Introduction to Scala

Steve Roggenkamp
Perceptive Software, Inc.[1]
roggenkamps at acm.org

13 October 2012

[1] The views expressed in this presentation are my own and not Perceptive Software, Inc. or Lexmark, Inc.

Steve's Bio

- Programming professionally since about 1983
- Worked on wide variety of software
  - Engineering analysis, FEA modeling, 3D computer graphics
  - Graphical User Interfaces (before Windows!)
  - Industrial process control systems
  - Call Record processing for telecom
  - Unix kernel and systems programming
  - Large-scale document processing and transformation
  - Large-scale bibliographic transformation systems
  - Medical report generator for multiple languages
- Computer languages: BASIC, C, C++, Fortran, Groovy, Haskell, Lisp, Perl, PostScript, R, Scala, TEX, XML

Agenda

- Why Scala?
- The Scala Ecosystem
- What makes Scala sizzle?
- Resources

Why Scala?

- *SCA*lable *LA*nguage $=>$ *SCALA*
- Makes efficient use of resources
- Merges object-oriented and functional programming paradigms
- Works in the Java/JVM ecosystem
- Static type system
- Scales well
- Good performance
- Read-Evaluate-Print-Loop (REPL)
- XML integration

Makes efficient use of resources

- Concise language syntax - minimizes code to write (DRY)
- Well defined language specification
- Well defined standard API
- Very good performance
- Memory usage similar to Java

Merges object-oriented and functional programming techniques

- Organize problem solutions into classes/objects/methods
- Incorporates functional programming techniques
  - Higher-order functions
  - Closures and curried functions
  - Pattern matching

Works in the Java/JVM ecosystem

- Integrates seamlessly with Java
- Can use existing frameworks such as Spring
- Other JVM languages, Groovy, Clojure, etc. can use Scala libraries

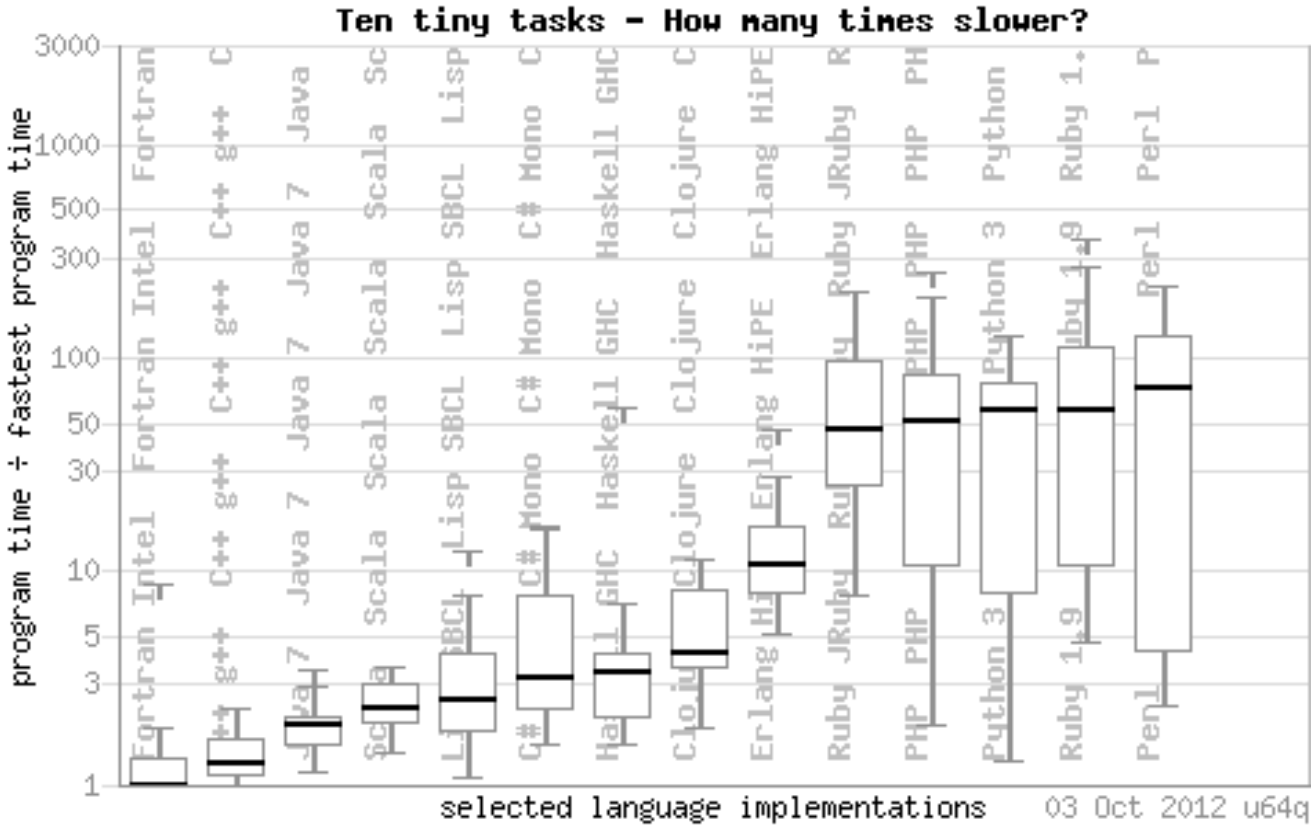
Static Type Checking System

- Each expression and value has a type
- Compiler checks types at compile time
- Eliminates most runtime type checks and errors
- Infers types from expressions in most situations

Scales Well

- Thread-safe immutable objects
- Provides Erlang Actor messaging
- Used by Twitter, LinkedIn, Sony, Foursquare and others for web-scale projects

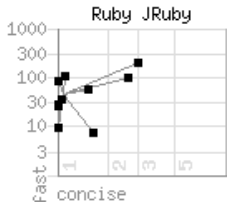
Good Performance

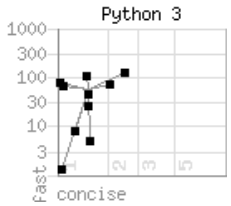


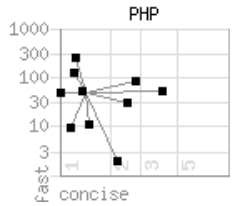
- From The Computer Language Benchmarks Game (http://shootout.alioth.debian.org/)
- Note logarithmic scale on program times

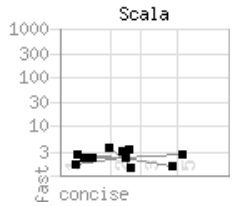
Good Performance 2

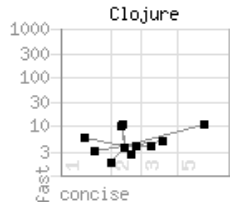
Slow

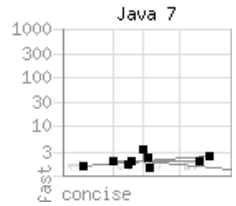












Fast

Concise Verbose

- From The Computer Language Benchmarks Game
- Logarithmic scale on both axes
- Conciseness based on gziped size of benchmark program sources.

Good Performance 3

- Remember Little's Law $N = A * T$
  - N – the number of items in a system
  - A – the average response time of the system
  - T – the arrival rate of items
- Reducing the average response time reduces the number of items in a system.
- Reducing the number of items reduces cost.

$$100 requests = 1.0 sec * 100 requests/sec$$

$$10 requests = 0.10 sec * 100 requests/sec$$

The Scala Software Ecosystem

- IDE Support
- Tools
- Frameworks

IDE Support

- Scala has plugins (supports) for
  - Eclipse
  - IntelliJ
  - NetBeans
- Typesafe provides a version of Eclipse with a worksheet

Scala Tools

- Simple Build Tool (sbt)
  - Build tool for scala
  - Uses Scala as a domain specific language to describe build process
  - Works with Maven and Ivy repositories to resolve build and run-time dependencies.
- Ensime for Emacs
  - Provides 80% of an IDE with Emacs editing capabilities

Scala Frameworks - Lift

- Lift Web provides an MVC-oriented framework for web applications
- Lift Web - applications are[2]
  - Secure – Lift apps are resistant to common vulnerabilities including many of the OWASP Top 10
  - Developer centric – Lift apps are fast to build, concise and easy to maintain
  - Designer friendly – Lift apps can be developed in a totally designer friendly way
  - Scalable – Lift apps are high performance and scale in the real world to handle insane traffic levels
  - Modular – Lift apps can benefit from, easy to integrate, pre-built modules
  - Interactive like a desktop app – Lift's Comet support is unparalleled and Lift's ajax support is super-easy and very secure

[2] From http://www.liftweb.net

Scala Frameworks - Akka

- Expands Actors model to distributed systems
- Event-driven model
- Can scale to very large systems
- Provides a form of transactions

Scala Frameworks - Play

- Provides a RESTful web application framework
- Based on MVC pattern.
- Uses Scala type system to provide high-performance type-checked templates
- RDBMS support

What makes Scala sizzle

- Scala works with existing Java and .NET code.
- There are no ”primitives”, every value is an Object.
- Every function is a value.
- Every value and expression have a type that can be statically checked.
- Scala can be extended with new operators.
- Implements both call-by-value and call-by-name conventions
- Pattern matching
- XML integration
- Actors for efficient concurrent objects

Hello World!

```
object HelloWorld { def main(args: Array[String]) = println( "Hello World!")}
```

or

```
object HelloWorld {
  def main(args: Array[String]) = println( "Hello World!")
}
```

or

```
object HelloWorld extends Application { println( "Hello World!") }
```

runs thus:

```
lthp01:code$
lthp01:code$ cat HelloWorld.scala
object HelloWorld { def main(args: Array[String]) = println( "Hello World!")}
lthp01:code$ scala HelloWorld.scala
Hello World!
lthp01:code$
```

Every value is an Object

- No “primitives”
- All operations are method calls

  `1 + 2`

  is implemented as

  `(1).+(2)`
- Syntactic “sugar” makes this bearable

Singleton Objects

- Scala does not provide “static” objects for classes
- It provides singleton objects

Singleton Object Example

```
object HelloWorld extends Application {
  val h = "Hello World!"
  def p( s : String ) = println(s )
  p(h)
}
```

Every function is a value

- Scala is a functional programming language
- Scala supports closures and curried functions
- Functions can contain functions

Closures

```
scala> var factor=4
factor: Int = 4

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

scala> def multiply( x: Int ) = x * factor
multiply: (x: Int)Int

scala> multiply( 3 )
res0: Int = 12

scala> l.map( multiply )
res1: List[Int] = List(4, 8, 12)

scala> factor=5
factor: Int = 5

scala> l.map( multiply )
res2: List[Int] = List(5, 10, 15)
```

Curried Functions

```
scala> def mult(x: Int) (y: Int) = x * y
mult: (x: Int)(y: Int)Int

scala> l.map( mult(4))
res4: List[Int] = List(4, 8, 12)

scala> l.map( mult(factor))
res5: List[Int] = List(5, 10, 15)

scala> def cmult(x:Int) (y:Int) = x * y * factor
cmult: (x: Int)(y: Int)Int

scala> l.map(cmult(3))
res6: List[Int] = List(15, 30, 45)
```

More fun with functions

```
scala> def f( i: Int ) = i + 7
f: (i: Int)Int

scala> def g( i: Int ) = i  * 3
g: (i: Int)Int

scala> def h( m: Int => Int, n : Int => Int, i : Int ) = m(n(i))
h: (m: Int => Int, n: Int => Int, i: Int)Int

scala> h(f, g, 5)
res1: Int = 22

scala> h(g, f, 5)
res2: Int = 36

scala> h( x => x + 22, g, 4)
res3: Int = 34

scala> h( x => x * 22, g, 4)
res4: Int = 264

scala> h( x => x * 22, x => x * 8, 4)
res5: Int = 704
```

Scala implements static type checking

- Every expression has a type
- Eliminates many of run-time errors and checks
- Can infer the type of many expressions

Scala can be extended with new operators

- Scala allows flexible names for methods (operators)
- Makes it easy to create new domain specific languages

New operators, cont

```
case class WeirdInt( i: Int ) {
  def ?+ (j: WeirdInt) = WeirdInt( i + j.i )
  def ?- (j: WeirdInt) = WeirdInt( i - j.i )
  def ?* (j: WeirdInt) = WeirdInt( i * j.i )
  def ?/ (j: WeirdInt) = WeirdInt( i / j.i )
  def toInt = i
}
--
scala> :load WeirdInt.scala
Loading WeirdInt.scala...
defined class WeirdInt
scala> val z3=WeirdInt(3)
z3: WeirdInt = WeirdInt(3)
scala> val z5=WeirdInt(5)
z5: WeirdInt = WeirdInt(5)
scala> z3 ?* z5
res0: WeirdInt = WeirdInt(15)
scala> ( z3 ?* z5 ) + z5
<console>:12: error: type mismatch;
 found   : WeirdInt
 required: String
              ( z3 ?* z5 ) + z5
                           ^
scala> ( z3 ?* z5 ) ?+ z5
res2: WeirdInt = WeirdInt(20)
scala> z3 ?* z5 ?+ z5 toInt
res3: Int = 20
scala> (z3.?*(z5)).?+( z5).toInt
res4: Int = 20
```

Implements both call-by-value and call-by-name conventions

- Call-by-value: evaluate function parameters, then call function
- Call-by-name: call function, let it decide whether it needs to evaluate its parameters

Call By Value

```
scala> def cbv( a: Int, b: Int ) = if ( a < 5 ) a else b
cbv: (a: Int, b: Int)Int
scala> cbv( 3, 10 )
res14: Int = 3
scala> cbv( 7, 10 )
res15: Int = 10
scala> cbv( 3, 10/0 )
java.lang.ArithmeticException: / by zero
...
```

Call By Name

```
scala> def cbv( a: Int, b: => Int ) = if ( a < 5 ) a else b
cbv: (a: Int, b: Int)Int
scala> cbv( 3, 10 )
res14: Int = 3
scala> cbv( 7, 10 )
res15: Int = 10
scala> cbv( 3, 10/0 )
res16: Int = 3
scala> cbv( 7, 10/0 )
java.lang.ArithmeticException: / by zero
...
```

Classes and Traits

- Scala has single inheritance, just like Java
- Scala provides 'traits' to provide composition
- Traits similar to Java's interfaces, but more powerful
- Traits can include code, unlike Java's interfaces
- Classes provide the model for how to create objects
- Classes can be parametrized instead of explicit constructors
- Case classes provide syntactic sugar to
  - provide factory methods
  - implicitly make vals of fields
  - permit pattern matching expressions on the class

Classes and Traits example

```
package examples
case class Person( firstName: String, lastName: String, age: Int )
trait USCitizen {
  private val myRep = Person( "Myke", "Representative", 45 )
  def representative = myRep
}
trait Senior { def ssOffice = "Unknown" }
case class USPerson( override val firstName: String,
     override val lastName:  String,
     override val age:       Int,
                  ssn:       String )
     extends Person( firstName, lastName, age) with USCitizen
case class USSenior( override val firstName: String,
     override val lastName:  String,
     override val age:       Int,
     override val ssn:       String )
     extends USPerson( firstName, lastName, age, ssn )
         with Senior
object ClassExample2 extends Application {
  val sally = Person( "Sally", "Simon", 20 )
  val billy = USPerson( "Billy", "Redmond", 30, "123-45-6789" )
  val mark  = USSenior( "Mark", "Miraldi", 66, "789-01-2345" )
  println( sally )
  println( billy )
  println( billy representative )
  println( mark )
  println( mark representative )
  println( mark ssOffice )
}
```

Classes and Traits example execution

```
lthp01:code$ scala examples.ClassExample2
Person(Sally,Simon,20)
USPerson(Billy,Redmond,30,123-45-6789)
Person(Myke,Representative,45)
USSenior(Mark,Miraldi,66,789-01-2345)
Person(Myke,Representative,45)
Unknown
lthp01:code$
```

Pattern Matching - Classes

```
// from from http://www.scala-lang.org/node/52
package examples

object patterns {
  abstract class Tree
  case class Branch(left: Tree, right: Tree) extends Tree
  case class Leaf(x: Int) extends Tree

  val tree1 = Branch(Branch(Leaf(1), Leaf(2)), Branch(Leaf(3), Leaf(4)))

  def sumLeaves(t: Tree): Int = t match {
    case Branch(l, r) => sumLeaves(l) + sumLeaves(r)
    case Leaf(x) => x
  }

  def toString(t: Tree): String = t match {
    case Branch(l, r) => "Branch(" + toString(l) + "," + toString(r) + ")"
    case Leaf(x)      => "Leaf(" + x + ")"
    }

  def main(args: Array[String]) {
    println("sum of leafs=" + sumLeaves(tree1))
    println("Tree: " + toString(tree1))
  }
}
Output:
lthp01:code$ scala examples.patterns
sum of leafs=10
Tree: Branch(Branch(Leaf(1),Leaf(2)),Branch(Leaf(3),Leaf(4)))
lthp01:code$
```

Scala and XML

- Scala integrates XML into the language
- XML trees are just expressions
- Provides operators to search XML trees much like XPath
- Scala provides pattern matching for XML

XML processing

```
case class Person( firstName: String, lastName: String, age: Int )

val sallyX =
<Person>
<FName>Sally</FName>
<LName>Simon</LName>
<Age>20</Age>
</Person>

def XMLtoPerson( node: scala.xml.Node ) : Person =
   Person((node \  "FName").text,
  (node \\ "LName").text,
  (node \  "Age").text.toInt)

def personToXML(fname: String,
                lname: String,
       age:    Int ) =
<Person>
<FName>{fname}</FName>
<LName>{lname}</LName>
<Age>{age}</Age>
</Person>

val sally = XMLtoPerson( sallyX )

val sallyX1 = personToXML( sally.firstName, sally.lastName, sally.age )
```

XML code output

```
scala> :load xml-1.scala
Loading xml-1.scala...
defined class Person
sallyX: scala.xml.Elem =
<Person>
<FName>Sally</FName>
<LName>Simon</LName>
<Age>20</Age>
</Person>
XMLtoPerson: (node: scala.xml.Node)Person
personToXML: (fname: String, lname: String, age: Int)scala.xml.Elem
sally: Person = Person(Sally,Simon,20)
sallyX1: scala.xml.Elem =
<Person>
<FName>Sally</FName>
<LName>Simon</LName>
<Age>20</Age>
</Person>

scala>
```

Actors

- Actors provide a concurrency model
- Actors originated with the Erlang language
- An Actor provides a thread-like abstraction with a mailbox for messages
- Actors do not block, they run to completion
- Scala provides two functions for receiving messages:
  - `receive` - receive a message and return a result
  - `react` - receive a message and do not return a result
- `react` can be much more efficient than `receive` since it doesn't have to return a result
- `react` reuses threads to save setup/tear down time

Actors (Ping Pong)

From http://www.scala-lang.org/node/54

```
package examples.actors
import scala.actors.Actor
import scala.actors.Actor._

abstract class PingMessage
case object Start extends PingMessage
case object SendPing extends PingMessage
case object Pong extends PingMessage
abstract class PongMessage
case object Ping extends PongMessage
case object Stop extends PongMessage

object pingpong extends Application {
  val pong = new Pong
  val ping = new Ping(100000, pong)
  ping.start
  pong.start
  ping ! Start
}
```

Actors (Ping)

```
class Ping(count: Int, pong: Actor) extends Actor {
  def act() {
    println("Ping: Initializing with count "+count+": "+pong)
    var pingsLeft = count
    loop {
      react {
        case Start =>
          println("Ping: starting.")
          pong ! Ping
          pingsLeft = pingsLeft - 1
        case SendPing =>
          pong ! Ping
          pingsLeft = pingsLeft - 1
        case Pong =>
          if (pingsLeft % 1000 == 0)
            println("Ping: pong from: "+sender)
          if (pingsLeft > 0)
            self ! SendPing
          else {
            println("Ping: Stop.")
            pong ! Stop
            exit('stop)
          }
      }
    }
  }
}
```

Actors, (Pong)

```
class Pong extends Actor {
  def act() {
    var pongCount = 0
    loop {
      react {
        case Ping =>
          if (pongCount % 1000 == 0)
            println("Pong: ping "+pongCount+" from "+sender)
          sender ! Pong
          pongCount = pongCount + 1
        case Stop =>
          println("Pong: Stop.")
          exit('stop)
      }
    }
  }
}
```

```
lthp01:code$ scala examples.actors.pingpong
Ping: Initializing with count 10000: examples.actors.Pong@1f4384c2
Ping: starting.
Pong: ping 0 from examples.actors.Ping@19484a05
Ping: pong from: examples.actors.Pong@1f4384c2
Pong: ping 1000 from examples.actors.Ping@19484a05
Ping: pong from: examples.actors.Pong@1f4384c2
Pong: ping 2000 from examples.actors.Ping@19484a05
Ping: pong from: examples.actors.Pong@1f4384c2
Pong: ping 3000 from examples.actors.Ping@19484a05
Ping: pong from: examples.actors.Pong@1f4384c2
Pong: ping 4000 from examples.actors.Ping@19484a05
Ping: pong from: examples.actors.Pong@1f4384c2
Pong: ping 5000 from examples.actors.Ping@19484a05
Ping: pong from: examples.actors.Pong@1f4384c2
Pong: ping 6000 from examples.actors.Ping@19484a05
Ping: pong from: examples.actors.Pong@1f4384c2
Pong: ping 7000 from examples.actors.Ping@19484a05
Ping: pong from: examples.actors.Pong@1f4384c2
Pong: ping 8000 from examples.actors.Ping@19484a05
Ping: pong from: examples.actors.Pong@1f4384c2
Pong: ping 9000 from examples.actors.Ping@19484a05
Ping: pong from: examples.actors.Pong@1f4384c2
Ping: Stop.
Pong: Stop.
lthp01:code$
```

Resources

Web sites

http://www.scala-lang.org – official Scala site
http://www.scala-lang.org/api/current – Scala API Documentation
http://shootout.alioth.debian.org – Computer language shootout

Books

Programming in Scala, Second Edition; Odersky, Spoon, Venners
Programming Scala; Wampler, Payne; http://ofps.oreilly.com/titles/9780596155957/index.html

Tutorials and presentations

http://www.slideshare.net/Odersky/fosdem-2009-1013261 – Scala - A scalable language
http://www.slideshare.net/michael.galpin/introduction-to-scala-for-java-developers-presentation – Scala for Java Developers

Questions?