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:

   ("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
Why Scala?
Scala is a Unifier

Agile, with lightweight syntax

Object-Oriented → Scala ← Functional

Safe and performant, with strong static typing
Let’s see an example:
A class ...

... in Java:

```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:

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

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

---

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, sqreyl, querulous

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

```scala
// asynchronous message send
actor ! message

// message receive
receive {
  case msgpat₁ ⇒ action₁
  ...
  case msgpatₙ ⇒ actionₙ
}
```
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`:

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

Compare with using Java's class:

```java
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

```scala
import java.math.BigInteger

class BigInt(val bigInteger: BigInteger) extends 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
}
```

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

• For instance using for resource control (in Java 7)

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

• Instead of:

```scala
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`:

```scala
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 => }
}
```

`T` is a type parameter supporting a `close` method. 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
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.

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))
```
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)
Using Collections: For Notation

```scala
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
scala> val m = Map('1' -> "ABC", 2 -> "DEF", 3 -> "GHI")
m: Map[Char, 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))
```
An Example

• Task: Phone keys have mnemonics assigned to them.
  ```scala
  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
  ```scala
  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
  Scala rocks
  ```
as one element of the list of solution phrases.
Program Example: Phone Mnemonics

- This example was taken from:

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

- Code size medians:
  - 100 loc for scripting languages
  - 200-300 loc for the others
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 " ")
}
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) <- mnemonics) 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 (1)
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 " ")
}
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
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
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

```scala
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] = ...
}
```
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.
Excerpts from List.scala

```scala
// and elements are in the range between `start` (inclusive) and `end` (exclusive)

@require start <= end 

/**
 * @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 gives \( \nu < \langle \text{sub} \n+ 1/\text{sub} \rangle \). Must be monotonically increasing or decreasing.
 * @return the sorted list of all integers in range [start;en)
 */
@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`
 *
 */
def lefts[A,B](xs: Iterable[Either[A,B]]): List[A] = xs.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 `Either`
 *
 */
def rights[A,B](xs: Iterable[Either[A,B]]): List[B] = xs.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.
 *
 */
def separate[A,B](xs: Iterable[Either[A,B]]): (List[A], List[B]) = xs.foldRight((List[A],List[B]))((Nil,Nil)) {
  case (Left(a),(lefts,rights)) => (a :: lefts, rights)
  case (Right(b),(lefts,rights)) => (lefts, b :: rights)
}
```
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.

```scala
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-kindled types!
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.

```scala
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
  }
}
```
Unfortunately ... 

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

```scala
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.
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.
Implicitly Injected Theories

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

A predicate:

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

Axioms:

```scala
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:

```scala
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 of `CanBuildFrom`:

```scala
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
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...

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

...would work in either version. The library is eminently useable: in fact it's fantastic. However, those
Use Cases

• How to explain
  ```scala
  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
  ```scala
  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

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

// def map[B](f: (A) => B): Traversable[B]
// Builds a new collection by applying a function to all elements of this collection.
int B, // the element type of the returned collection.
int f, // the function to apply to each element.
int 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.
int B, // the element type of the returned collection.
int 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.
int f, // the function to apply to each element.
int 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.
int 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]
// 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?
Learning Curves

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.

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.”

Alex Payne, Twitter:
“Ops doesn’t know it’s not Java”
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

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 cheat sheet (1): Definitions

### Scala method definitions:

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

or

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

```scala
def fun = result
```

### Scala variable definitions:

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

or

```scala
var x = expression
val x = expression
```

### Java method definition:

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

(no parameterless methods)

### Java variable definitions:

```java
int x = expression
final String x = expression
```
Scala method calls:

```
obj.meth(arg)
```

or

```
obj meth arg
```

Scala choice expressions:

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

```
expr match {
  case pat_1 => expr_1
  ....
  case pat_n => expr_n
}
```

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 pat_1 : return expr_1;
  ...
  case pat_n : return expr_n;
} // statement only
Scala cheat sheet (3): Objects and Classes

Scala Class and Object

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

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

Java Class with static

```java
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**

```scala
trait T {
  def absMeth(x: String): String
  def concreteMeth(x: String) = x + field
  var field = "!
}
```

**Scala mixin composition:**

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

**Java Interface**

```java
interface T {
  String absMeth(String x)
  // (no concrete methods)
  // (no fields)
}
```

**Java extension + implementation:**

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