Scala – The Next 5 Years

Martin Odersky
It’s been quite a ride
The last 6 years

2003  Scala first used in classroom at EPFL.
2004  First public release, Jamie & Jon at Sygneca are probably first outside professional users.
2005  Scalable component abstractions paper.
2006  2.x version released, David Pollak gets interested.
2007  First JavaOne talk.
2008  First Scala LiftOff Unconference.
2009  JavaOne talks on Scala almost fill a full day.
Even more rewarding:

We have a fantastic user community:
Friendly, helpful, and intelligent.

A big Thank You to our contributors. We could not have done it without you.
What is Scala, exactly?
A scripting language?

Scala is a regular contender at JavaOne’s ScriptBowl.

Syntax is lightweight and concise, due to:

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

Average reduction in LOC wrt Java: ≥ 2
due to concise syntax and better abstraction capabilities.
Explorative programming with a read-eval-print-loop (REPL)

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

capital += "Russia" → "Moskow"

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

assert ( capital("Japan") == "Tokyo" )
```
The Java of the future?

It has basically everything Java has now.
(sometimes in different form)
It has closures.
(proposed for Java 7, but rejected)
It has traits and pattern matching.
(I would not be surprised to see them in Java 8, 9 or 10)
It compiles to .class files, is completely interoperable and runs about as fast as Java

```
object App {
  def main(args: Array[String]) {
    if (args exists (_.toLowerCase == "-help"))
      printUsage()
    else
      process(args)
  }
}
```
A language for concurrent programming?

It has powerful concurrency abstractions such as actors.

It’s used in heavy duty applications: kestrel message queue, lift web framework.

It makes functional programming with immutable data easy.

```java
// asynchronous message send
actor ! message

// message receive
receive {
    case msgpat_1 => action_1
    ...
    case msgpat_n => action_n
}
```
A composition language?

- New approach to module systems:
  - component = class or trait composition via mixins
- Components connect with
  - parameters,
  - abstract members (both types and values),
  - self types
- gives dependency injection for free

```
trait Analyzer { this: Backend =>
    ...
  }

trait Backend extends Analyzer
    with Optimization
    with Generation {
    val global: Main
    import global._

    type OutputMedium <: Writable
  }
```
How can it be all these things?

Q. Is there not a danger that a language being all of these things becomes too complex?

A. Normally, yes. However, Scala is deep where other languages are broad.

Cuts down on boilerplate.

Concentrates on powerful abstraction methods, so that users can write powerful libraries.
The Essence of Scala

Key design principles of Scala:

1. Concentrate on scalability: The same constructions should work for small as well as large programs.

2. Unify object-oriented and functional programming.
Where do we go from here?

• Summer/fall 2009: Scala 2.8, with:
  – new collections
  – package objects
  – named and default parameters
  – `@specialized` for faster generics
  – improved tool support for IDEs and REPL.

• The next 5 years:

  Concurrency // Parallelism
  (on all levels: primitives, abstractions, types, tools)
Old Scala collections

• Pre 2.8 collections have grown gradually. They lack a uniform structure.
• Examples:

  scala.Seq
  scala.RandomAccessSeq.Mutable
  scala.collection.mutable.Set

• Operations like map need to be redefined for every collection type:

  List(1, 2, 3).map(inc) : List[Int]
  Array(1, 2, 3).map(inc): Array[Int]
  Set(1, 2, 3).map(inc) : Set[Int]
New Scala collections

• Uniform structure:
  scala.collection.immutable.List
  scala.collection.immutable.Set
  scala.collection.mutable.Vector

• Every operation is implemented only once.
• Selection of building blocks in package
  scala.collection.generic.
Collection Hierarchy

Traversable
  |
  |
  Iterable
    |
    |
    Map     Set     Sequence
      |
      |
      SortedMap  SortedSet  BitSet  Buffer  Vector  LinearSequence
Package Objects

Problem: Everything has to live inside an object or class. => No top-level value definitions, type aliases, implicits.

Package objects make packages more like objects.

They also let you define your own "Predefs".

```
package object scala {
  type List[+T] = scala.collection.immutable.List
  val List = scala.collection.immutable.List
...
}
```
Named and default arguments

```kotlin
def newTable(size: Int = 100, load: Float = 0.5f) =
    new HashMap[String, String] {
        override val initialSize = size
        override val loadFactor = (load * 1000).toInt
    }

newTable(50, 0.75f)
newTable()
newTable(load = 0.5f)
newTable(size = 20, load = 0.5f)
```

Defaults can be arbitrary expressions, not just constants. Default evaluation is dynamic.
Copy method

Problem: Case classes are great for pattern matching, but how do you transform data?
Often need to update (functionally) some specific field(s) of a data structure defined by case classes.

Solution: Selective copy method.

```scala
case class Tree[+T](elem: T, left: Tree[T], right: Tree[T])

def incElem(t: Tree[Int]) = t copy (elem = t.elem + 1)
```

- Problem: How to define “copy”?
Defining copy

• Problem: A class with \( n \) parameters allows \( 2^n \) possible combinations of parameters to copy.
• Default parameters to the rescue.

```scala
case class Tree[+T](elem: T, left: Tree[T], right: Tree[T]) {
  // generated automatically:
  def copy[U](elem: U = this.elem,
               left: Tree[U] = this.left,
               right: Tree[U] = this.right) =
    Branch(elem, left, right)
}
```
Using copy

```scala
case class Tree[T](elem: T, left: Tree[T], right: Tree[T])
defined class Tree

val t = Tree(1, null, null)
t: Tree[Int] = Tree(1,null,null)

val u = t.copy(left = Tree(2, null, null))
u: Tree[Int] = Tree(1,Tree(2,null,null),null)

val v = u.copy(elem = 3, right = Tree(4, null, null))
v: Tree[Int] = Tree(3,Tree(2,null,null),Tree(4,null,null))
```
Faster Generics

- Problem: Generics cause a performance hit because they require a uniform representation: everything needs to be boxed.

- Example:

```scala
trait Function1[-T, +U] { def apply(x: T): U }
def app(x: Int, fn: Int => Int) = fn(x)
app(7, x => x * x)
```

becomes:

```scala
trait Function1 { def apply(x: Object): Object }
class anonfun extends Function1 {
  def apply(x: Object): Object =
    Int.box(apply(Int.unbox(x)))
  def apply(x: Int): Int = x * x
}
def app(x: Int, fn: Function1) =
  Int.unbox(fn(Int.box(x)))
app(7, new anonfun)
```
Solution: @specialized

- We should *specialize* functions that take primitive parameter types.
- If we had the whole program at our disposal, this would be easy: Just do it like C++ templates.
- The problem is that we don’t.
- Idea: specialize opportunistically, when types are known.
If Function1 is `@specialized` ...

```scala
trait Function1[@specialized -T, @specialized +U] {
  def apply(x: T): U
}
def app(x: Int, fn: Int => Int) = fn(x)
app(7, x => x * x)
```

... the compiler generates instead:

```scala
trait Function1 {
  def apply(x: Object): Object
  def applyIntInt(x: Int): Int = Int.unbox(apply(Int.box(x)))
  // specializations for other primitive types
}
class anonfun extends (Int => Int) {
  def apply(x: Object): Object = Int.box(applyIntInt(Int.unbox(x)))
  override def applyIntInt(x: Int): Int = x * x
}
def app(x: Int, fn: Function1) = fn.applyIntInt(x)
app(7, anonfun)
```

No boxing!
Better tools

• REPL with command completion (courtesy Paul Phillips).

• Reworked IDE/compiler interface (Miles Sabin, Martin Odersky, Iulian Dragos).

Developed originally for Eclipse, but should be reusable for other IDEs such as Netbeans, VisualStudio.
New control abstractions:

- Continations plugin
  - allow to unify receive and react in actors.
  - give a clean basis for encapsulating nio.
  - let you do lots of other control abstractions.
- Break (done in library)
- Trampolining tail calls (done in library)

```scala
import scala.util.control.Breaks._
breakable {
  for (x <- elems) {
    println(x * 2)
    if (x > 0) break
  }
}
```
Slipping break back into Scala.

```scala
package scala.util.control // Scala v2.8 only!

object Breaks {
    private class BreakException extends RuntimeException
    private val breakException = new BreakException

    /** A block from which one can exit with a `break` */
    def breakable(op: => Unit) {
        try {
            op
        } catch {
            case ex: BreakException =>
        }
    }

    /** Break from closest enclosing breakable block */
    def break { throw breakException }
}
```

--(Unix)** Breaks.scala.l All L14 (Scala)-----------------
More features

• Extended swing library.
• Performance improvements, among others for
  – structural type dispatch,
  – actors,
  – vectors, sets and maps.
• Standardized compiler plugin architecture.
How to get 2.8

Can use trunk today (bleeding edge).

Stable preview planned for July.

Final version planned for September or October.
Beyond 2.8
New challenges

Concurrency:

How to make concurrent activities play well together?

Parallelism:

How to extract better performance from today’s and tomorrow’s parallel processors (multicores, gpus, etc).

The two do not necessarily have the same solutions!
What can be done?

(1) **Better abstractions:**

- Actors
- Transactions
- Join patterns
- Stream processing with single-assignment variables
- Data parallelism
(2) Better Analyses. Distinguish between:

- **pure**: no side effects
- **thread-local**: no interaction
- **transactional**: atomic interaction
- **unrestricted access** (danger!)

Need *alias tracking* and *effect systems*.
Problem: can we make these lightweight enough?
(3) **Better support for static types.**

Need to overcome the “type wall”.

Static typing is a key ingredient for reliable concurrent programs, yet programs are annoyingly hard to get right.

Need better tools: Type browsers, type debuggers.

Need better presentations: Users might not always want to see the most general type!
Why Scala?

I believe Scala is well placed as a basis for concurrent and parallel programming, because

– It runs natively on the JVM.
– It supports functional programming.
– It has great syntactic flexibility.

• Btw we are not the only ones to have recognized this:
  – The X10, PPL, Fortress are all using Scala in some role or other.
Learning Scala

- To get started:
  - First steps in Scala, by Bill Venners published in Scalazine at www.artima.com
  - Scala for Java Refugees by Daniel Spiewack (great blog series)

- To continue:
  3+1 books published; more in the pipeline.
How to find out more

Track it on scala-lang.org:

Thank You