameters

Mixin class composition

- Basic inheritance model is single inheritance
- But mixin classes allow more flexibility

```
class Point2D(xc: Int, yc: Int) {
  val x = xc;
  val y = yc;
  // methods for manipulating Point2Ds
}
class ColoredPoint2D(u: Int, v: Int, c: String)
              extends Point2D(u, v) {
  var color = c;
  def setColor(newCol: String): Unit = color = newCol;
}
```

Mixin class composition example




```
class Point3D(xc: Int, yc: Int, zc: Int)
              extends Point2D(xc, yc) {
   val z = zc;
   // code for manipulating Point3Ds
}

class ColoredPoint3D(xc: Int, yc: Int, zc: Int, col: String)
      extends Point3D(xc, yc, zc)
      with ColoredPoint2D(xc, yc, col);
```

Concurrency

- “Concurrency is the new black.”
- Broadly speaking, concurrency can be either:
  - **Fine-grained:** Frequent interactions between threads working closely together (extremely challenging to get right)
  - **Coarse-grained:** Infrequent interactions between largely independent sequential processes (much easier to get right)
- Java 5 and 6 provide reasonable support for traditional fine-grained concurrency
- Scala has total access to the Java API
  - Hence, it can do anything Java can do
  - And it can do much more (see next slide)
- Scala also has Actors for coarse-grained concurrency

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 =>
    ...on_n
```

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.isDefined
At)
    match {
        case Some(msg) =>
            f(msg)
        case None =>
            self.wait(messageSent)
    }
}
```

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 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
```

Traits

- Similar to interfaces in Java
- They may have implementations of methods
- But can’t contain state
- Can be multiply inherited from

But how long will it take me to switch?

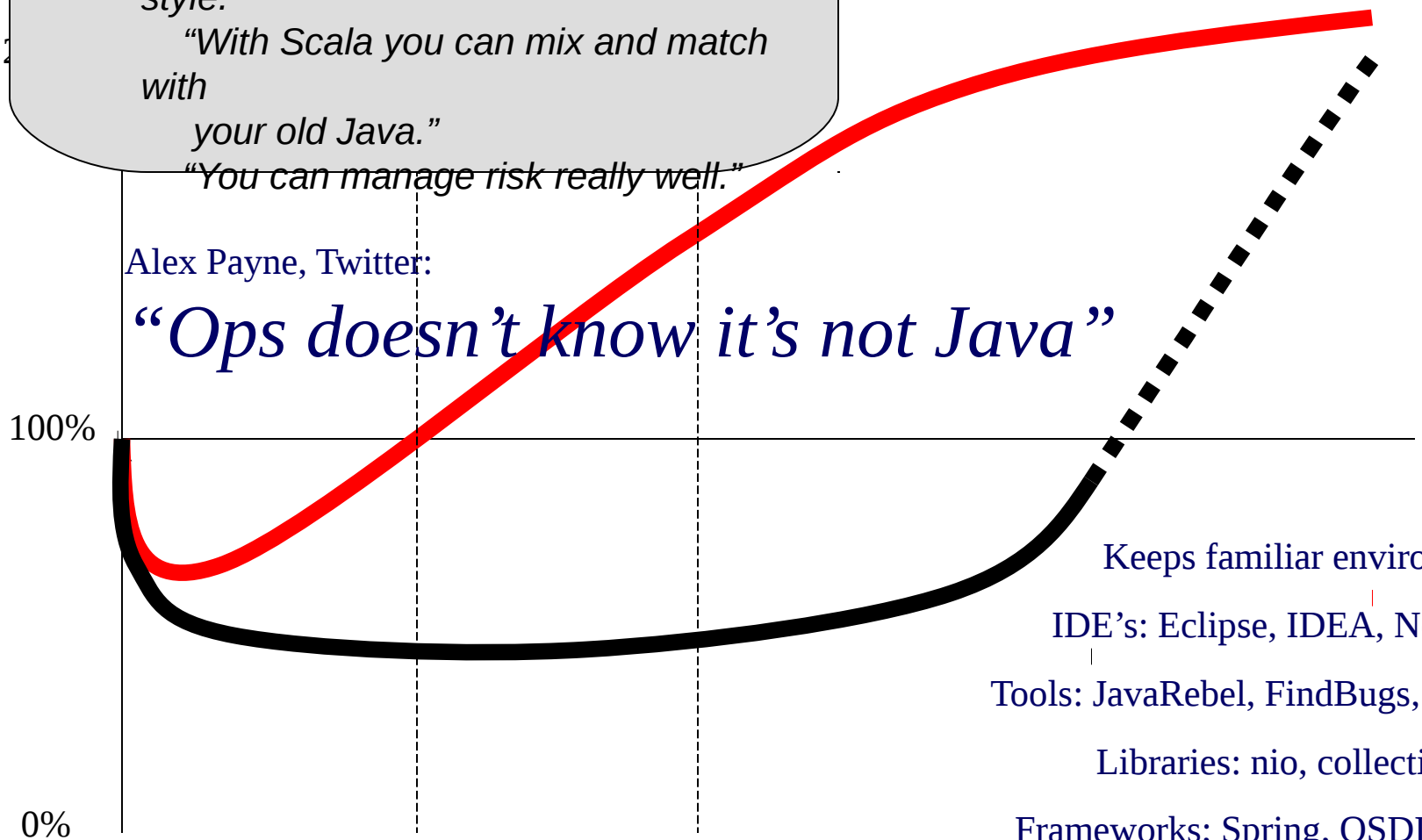
Alex McGuire, EDF, who replaced majority of 300K lines Java with Scala:

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

Curves

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, ...

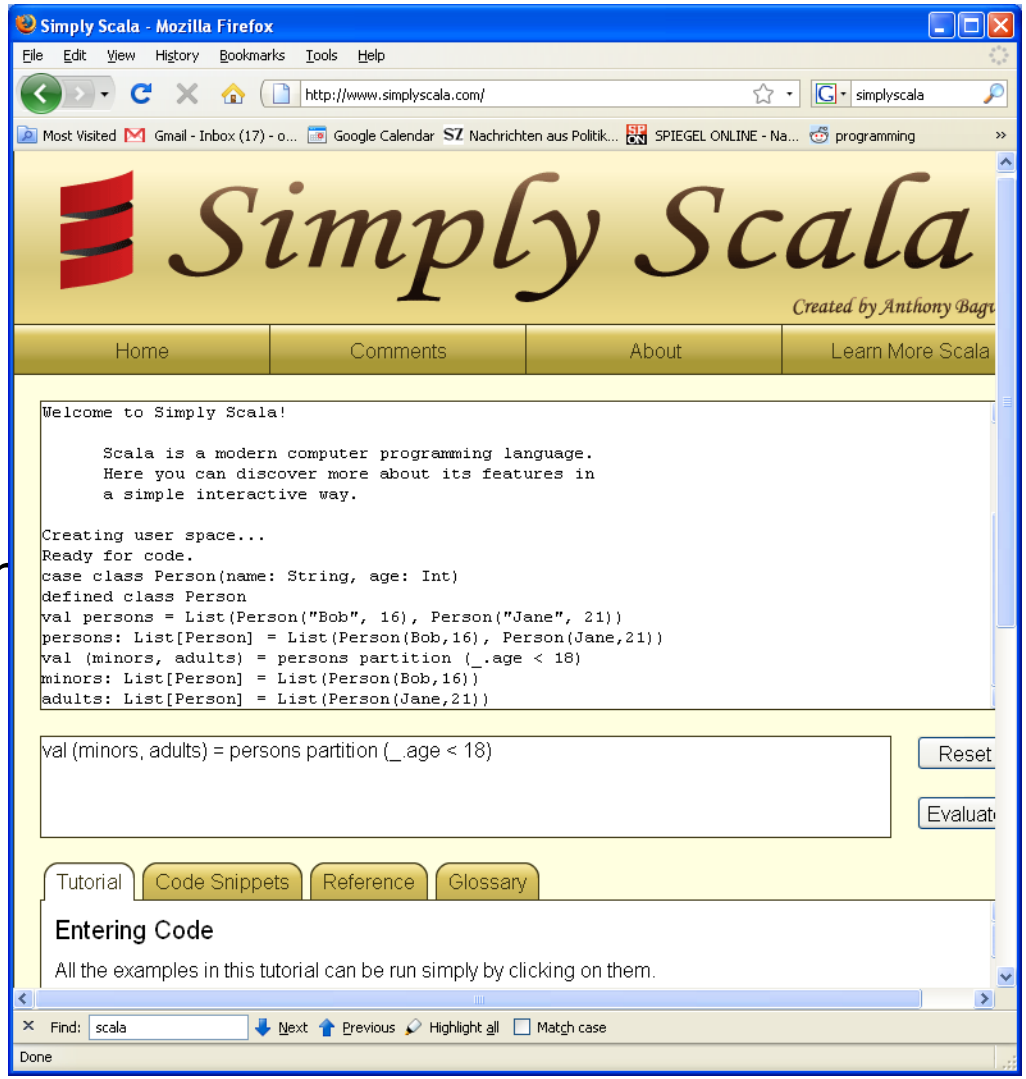
...all work out of the box.

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 this

Long term focus: Concurrency & Parallelism

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.

Questions??