

A Brief Introduction to Scala for Java Developers

Miles Sabin, Chuusai Ltd.
<http://www.chuusai.com/>

<http://uk.linkedin.com/in/milessabin>
<http://twitter.com/milessabin>

Outline

- Background
- Object–Oriented Features
- Functional Features
- Selected Highlights
- Tools and Frameworks
- Roadmap

Background

- Designed by Martin Odersky (Pizza, GJ and Java 5) of EPFL as the successor to Funnel
- Objectives,
 - Blend object-oriented and functional styles
 - Seamless interoperability with Java (and .Net?)
- Project started in 2003
 - First public release in 2004
 - Currently at 2.7.7
 - 2.8 release due very soon

Scala is the Future of Java

- Scala can be thought of as a superset of current Java
 - It has all the object-oriented features of current Java (some in a slightly different and improved form)
 - It already has many of the most desirable proposed extensions to Java (eg. true closures)
- Nevertheless, it compiles to Java bytecode
 - Flawless interoperability with Java, leverages the mature Hotspot JIT

Interoperability with Java

- There are many alternative JVM languages. Some also strongly “Java-compatible”
 - Close source and binary mapping to Java
 - Scala, Groovy, JavaFX, AspectJ
- This holds out the prospect that most Java tools, libraries and frameworks will Just Work, or work with only minor adaptation
- Additional promise of gradual migration of existing Java projects

Who's Using It?

- A rapidly growing list of companies and projects,
 - Twitter (infrastructure)
 - LinkedIn (analytics, infrastructure)
 - EDF Trading (analytics, infrastructure)
 - Sony Pictures Imageworks (infrastructure)
 - SAP/Siemens (ESME, enterprise messaging)
 - Novell (Pulse, enterprise messaging)

Why Mix OO and FP?

- Each has complementary strengths —
 - OO provides excellent support for,
 - Subtyping and inheritance
 - Modular components
 - Classes as partial abstractions
 - FP provides excellent support for,
 - Higher order functions
 - ADTs, structural recursion and pattern matching
 - Parametric polymorphism
- They've been converging for a while now

Scala has a REPL

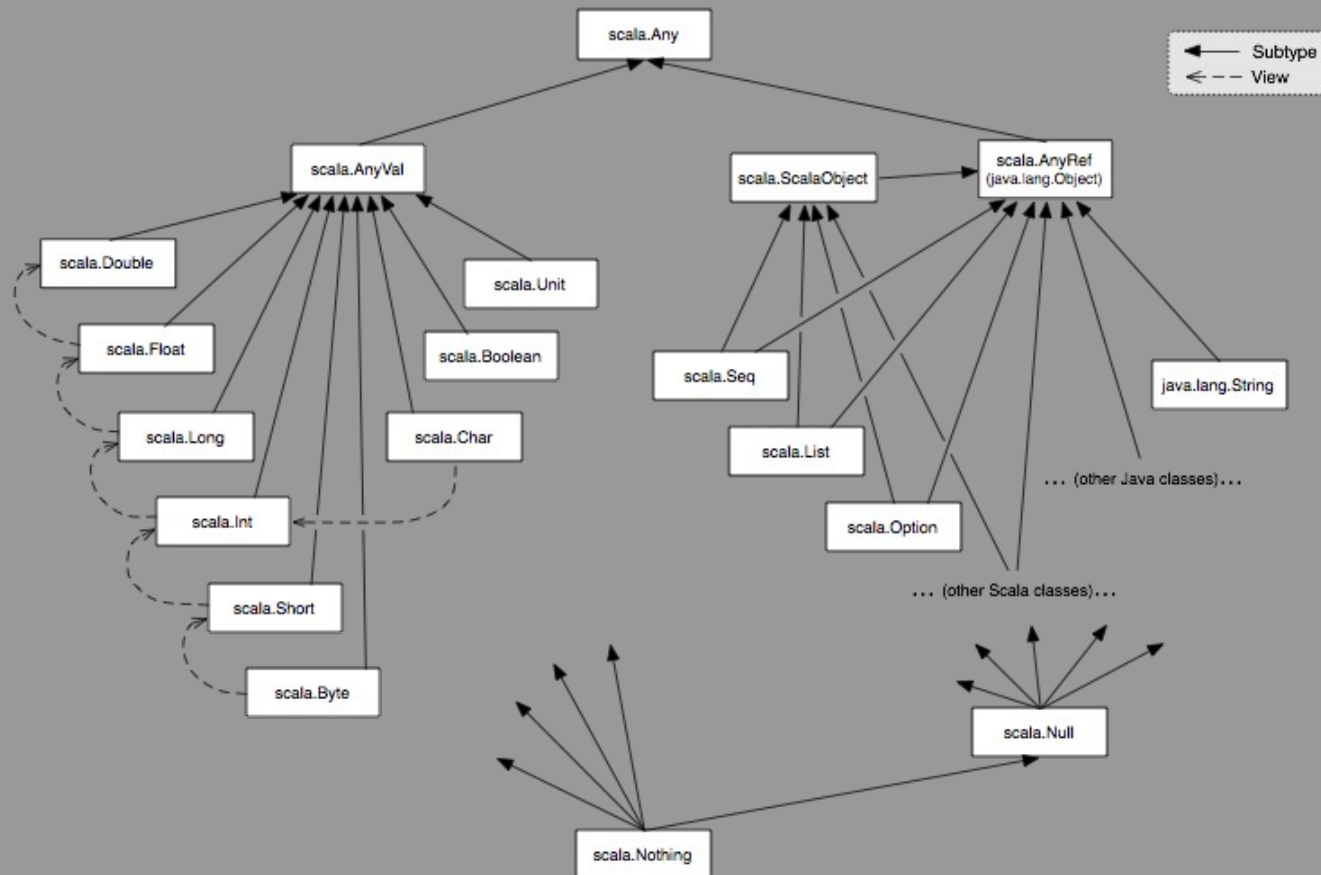
- Common in scripting and functional languages
- Great for exploratory programming
- We'll be using it as we go along

Scala Cleans Up Java Syntax

- Semi-colons are optional, equals is ==
- All statements are expressions and have a value
- Type inference eliminates the most annoying explicit typing annotations
- Case-classes have minimal boilerplate
- Closures and by name arguments allow libraries to define specialized control structures

Scala is Object Oriented

- Scala has a uniform object model: no primitive types



Scala is Object Oriented

- Operators aren't special, they're methods like any other,

```
val x = 23
x + 1 // equivalent to x.+(1)
```

- Similarly, methods can be invoked in operator form,

```
val s = "Hello world"
s length // equivalent to s.length
s contains "world" // equivalent to s.contains("world")
```

No Statics

- No static fields or methods
- Instead we have first-class "object" entities similar to singletons and ML modules

```
// Java
public class User {
    public static newUser(String name, String pass) {
        return new User(name, pass);
    }
    // Non-static methods ...
}
// Scala
object User {
    def apply(name : String, pass : String) =
        new User(name, pass)
}
val joe = User("Joe Bloggs", "<secret>")
```

Vals, Vars and Uniform Access

- Scala makes immutable definitions easy,

```
val s : String = "Hello immutable world"
s = "Bye ..." // Error
var t : String = "Hello mutable world"
t = "Bye ..." // OK
```

- (Im)mutable properties can be implemented by vals, vars or methods

```
class User {
  val name : String      // Immutable
  var email : String     // Mutable
  def pass : String      // Computed
  def pass_=(s : String) //
}
```

```
user.pass = "<secret>" // Sugar for user.pass_("<secret>")
```

Statically Typed with Inference

- All definitions must have a static type

```
val m : Map[Int, String]
```

- However these types are typically inferred

```
scala> val m = Map(1 -> "one", 2 -> "two")  
m : Map[Int, String] = Map(1 -> one, 2 -> two)
```

- Method return types are also inferred

```
scala> def twice(i : Int) = i*2  
twice: (i: Int)Int
```

Argument types must be explicit, also the return types for recursive functions

Named and Default Arguments

- Arguments can be specified at call sites by name

```
def coord(x : Double, y : Double)
  coord(y = 1.0, x = 0.7)
```

Allows them to be provided naturally for the call site, and eliminates ambiguity

- Arguments can also be given default values

```
def printList(l : List[Int], sep : String = ", ") { ... }

val l = List(1, 2, 3)
printList(l) // equivalent to printList(l, ", ")
```

Case Classes are Lightweight

- Eliminate a lot of the boilerplate associated with Java implementations of simple data types

```
public class User {
    private final String name;
    private final String pass;
    public User(String name_, String pass_) {
        name = name_;
        pass = pass_;
    }
    public boolean equals(Object other) {
        // Stuff ...
    }
    public int hashCode() {
        // More stuff ...
    }
}
```

Case Classes are Lightweight

- The Scala equivalent

```
case class User(name : String, pass : String)
```

- Accessors, equals, hashCode and toString are provided automatically
- “new” is optional

```
val joe = User("Joe Bloggs", "<secret>")
```

- “Copy with changes” supports immutable functional objects

```
val joeUpdated = joe.copy(pass = "<still secret>")
```

Pattern Matching

- Case classes model ADTs from functional languages and support pattern matching

```
sealed trait Tree[T]
case class Leaf[T](elem : T) extends Tree[T]
case class Node[T](left : Tree[T], right : Tree[T])

val t = Node(Node(Leaf("bar"), Leaf("baz")), Leaf("foo"))

def find[T](tree : Tree[T], elem : T) : Boolean =
  tree match {
    case Node(l, r)    => find(l, elem) || find(r, elem)
    case Leaf(`elem`) => true
    case _            => false
  }
```

- Matching is the inverse of construction

The Option Type

- “Null References: The Billion Dollar Mistake” — Tony Hoare
- Scala provides a safe alternative

```
scala> List(1, 2, 3) find (_ == 2)
res0: Option[Int] = Some(2)
```

```
scala> List(1, 2, 3) find (_ == 4)
res0: Option[Int] = None
```

- Option interacts nicely with matching

```
List(1, 2, 3) find (_ == 2) match {
  case Some(i) => println("Found "+i)
  case None => println("Not found")
}
```

Tuples

- Scala has tuple types and literals

```
val coord : (Double, Double) = (1.0, 0.5)
println("x = "+coord._1+", y =" +coord._2)
```

- These are first-class types like any other

```
val coords = new ListBuffer[(Double, Double)]
coords += coord
```

- Provide ad hoc grouping and multiple return values

```
def firstWord(s : String) = {
  val i = s.indexOf(' ')
  (s.substring(0, i), s.substring(i, s.length))
}
val (first, rest) = firstWord("The quick brown fox ...")
```

Mixin Composition

- Java interfaces are replaced by traits and mixin composition
 - Traits can provide method implementations
 - Traits can provide fields

```
trait UserId {  
  val name : String  
  var pass : String = "change me"  
  def display = name+": "+pass  
}  
class User(val name : String) extends UserId  
  
val joe = new User("Joe Bloggs")  
println(joe.display)
```

Mixin Composition

- Traits support multiple inheritance whilst avoiding the problems of “diamond” inheritance

```
trait Email {
  val address : String
  def send(message : String {
    // Concrete implementation
  })
}
class User(val name : String, val address : String)
  extends UserId with Email

val joe = new User("Joe Bloggs", "joe@gmail.com")
println(joe.display)
joe.send("Don't forget to change your password!")
```

Laziness and By-Name Args

- Values can be declared to be initialized lazily

```
val (title, givenName, surname) = ("Mr.", "John", "Smith")  
lazy val fullName = title+" "+givenName+" "+surname
```

The value is computed the first time it is used (if at all)

```
val displayName = if (full) fullName else givenName
```

- Can be used to create circular structures

```
abstract class Link { val next : Link }  
val (a : Link, b : Link) =  
  (new Link { lazy val next = b },  
   new Link { lazy val next = a })
```

Laziness and By-Name Args

- Arguments can be passed by name
 - Similar to laziness in that evaluation is deferred until use
 - Can be used to build specialized control structures and enables internal DSLs

```
def locked[T](l : Lock)(op : => T) = {  
  l.lock  
  try { op } finally { l.unlock }  
}  
val lock = new ReentrantLock  
var shared = ...  
locked(lock) {  
  /* Use shared while holding lock */  
}
```

Structural Typing

- Scala has a form of statically checkable duck-typing
- We can use this to generalize libraries to pre-existing types

```
type Closeable = { def close() }
```

```
def using[R <: Closeable, T](res : R)(op : R => T) =  
  try { op(res) } finally { res.close() }
```

```
val b = using(new FileInputStream("test.txt")) { _.read }
```

Implicits

- Implicit functions provide a way to attach new behaviours to existing types
- Invoked automatically if needed to satisfy the typechecker

```
trait Closeable { def close : Unit }

def using[R <% Closeable, T](res : R)(op : R => T) =
  try { op(res) } finally { res.close() }

implicit def InputStreamIsCloseable(is : InputStream) =
  new Closeable { def close = in.close }

val b = using(new FileInputStream("test.txt")) { _.read }
```

First-Class Functions

- Scala allows the definition of functions

```
def plusOne(x : Int) = x+1
```

- Functions can be arguments and results

```
def applyTwice(x : Int, f : Int => Int) = f(f(x))  
applyTwice(3, plusOne) // == 5
```

- Can be anonymous and close over their environment

```
def twice(f : Int => Int) = (x : Int) => f(f(x))  
twice(plusOne)(3) // == 5
```

- Function literal syntax is concise

```
twice(_+1)(3) // == 5
```

Higher-Order Functions

- Higher-order functions are used extensively in Scala's standard library

```
List(1, 2, 3).map(_*2) // == List(2, 4, 6)
```

```
List(1, 2, 3, 4).find(_%2 == 0) // Some(2)
```

```
List(1, 2, 3, 4).filter(_%2 == 0) // List(2, 4)
```

```
def recip(x : Int) = if(x == 0) None else Some(1.0/x)
```

```
scala> List(0, 1, 2, 3).flatMap(recip)
```

```
res0: List[Int] = List(1.0, 0.5, 0.3333333333333333)
```

For Comprehensions

- Scala's “for comprehensions” capture common patterns of use of map, flatMap and filter

```
val l1 = List(0, 1)
val l2 = List(2, 3)
```

```
scala> for (x <- l1; y <- l2) yield (x, y)
res0: List[(Int, Int)] = List((0,2), (0,3), (1,2), (1,3))
```

The for expression desugars to,

```
l1.flatMap(x => l2.map(y => (x, y)))
```

- These patterns are the monad laws for the subject type

For Comprehensions

- Option implements map, flatMap and Filter so works nicely with for comprehensions

```
def goodPair(x : Int, y : Int) =  
  for(fst <- recip(x); snd <- recip(y))  
    yield (x, y)
```

```
scala> goodPair(1, 2)  
res0: Option[(Int, Int)] = Some((1,2))
```

```
scala> goodPair(1, 0)  
res0: Option[(Int, Int)] = None
```

- Using Option rather than null has clear benefits when used in this way

Language-Level XML Support

- Scala has language level support for XML via XML literals, XPath and pattern matching,

```
val book =  
  <book>  
    <author>Martin Odersky</author>  
    <title>Programming in Scala</title>  
  </book>
```

```
val title = book \\ "title" text // == "Programming in ..."
```

```
for (elem <- book.child) elem match {  
  case <author>{name}</author> => println(name)  
  case _ =>  
} // Prints "Martin Odersky"
```

Actor-based Concurrency

- Scala has library level support for actor-based, message-passing concurrency
 - An embarrassment of riches —
 - `scala.actors`
 - Lift actors
 - Akka
- Not obligatory, these are all libraries, and all the traditional Java approaches to concurrency are available

Tools and Frameworks

- Most Java tools and frameworks work with Scala Out Of The Box
- Testing frameworks —
 - Specs, ScalaTest, ScalaCheck
- Web frameworks —
 - Lift
 - Wicket and Play recently added Scala support
- IDE support — the big three (Eclipse, Netbeans, IDEA) all actively developed

Roadmap

- Upcoming releases,
 - Beta of 2.8 now available
 - First 2.8 Release Candidate expected in March
 - 2.8 Final expected June/July
- Subsequent 2.8-series releases will continue current exploratory work,
 - Type specialization (still needs library support)
 - Continuations (currently a compiler plugin)
 - Linear/immutable/non-null types

Find Out More

- Scala's home at EPFL
<http://www.scala-lang.org>
 - See also the scala and scala-user mailing list
- The London Scala Users' Group
<http://www.meetup.com/london-scala>
- Publications
 - Programming in Scala
Odersky, Venners and Spoon
 - Programming in Scala
Wampler and Payne

A Brief Introduction to Scala for Java Developers

Miles Sabin, Chuusai Ltd.
<http://www.chuusai.com/>

<http://uk.linkedin.com/in/milessabin>
<http://twitter.com/milessabin>