Scala at Work

Martin Odersky
Scala Solutions and EPFL

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*”)

Momentum

Open-source language with

- Site scala-lang.org: 100K+ visitors/month
- 40,000 downloads/month, 10x growth last year
- 12 books in print
- Two conferences: Scala Liftoff and ScalaDays
- 33+ active user groups
- 60% USA, 30% Europe, 10% rest

LinkedIn

SONY make.believe

twitter

xerox

Sony Pictures imageworks

SIEMENS

Novell

NASA

eDF

foursquare

TomTom

Thatcham

Office DEPOT Taking Care of Business

NHS

OPOWER ENERGY EFFICIENCY. DELIVERED

guardian.co.uk

UBS

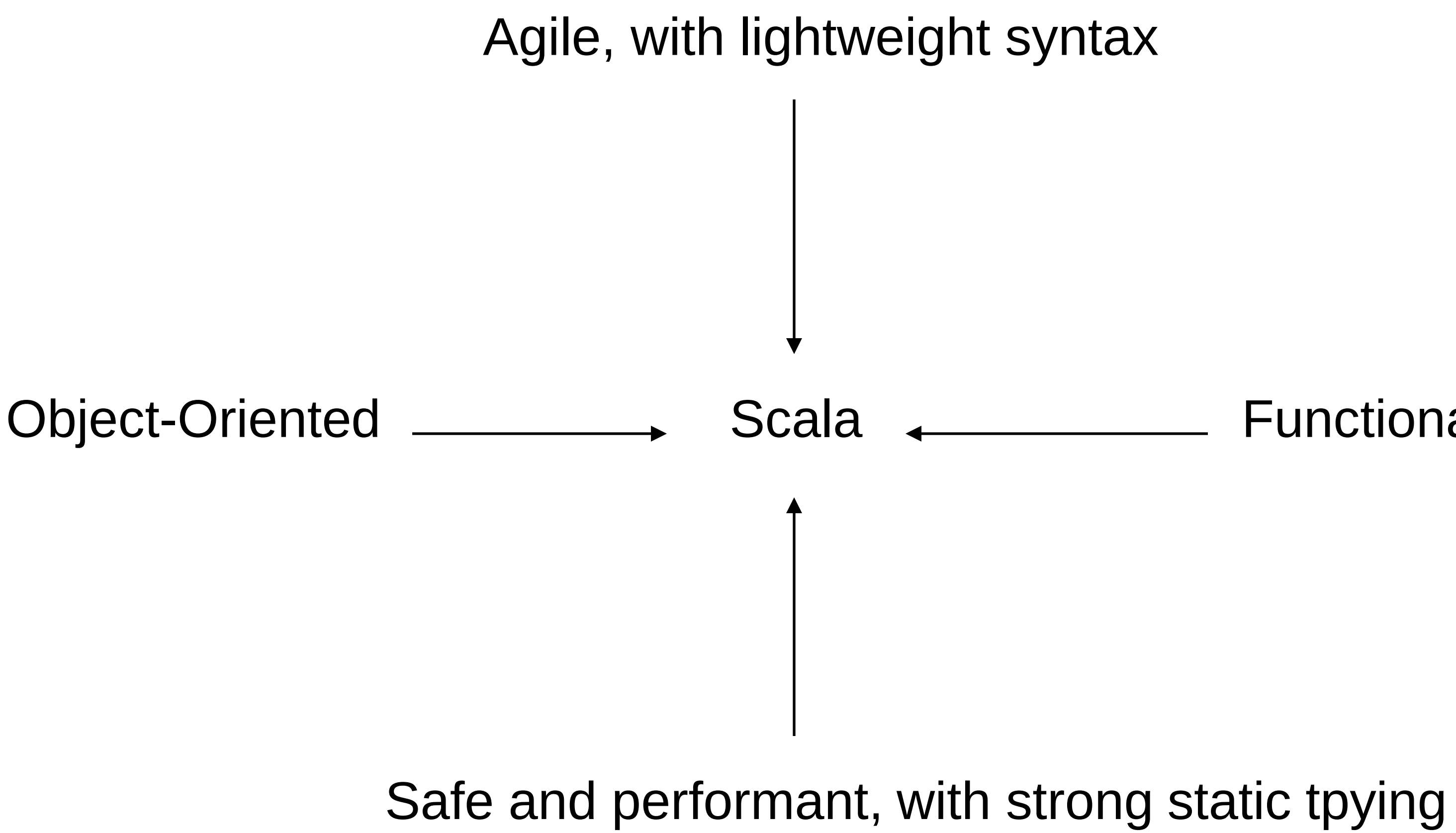
Autodesk

yammer

Scala Solutions

Why Scala?

Scala is a Unifier



Scala Solutions

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)
```

... 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 Solutions

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.

Scala Solutions

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:

- actors (akka, Twitter's message queues)
- specs, ScalaCheck
- ScalaQuery, squeryl, querulous

```
// 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 Solutions

Scalability demands extensibility

Take numeric data types:

- Today's languages support `int, long, float, double`.
- Should they also support `BigInt, BigDecimal, Complex, Rational, Interval, Polynomial`?

There are good reasons for each of these types

But a language combining them all would be too complex.

Better alternative: Let users *grow* their language according to their needs.

Adding new datatypes - seamlessly

For instance type **`BigInt`**:

```
def factorial(x: BigInt): BigInt =
  if (x == 0) 1 else x * factorial(x - 1)
```

Compare with using Java's class:

```
import java.math.BigInteger
def factorial(x: BigInteger): BigInteger =
  if (x == BigInteger.ZERO)
    BigInteger.ONE
  else
    x.multiply(factorial(x.subtract(BigInteger.ONE)))
}
```

Scala Solutions

Implementing new datatypes - seamlessly

Here's how BigInt is implemented

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

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

```
im         ava.math.BigInteger

clas     igInt(val bigInteger: BigInte
exten      java.lang.Number {

  def + (that: BigInt) =
    new BigInt(this.bigInteger add that.bigInteger)

  def - (that: BigInt) =
    new BigInt(this.bigInteger subtract that.bigInteger)

  …   // other methods implemented analogously
}
```

Scala Solutions

Adding new control structures

- For instance `using` for resource control (in Java 7)

```
using (new BufferedReader(new FileReader(path))) {
  f => println(f.readLine())
}
```

- Instead of:

```
val f = new BufferedReader(new FileReader(path))
try {
  println(f.readLine())
} finally {
  if (f != null)
    try f.close()
    catch { case ex: IOException => }
}
```

Implementing new control structures:

Here's how one would go about implementing **`using`**:

T is a type parameter...

… supporting a close method

```
def using[T <: { def close() }]
         (resource: T)
         (block: T => Unit) =
  try {
    block(resource)
  } finally {
    if (resource != null)
      try resource.close()
      catch { case ex: IOException => }
  }
```

A closure that takes a T parameter

Producer or Consumer?

Scala feels radically different for producers and consumers of advanced libraries.

For the consumer:

- Really easy
- Things work intuitively
- Can concentrate on domain, not implementation

For the producer:

- Sophisticated tool set
- Can push the boundaries of what's possible
- Requires expertise and taste

Scalability at work: Scala 2.8 Collections

Scala Solutions

Collection Properties

- object-oriented
- generic: `List[T], Map[K, V]`
- optionally persistent, e.g. `collection.immutable.Seq`
- higher-order, with methods such as `foreach, map, filter.`
- Uniform return type principle: Operations return collections of the same type (constructor) as their left operand, as long as this makes sense.

```
scala> val ys = List(1, 2, 3)
ys: List[Int] = List(1, 2, 3)

scala> val xs: Seq[Int] = ys
xs: Seq[Int] = List(1, 2, 3)

scala> xs map (_ + 1)
res0: Seq[Int] = List(2, 3, 4)

scala> ys map (_ + 1)
res1: List[Int] = List(2, 3, 4)
```

This makes a very elegant and powerful combination.

Using Collections: Map and filter

```
scala> val xs = List(1, 2, 3)
xs: List[Int] = List(1, 2, 3)

scala> val ys = xs map (x => x + 1)
ys: List[Int] = List(2, 3, 4)

scala> val ys = xs map (_ + 1)
ys: List[Int] = List(2, 3, 4)

scala> val zs = ys filter (_ % 2 == 0)
zs: List[Int] = List(2, 4)

scala> val as = ys map (0 to _)
as: List(Range(0, 1, 2), Range(0, 1, 2, 3), Range(0, 1, 2, 3, 4))
```

Scala Solutions

Using Collections: Flatmap

```
scala> val bs = as.flatten
bs: List[Int] = List(0, 1, 2, 0, 1, 2, 3, 0, 1, 2, 3, 4)

scala> val bs = ys flatMap (0 to _)
bs: List[Int] = List(0, 1, 2, 0, 1, 2, 3, 0, 1, 2, 3, 4)
```

Scala Solutions

Using Collections: For Notation

```
scala> for (x <- xs) yield x + 1                // same as map
res14: List[Int] = List(2, 3, 4)

scala> for (x <- res14 if x % 2 == 0) yield x   // ~ filter
res15: List[Int] = List(2, 4)

scala> for (x <- xs; y <- 0 to x) yield y       // same as flatMap
res17: List[Int] = List(0, 1, 0, 1, 2, 0, 1, 2, 3)
```

Using Maps

```
scala> val m = Map('1' -> "ABC", 2 -> "DEF", 3 -> "GHI")
m: Map[AnyVal, String] = Map((1,ABC), (2,DEF), (3,GHI))

scala> val m = Map(1 -> "ABC", 2 -> "DEF", 3 -> "GHI")
m: Map[Int, String] = Map((1,ABC), (2,DEF), (3,GHI))

scala> m(2)
res0: String = DEF

scala> m + (4 -> "JKL")
res1: Map[Int, String] = Map((1,ABC), (2,DEF), (3,GHI), (4,JKL))

scala> m map { case (k, v) => (v, k) }
res8: Map[String,Int] = Map((ABC,1), (DEF,2), (GHI,3))
```

Scala Solutions

An Example

- Task: Phone keys have mnemonics assigned to them.

```
val mnemonics = Map(
      '2' -> "ABC", '3' -> "DEF", '4' -> "GHI", '5' -> "JKL",
      '6' -> "MNO", '7' -> "PQRS", '8' -> "TUV", '9' -> "WXYZ")
```

- Assume you are given a dictionary `dict` as a list of words. Design a class `Coder` with a method `translate` such that

```
new Coder(dict).translate(phoneNumber)
```

  produces all phrases of words in dict that can serve as mnemonics for the phone number.

- Example: The phone number “7225276257” should have the mnemonic

```
Scala rocks
```

  as one element of the list of solution phrases.

Scala Solutions

Program Example: Phone Mnemonics

- This example was taken from:

  Lutz Prechelt: An Empirical Comparison of Seven Programming Languages. IEEE Computer 33(10): 23-29 (2000)

- Tested with Tcl, Python, Perl, Rexx, Java, C++, C

- Code size medians:
  - 100 loc for scripting languages
  - 200-300 loc for the others

Scala Solutions

Outline of Class Coder

```
import collection.mutable.HashMap

class Coder(words: List[String]) {

  private val mnemonics = Map(
      '2' -> "ABC", '3' -> "DEF", '4' -> "GHI", '5' -> "JKL",
      '6' -> "MNO", '7' -> "PQRS", '8' -> "TUV", '9' -> "WXYZ")

  /** Invert the mnemonics map to give a map from chars 'A' ... 'Z' to '2' ... '9' */
  private val upperCode: Map[Char, Char] = ??

  /** Maps a word to the digit string it can represent */
  private def wordCode(word: String): String = ??

  /** A map from digit strings to the words that represent them */
  private val wordsForNum = new HashMap[String, Set[String]] {
    override def default(number: String) = Set()
  }
  for (word <- words) wordsForNum(wordCode(word)) += word

  /** Return all ways to encode a number as a list of words */
  def encode(number: String): List[List[String]] = ??

  /** Maps a number to a list of all word phrases that can represent it */
  def translate(number: String): List[String] = encode(number) map (_ mkString " ")
}
```

Scala Solutions

Class Coder (1)

```
import collection.mutable.HashMap

class Coder(words: List[String]) {

  private val mnemonics = Map(
      '2' -> "ABC", '3' -> "DEF", '4' -> "GHI", '5' -> "JKL",
      '6' -> "MNO", '7' -> "PQRS", '8' -> "TUV", '9' -> "WXYZ")

  /** Invert the mnemonics map to give a map from chars 'A' ... 'Z' to '2' ... '9' */
  private val upperCode: Map[Char, Char] =
 for ((digit, str) <- m; letter <- str) yield (letter -> digit)

  /** Maps a word to the digit string it can represent */
  private def wordCode(word: String): String = word map (c => upperCode(c.toUpper))

  /** A map from digit strings to the words that represent them */
  private val wordsForNum = new HashMap[String, Set[String]] {
    override def default(number: String) = Set()
  }
  for (word <- words) wordsForNum(wordCode(word)) += word

  /** Return all ways to encode a number as a list of words */
  def encode(number: String): List[List[String]] = ??

  /** Maps a number to a list of all word phrases that can represent it */
  def translate(number: String): List[String] = encode(number) map (_ mkString " ")
}
```

Class Coder (2)

```
import collection.mutable.HashMap

class Coder(words: List[String]) {

  ...

  /** Return all ways to encode a number as a list of words */
  def encode(number: String): List[List[String]] =
    if (number.isEmpty)
      List(List())
    else
      for {
        splitPoint <- (1 to number.length).toList
        word <- wordsForNum(number take splitPoint)
        rest <- encode(number drop splitPoint)
      } yield word :: rest

  /** Maps a number to a list of all word phrases that can represent it */
  def translate(number: String): List[String] = encode(number) map (_ mkString " ")
}
```

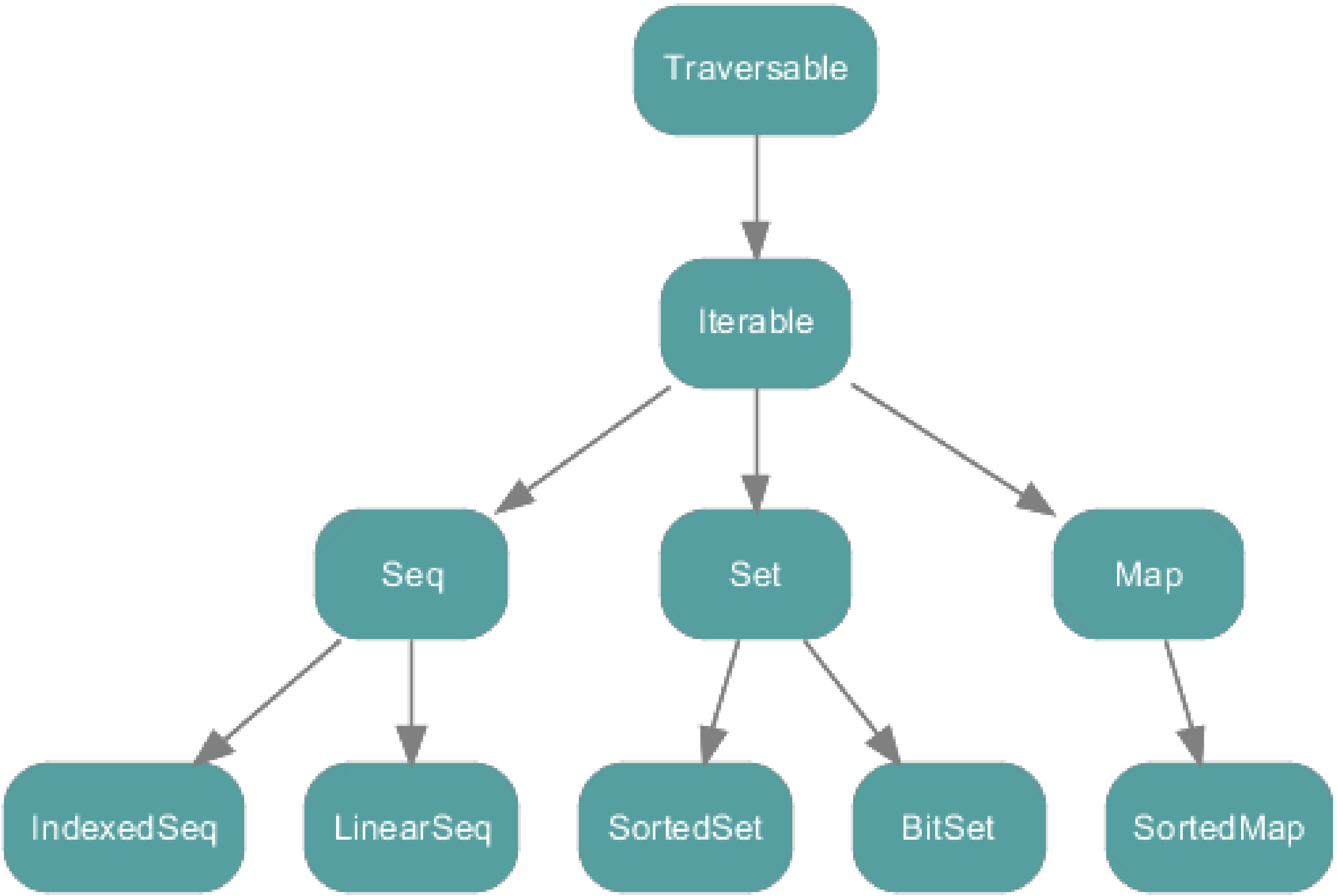
Scala Solutions

How is all this implemented?

Everything is a Library

- Collections feel like they are an organic part of Scala
- But in fact the language does not contain *any* collection-related constructs
  - no collection types
  - no collection literals
  - no collection operators
- *Everything* is done in a library
- *Everything* is extensible
  - You can write your own collections which look and feel like the standard ones

Some General Scala Collections



Mutable or Immutable?

- All general collections come in three forms, and are stored in different packages:

  scala.collection

  scala.collection.mutable

  scala.collection.immutable

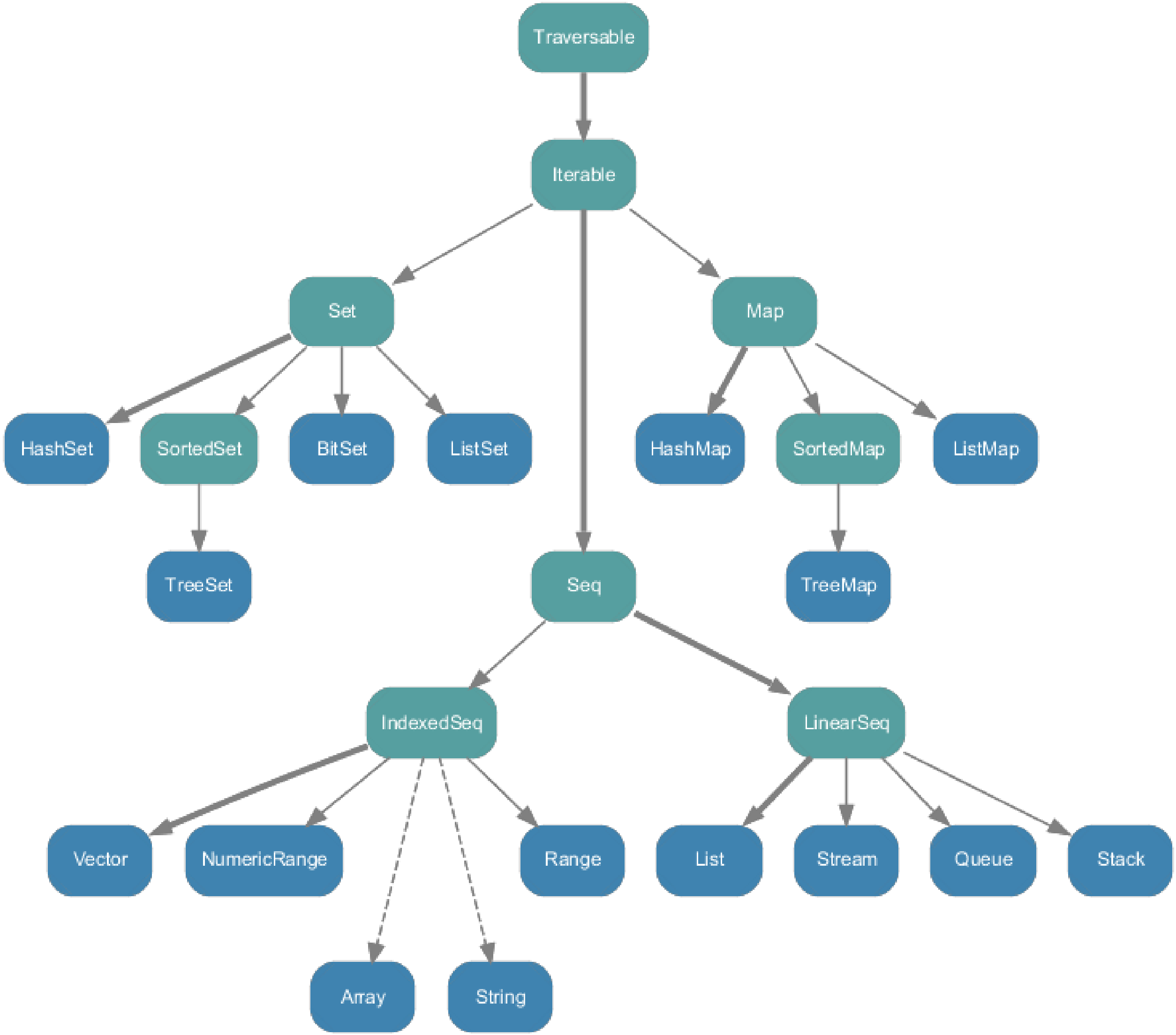
- Immutable is the default, i.e. predefined imports go to `scala.collection.immutable`
- General collections in `scala.collection` can be mutable or immutable.
- There are aliases for the most commonly used collections.

  `scala.collection.immutable.List` *where it is defined*

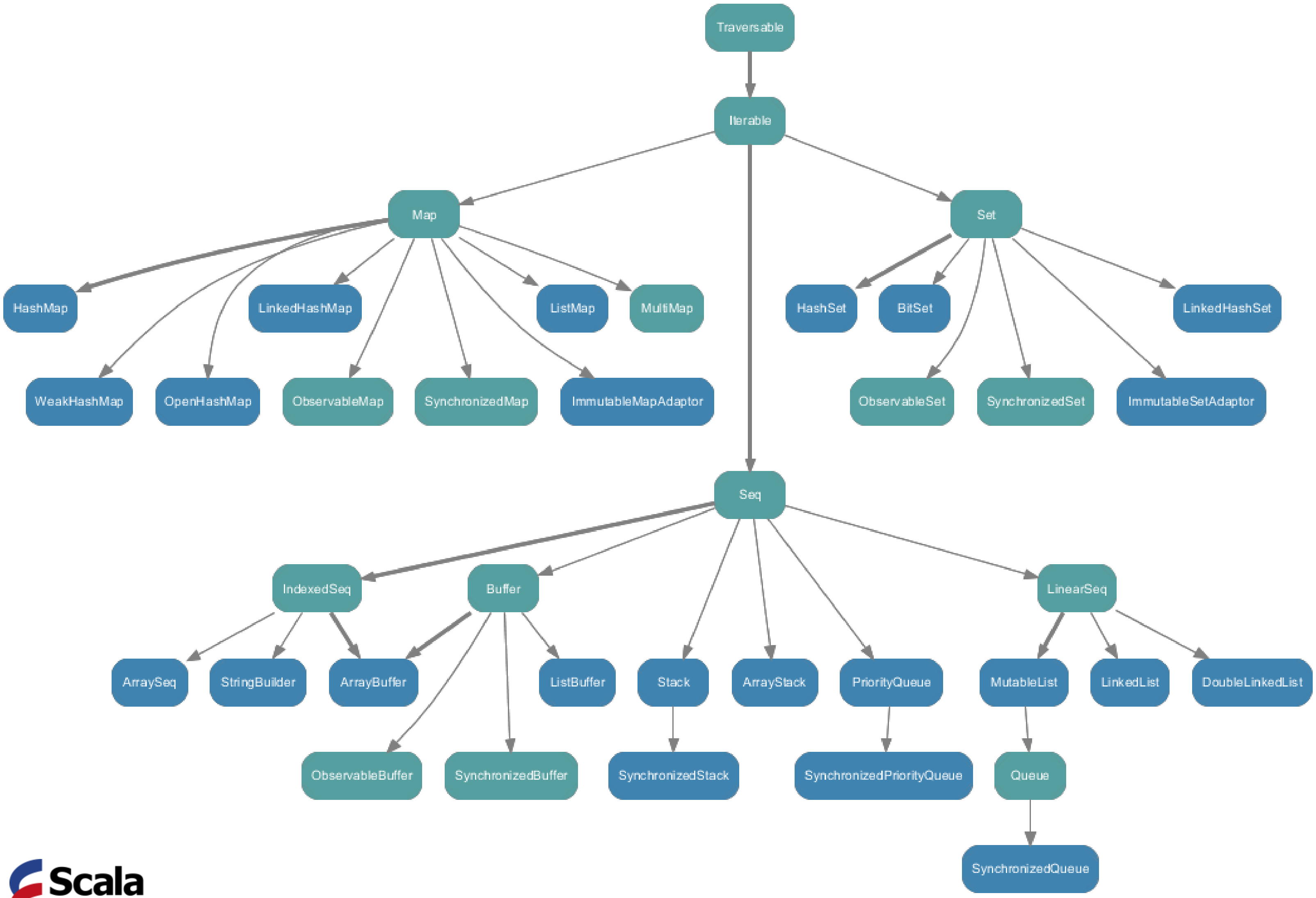
  `scala.List` *the alias in the scala package*

  `List` *because scala._ is automatically imported*

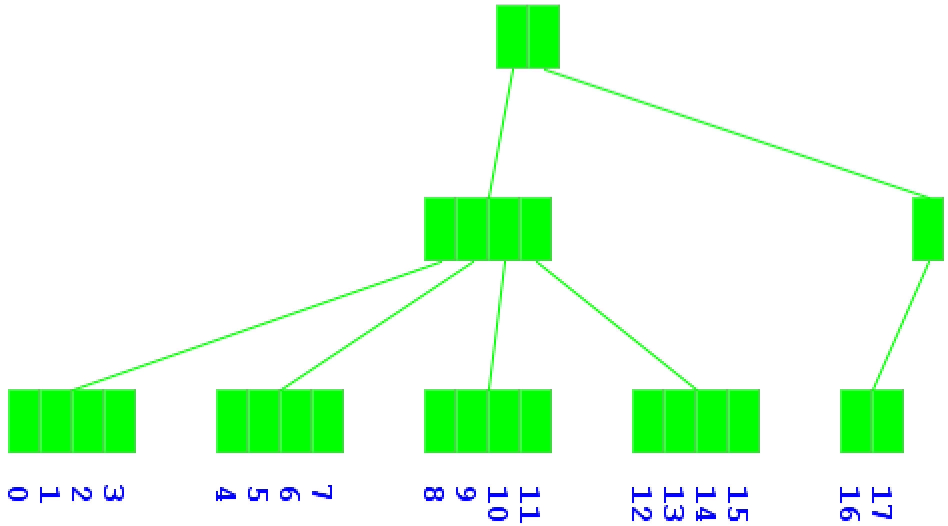
Immutable Scala Collections



Mutable Scala Collections



New Implementations: Vectors and Hash Tries



- Trees with branch factor of 32.
- Persistent data structures with very efficient sequential and random access.
- Invented by Phil Bagwell, then adopted in Clojure.
- New: Persistent prepend/append/update in constant amortized time.
- Next: Fast splits and joins for parallel transformations.

The Uniform Return Type Principle

Bulk operations return collections of the same type (constructor) as their left operand. (DWIM)

```
scala> val ys = List(1, 2, 3)
ys: List[Int] = List(1, 2, 3)

scala> val xs: Seq[Int] = ys
xs: Seq[Int] = List(1, 2, 3)

scala> xs map (_ + 1)
res0: Seq[Int] = List(2, 3, 4)

scala> ys map (_ + 1)
res1: List[Int] = List(2, 3, 4)
```

This is tricky to implement without code duplication!

Pre 2.8 Collection Structure

```
trait Iterable[A] {
  def filter(p: A => Boolean): Iterable[A] = ...
  def partition(p: A => Boolean) =
    (filter(p(_)), filter(!p(_)))
  def map[B](f: A => B): Iterable[B] = ...
}

trait Seq[A] extends Iterable[A] {
  def filter(p: A => Boolean): Seq[A] = ...
  override def partition(p: A => Boolean) =
    (filter(p(_)), filter(!p(_)))
  def map[B](f: A => B): Seq[B] = ...
}
```

Scala Solutions

Types force duplication

`filter` needs to be re-defined on each level

`partition` also needs to be re-implemented on each level, even though its definition is everywhere the same.

The same pattern repeats for many other operations and types.

Signs of Bit Rot

Lots of duplications of methods.

- Methods returning collections have to be repeated for every collection type.

Inconsistencies.

- Sometimes methods such as filter, map were not specialized in subclasses
- More often, they only existed in subclasses, even though they could be generalized

"Broken window" effect.

- Classes that already had some ad-hoc methods became dumping grounds for lots more.
- Classes that didn't stayed clean.

Scala Solutions

Excerpts from List.scala

File Edit Options Buffers Tools Scala Help

```scala
   * and elements are in the range between `start` (inclusive)
   * and `end` (exclusive)
   *
   *  @param start the start value of the list
   *  @param end   the end value of the list
   *  @param step  the increment function of the list, which given `v
   *               computes `v<sub>n+1</sub>`. Must be monotonically
   *               or decreasing.
   *  @return      the sorted list of all integers in range [start;en
   */
  @deprecated("use `iterate' instead")
  def range(start: Int, end: Int, step: Int => Int): List[Int] = {
    val up = step(start) > start
    val down = step(start) < start
    val b = new ListBuffer[Int]
    var i = start
    while ((!up || i < end) && (!down || i > end)) {
      b += i
      val next = step(i)
      if (i == next)
        throw new IllegalArgumentException("the step function did no
      i = next
    }
    b.toList
  }

  /** Create a list containing several copies of an element.
   *
   *  @param n    the length of the resulting list
   *  @param elem the element composing the resulting list
   *  @return     a list composed of n elements all equal to elem
   */
  @deprecated("use `fill' instead")
  def make[A](n: Int, elem: A): List[A] = {
    val b = new ListBuffer[A]
    var i = 0
    while (i < n) {
      b += elem
      i += 1
    }
    b.toList
  }
--(Unix)--  List.scala    55% L477    (Scala)----------------------------
```

```scala
   *  @param xs the iterable of pairs to unzip
   *  @return a pair of lists.
   */
  @deprecated("use `xs.unzip' instead of `List.unzip(xs)'")
  def unzip[A,B](xs: Iterable[(A,B)]): (List[A], List[B]) =
      xs.foldRight[(List[A], List[B])]((Nil, Nil)) {
        case ((x, y), (xs, ys)) => (x :: xs, y :: ys)
      }

  /**
   * Returns the `Left` values in the given `Iterable`
   * of `Either`s.
   */
  @deprecated("use `xs partialMap { case Left(x: A) => x }' inste
  def lefts[A, B](es: Iterable[Either[A, B]]) =
    es.foldRight[List[A]](Nil)((e, as) => e match {
      case Left(a) => a :: as
      case Right(_) => as
    })

  /**
   * Returns the `Right` values in the given`Iterable` of  `Eithe
   */
  @deprecated("use `xs partialMap { case Right(x: B) => x }' inst
  def rights[A, B](es: Iterable[Either[A, B]]) =
    es.foldRight[List[B]](Nil)((e, bs) => e match {
      case Left(_) => bs
      case Right(b) => b :: bs
    })

  /** Transforms an Iterable of Eithers into a pair of lists.
   *
   *  @param xs the iterable of Eithers to separate
   *  @return a pair of lists.
   */
  @deprecated("use `Either.separate' instead")
  def separate[A,B](es: Iterable[Either[A, B]]): (List[A], List[B
      es.foldRight[(List[A], List[B])]((Nil, Nil)) {
      case (Left(a), (lefts, rights)) => (a :: lefts, rights)
      case (Right(b), (lefts, rights)) => (lefts, b :: rights)
    }
--(Unix)--  List.scala    65% L538    (Scala)------------------------
```

How to do better?

Can we abstract out the return type?

Look at `map`: Need to abstract out the type constructor, not just the type.

```
trait Iterable[A]
  def map[B](f: A => B): Iterable[B]

trait Seq[A]
  def map[B](f: A => B): Seq[B]
```

But we can do that using Scala's higher-kinded types!

HK Types Collection Structure

```
trait TraversableLike[A, CC[X]] {
  def filter(p: A => Boolean): CC[A]
  def map[B](f: A => B): CC[B]
}

trait Traversable[A] extends TraversableLike[A, Traversable]
trait Iterable[A]    extends TraversableLike[A, Iterable]
trait Seq[A]         extends TraversableLike[A, Seq]
```

Here, CC is a parameter representing a type constructor.

Implementation with Builders

All ops in Traversable are implemented in terms of `foreach` and `newBuilder`.

```
trait Builder[A, Coll] {
  def += (elem: A)    // add elems
  def result: Coll    // return result
}
trait TraversableLike[A, CC[X]] {
  def foreach(f: A => Unit)
  def newBuilder[B]: Builder[B, CC[B]]
  def map[B](f: A => B): CC[B] = {
    val b = newBuilder[B]
    foreach (x => b += f(x))
    b.result
  }
}
```

Scala Solutions

Unfortunately ...

... things are not as parametric as it seems at first. Take:

```
class BitSet extends Set[Int]
```

```
scala> val bs = BitSet(1, 2, 3)
bs: scala.collection.immutable.BitSet = BitSet(1, 2, 3)

scala> bs map (_ + 1)
res0: scala.collection.immutable.BitSet = BitSet(2, 3, 4)

scala> bs map (_.toString + "!")
res1: scala.collection.immutable.Set[java.lang.String] = Set(1!, 2!, 3!)
```

Note that the result type is the “best possible” type *that fits the element type of the new collection*.

Other examples: `SortedSet, String`.

Scala Solutions

How to advance?

We need more flexibility. Can we define our own type system for collections?

Question: Given old collection type `From`, new element type `Elem`, and new collection type `To`:

Can an operation on `From` build a collection of type `To` with `Elem` elements?

Captured in: `CanBuildFrom[From, Elem, To]`

Facts about CanBuildFrom

Can be stated as axioms and inference rules:

```
CanBuildFrom[Traversable[A], B, Traversable[B]]

CanBuildFrom[Set[A], B, Set[B]]

CanBuildFrom[BitSet, B, Set[B]]

CanBuildFrom[BitSet, Int, BitSet]

CanBuildFrom[String, Char, String]

CanBuildFrom[String, B, Seq[B]]

CanBuildFrom[SortedSet[A], B, SortedSet[B]]  :-  Ordering[B]
```

where A and B are arbitrary types.

Scala Solutions

Implicitly Injected Theories

Type theories such as the one for `CanBuildFrom` can be injected using implicits.

A predicate:

```
trait CanBuildFrom[From, Elem, To] {
  def apply(coll: From): Builder[Elem, To]
}
```

Axioms:

```
implicit def bf1[A, B]: CanBuildFrom[Traversable[A], B, Traversable[B]]
implicit def bf2[A, B]: CanBuildFrom[Set[A], B, Set[B]]
implicit def bf3: CanBuildFrom[BitSet, Int, BitSet]
```

Inference rule:

```
implicit def bf4[A, B] (implicit ord: Ordering[B])
                   : CanBuildFrom[SortedSet[A], B, SortedSet[B]]
```

Connecting with Map

- Here's how `map` can be defined in terms `CanBuildFrom`:

```
trait TraversableLike[A, Coll] { this: Coll =>
  def foreach(f: A => Unit)
  def newBuilder: Builder[A, Coll]
  def map[B, To](f: A => B)
               (implicit cbf: CanBuildFrom[Coll, B, To]): To = {
    val b = cbf(this)
    foreach (x => b += f(x))
    b.result
  }
}
```

Objections

s the Scala 2.8 collections library a case of “the longest suicide note in history” ?

43

32

First note the inflammatory subject title is a quotation made about the manifesto of a UK political party in the early 1980s. This question is subjective but it is a genuine question, I've made it CW and I'd like some opinions on the matter.

Despite whatever my wife and coworkers keep telling me, I don't think I'm an idiot: I have a good degree in mathematics from the University of Oxford and I've been programming commercially for almost 12 years and in Scala for about a year (also commercially).

I have just started to look at the Scala collections library re-implementation which is coming in the imminent **2.8** release. Those familiar with the library from 2.7 will notice that the library, from a usage perspective, has changed little. For example...

```
> List("Paris", "London").map(_.length)
res0: List[Int] List(5, 6)
```

...would work in either versions. **The library is eminently useable**; in fact it's fantastic. However, those

Hello World!

Stack Overflow is a **collaboratively edited question and answer site for programmers** – regardless of platform or language. It's 100% free, no registration required.

about » faq »

tagged

subjective × 6349

collections × 1010

scala × 849

scala-2.8 × 32

asked

Use Cases

- How to explain

```
def map[B, To](f: A => B)
                        (implicit cbf: CanBuildFrom[Coll, B, To]): To
```

  to a beginner?
- Key observation: We can *approximate* the type of `map`.
- For everyone but the most expert user

```
def map[B](f: A => B): Traversable[B]  // in  class Traversable
def map[B](f: A => B): Seq[B]          // in  class Seq, etc
```

  is detailed enough.
- These types are correct, they are just not as general as the type that's actually implemented.

Part of the Solution: Flexible Doc Comments

```
def lastOption: Option[A]
```
Optionally selects the last element

```
def map[B](f: (A) ⇒ B): Traversable[B]
```
[use case]

Builds a new collection by applying a function to all elements of this collection.

B	the element type of the returned collection.
f	the function to apply to each element.
returns	a new collection resulting from applying the given function `f` to each element of this collection and collecting the results.

attributes: abstract

```
def map[B, That](f: (A) ⇒ B)(implicit bf: CanBuildFrom[Traversable[A], B, That]): That
```
Builds a new collection by applying a function to all elements of this collection.

B	the element type of the returned collection.
That	the class of the returned collection. Where possible, `That` is the same class as the current collection class `Repr`, but this depends on the element type `B` being admissible for that class, which means that an implicit instance of type `CanBuildFrom[Repr, B, That]` is found.
f	the function to apply to each element.
bf	an implicit value of class `CanBuildFrom` which determines the result class `That` from the current representation type `Repr` and and the new element type `B`.
returns	a new collection of type `That` resulting from applying the given function `f` to each element of this collection and collecting the results.

definition classes: TraversableLike

```
def max[B >: A](implicit cmp: Ordering[B]): A
```
Finds the largest element

```
def min: A
```
[use case] Finds the largest element

Going Further

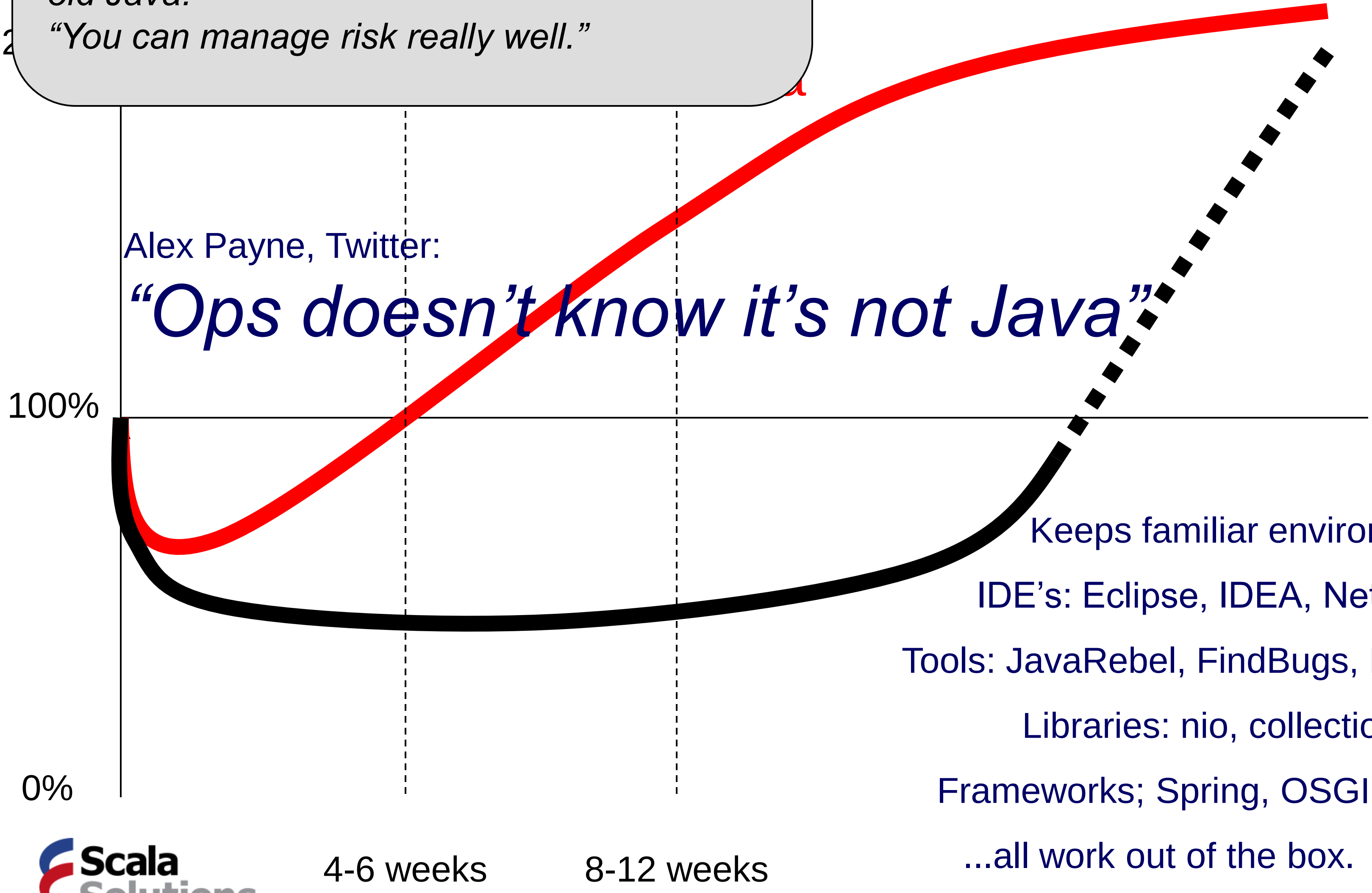
- In Scala 2.9, collections will support parallel operations.
- Will be out by January 2011.
- The right tool for addressing the PPP (popular parallel programming) challenge.
- I expect this to be the cornerstone for making use of multicores for the rest of us.

But how long will it take me to switch?

Scala Solutions

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, OSGI, J2EE, ...

...all work out of the box.

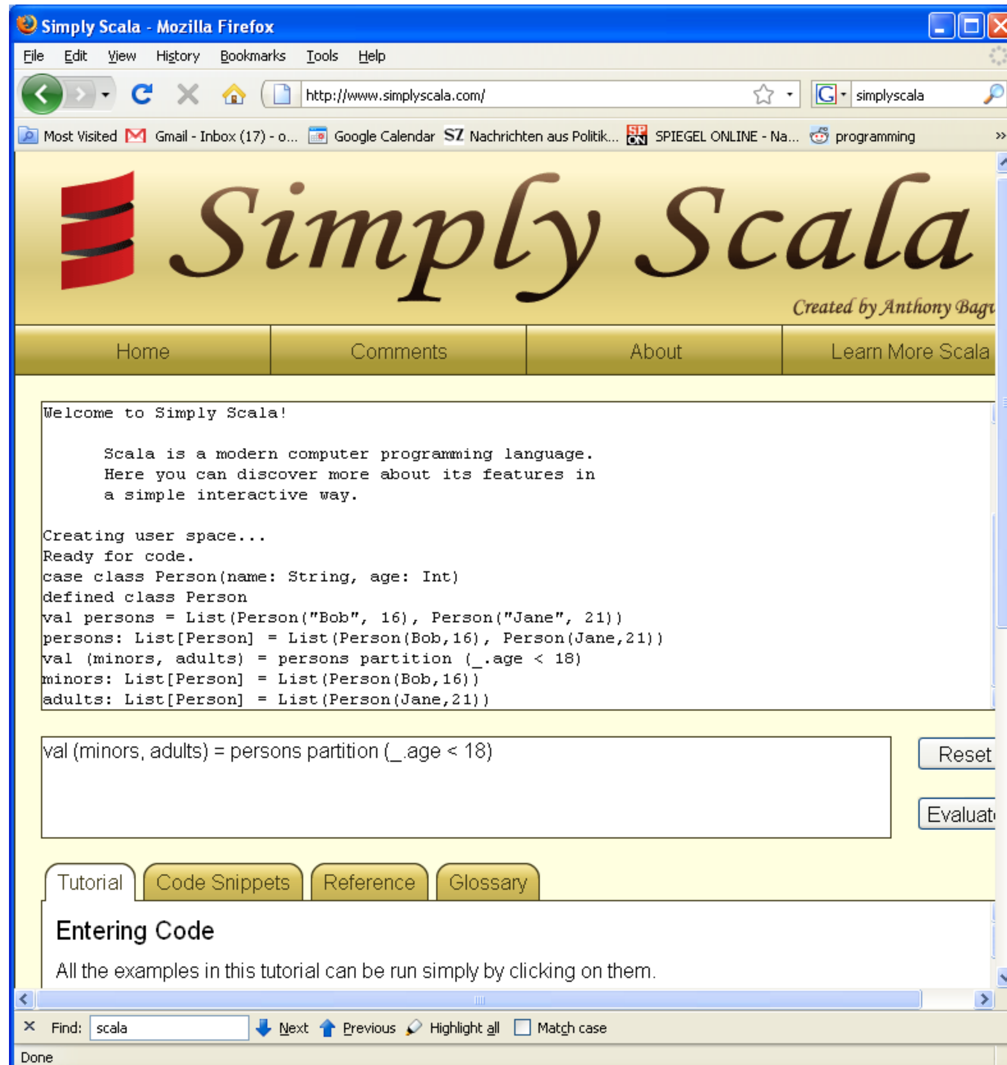
Scala Solutions

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



Scala Solutions

How to find out more

Scala site: www.scala-lang.org

12 books

Support

Open Source Ecosystem ...

akka	scalable actors
sbt	simple build tool
lift, play	web frameworks
kestrel, querulous	middleware from Twitter
Migrations	middleware from Sony
ScalaTest, specs, ScalaCheck	testing support
ScalaModules	OSGI integration

... complemented by commercial support

Thank You


Scala Solutions
Search
Your shopping basket is empty.
View your cart | Checkout now
Home The Company Products Training Consulting Partners Community What is Scala? Jobs
Scala Solutions
Unrivalled Expertise
Pro Development Products
Top-notch Consultants
We have founded Scala Solutions to provide the tools and services needed
To do this, Scala Solutions has brought together an outstanding technical team that has a passion for programming, takes pride in providing quality applications and is led by people who have unrivalled expertise in the field of professional Scala development. We want to make great tools for developers whether professional or just because they enjoy using Scala.
Many companies are already using Scala for mission critical applications and many more are on the way. People depend on Scala. Now you can depend on Scala Solutions to provide the development products and essential support services you need to be even more successful.
If you need our services or want to know more, just call, we would enjoy working
Training Lausanne
We hold regular training courses at our facility in Lausanne.
Next Course
Object Oriented Meets Functional, 25-26 November 2010, 1,500 chf
Call or send a message to reserve a place for the course.
On-line registration and payment will be available shortly.
Registration and Course Description
London 6-7 December 2010, Skills Matter
Registration and Course Description
Paris, 31 January - 1 February 2011, Xebia
Products
Migration Manager
Stable Versions
Consulting
On-Site Training

Scala cheat sheet (1): Definitions

Scala method definitions:

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

or
```
def fun(x: Int) = result
```

```
def fun = result
```

Scala variable definitions:

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

or
```
var x = expression
val x = expression
```

Java method definition:

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

(no parameterless methods)

Java variable definitions:

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

Scala Solutions

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

if (cond) return expr1;
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 Solutions

Scala cheat sheet (4): Traits

Scala Trait

```
trait T {
  def absMeth(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 absMeth(String x)

  (no concrete methods)


  (no fields)
}
```

Java extension + implementation:

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