Scala *for Java Devs*

Christos KK Loverdos

loverdos at github, twitter, gmail

-

Java meetup @ Thessaloniki

2017-10-12

About me

- Software engineer, nearly 20 years now
  - Architect, team lead, technical PM
  - Telcos, startups
  - employee, freelancer, consultant
- Java enthusiast since 1997
- Scala enthusiast since 2004 (2007)
- Co-author of "Steps in Scala"

A very incomplete timeline

- 2004 v1.0
- 2006 v2.0
- **2007** v2.7 (+**Lift** web framework)
- 2010 v2.8 (+**ScalaDays** @ EPFL, 180 participants)
- 2011 v2.9
- 2012 v2.10
- now v2.12 (+**Java 8** interoperability)

Main ideas - Why Scala?

- Synthesis of OOP and FP
- Being **scalable**
    - Programming in the small vs programming in the large
    - Provide the right abstractions in the core, everything else in libraries
- Rich type system

A few highlights

- **Immutable** objects and collections
- **Type** inference
- **Functions** are first-class
- Everything is an **object**
- Pattern matching
- Domain Specific Languages
- **REPL**

Warm-up

Hello world

`hello.scala`

```scala
object hello {
  def main(args: Array[String]): Unit =
    println("Hello world")
}
```

Hello world

`hello.scala`

```scala
object hello {
  def main(args: Array[String]): Unit =
    println("Hello world")
}
```

```
$ scala -save hello.scala
Hello world
```

Hello world

`hello.scala`

```scala
object hello {
  def main(args: Array[String]): Unit =
    println("Hello world")
}
```

```
$ scala -save hello.scala
Hello world

$ ls hello.*
hello.jar hello.scala
```

Hello world Scala vs Java

```
object hello {}
```

```
public class hello {}
```

- **object**, **class**
- No **public**

Hello world Scala vs Java

```
def main(...): Unit = {}
```

```
public static void main(...) {}
```

- **Unit** vs **void**
- No **static**
- Return type **after** vs **before** method name & args
- Scala **object** implies Java **static**

Hello world Scala vs Java

```
args: Array[String]
```

```
String[] args
```

- Type **after** vs **before** the name

Hello world Scala vs Java

```
args: Array[String]
```

```
String[] args
```

- Type **after** vs **before** the name

```
DIM X AS INTEGER
```

```
VAR X: Integer
```

REPL: a value is an object

```
evilvte
 ▾  scala
Welcome to Scala 2.12.3 (OpenJDK 64-Bit Server VM, Java 1.8.0_131).
Type in expressions for evaluation. Or try :help.

scala> 1.
!=   >             floatValue      isValidInt     to               toRadians
%    >=            floor           isValidLong    toBinaryString   toShort
&    >>            getClass        isValidShort   toByte           unary_+
*    >>>           intValue        isWhole        toChar           unary_-
+    ^             isInfinite      longValue      toDegrees        unary_~
-    abs           isInfinity      max            toDouble         underlying
/    byteValue     isNaN           min            toFloat          until
<    ceil          isNegInfinity   round          toHexString      |
<<   compare       isPosInfinity   self           toInt
<=   compareTo     isValidByte     shortValue     toLong
==   doubleValue   isValidChar     signum         toOctalString

scala> 1.doubleValue
res0: Double = 1.0

scala> 
```

A simple class

```scala
class Complex(val re: Double, val im: Double) {
  def this() = this(0, 0)

  def mag = Math.sqrt(re*re + im*im)

  def polarCoordinates = {
    val r   = Math.sqrt(re*re + im*im)
    val phi = Math.atan2(im, re)

    (r, phi)
  }

  override def toString = s"Complex($re, $im)"
}
```

A simple class

```
class Complex(val re: Double, val im: Double) {
  def this() = this(0, 0)

  def mag: Double = Math.sqrt(re*re + im*im)

  def polarCoordinates: (Double, Double) = {
    val r   = Math.sqrt(re*re + im*im)
    val phi = Math.atan2(im, re)

    (r, phi)
  }

  override def toString: String = s"Complex($re, $im)"
}
```

Super constructor

```
class Animal(name: String)

class Dog(name: String) extends Animal(name)
```

Some more fun

Trait (interface)

```scala
trait Printer {
  def printPDF(pdf: File): Unit

  def printRTF(rtf: File): Unit = {
    val pdf = ... // convert RTF to PDF
    printPDF(pdf)
  }
}
```

- Using default methods (https://github.com/scala/scala/pull/5003) as of Scala 2.12

Trait (mixin)

```scala
trait Mage {
  def castSpell(spell: Spell, target: Target) = ...
}
trait Fighter {
  def useSword(target: Target) = ...
}

class Player1 extends Mage
class Player2 extends Mage with Fighter
```

Case class

```scala
case class Complex(re: Double, im: Double)
```

Case class

```
case class Complex(re: Double, im: Double)

val c  = new Complex(1.2, 2.0)
val cc =     Complex(1.2, 2.0)
```

- No need to use **new**

Case class

```scala
case class Complex(re: Double, im: Double)

val c  = new Complex(1.2, 2.0)
val cc =     Complex(1.2, 2.0)

// (c == cc) is true
```

- Automatic, derived, structural equality
- Compiler implements `hashCode` and `equals`
- Very handy for immutable domain objects

Case class

```scala
case class Complex(re: Double, im: Double)

// instead of
// case class Complex(val re: Double, val im: Double)
```

- Constructor arguments are promoted to class attributes

Case class

```scala
case class Complex(re: Double, im: Double)

val zero = Complex(0, 0)
val one  = zero.copy(re = 1)
val i    = zero.copy(im = 1)
```

- Builtin `copy()`

Case class - Pattern matching

```scala
case class Complex(re: Double, im: Double)
```

```scala
val c: Complex = ...
```

```scala
c match {
  case Complex(0, 0) => // zero
  case Complex(1, 0) => // real unit
  case Complex(0, 1) => // imaginary unit
  case Complex(a, b) if a == b => // equal coordinates
  case Complex(a, b) => // all other cases ...
}
```

Case class - ASTs

`ast.scala`

```scala
sealed trait AstNode
case class Expr(n: Number) extends AstNode
case class Add(left: Expr, right: Expr) extends AstNode
```

`other.scala`

```scala
val x: AstNode = parseSourceCode(...)
x match {
  case Expr(n) => ...
  case Add(left, right) => ...
}
```

Generics

- type parameter in a class

```
trait Ordered[T] {
  def < (that: T): Boolean
  def <=(that: T): Boolean
  ...
}

class ANumber(x: Int) extends Ordered[ANumber] { ... }
```

`Ordered[T]` characterizes a type `T` that has a single, natural ordering.

Generics

- Collections

```scala
final class Array[T] // mutable
sealed abstract class List[T] // immutable

val a = Array(1, 2, 3)
val b = List(1, 2, 3)
val c = 1 :: 2 :: 3
```

Java's `Collection<T>` becomes `Collection[T]`

digression: List again

digression: List again

```
sealed abstract class List[+A] extends AbstractSeq[A]
with LinearSeq[A]
with Product
with GenericTraversableTemplate[A, List]
with LinearSeqOptimized[A, List[A]]
with Serializable
```

Generics (method)

- Parametric polymorphism

```
object Sorter {
  def quickSort[T](elems: Array[T], /*what else?*/)
}
```

Now that we've talked about generics, let's talk about functions

Function (definition)

```scala
trait Function1[-A, +B] extends AnyRef {
  def apply(a: A): B
}
```

Function (definition)

```scala
trait Function1[A, B] {
  def apply(a: A): B
}
```

We can also represent the type `Function1[A, B]` as

- `A => B` or
- `(A) => B`

This is a **total** function

Function

Math:

$$f : A \to B$$

Scala:

```
f: A => B
f: Function1[A, B]
```

Java:

```
Function1<A, B> f
```

Function (declaration)

```
val s_length:
  (String) => Int =
  (s: String) => s.length
```

Notice how the method `length` of class `String` was promoted to a function.

Function (declaration)

```
val s_length: (String) => Int = s => s.length

// or

val s_length: (String) => Int = _.length
```

With some nice syntactic sugar

Function (application)

We **apply** a function `f` of type `A => B`

```
val f: A => B = ...
```

to a value of type `A`

```
val x: A = ...
```

using intuitive syntax

```
f(x)
```

Function (application)

... which gets desugared to

```
f.apply(x)
```

Remember that `Function1` is defined with one method:

```
trait Function1[A, B] {
  def apply(a: A): B
}
```

Function (application)

… which gets desugared to

```
f.apply(x)
```

Everything is an object

Function (application)

`apply` is a binary method and you can also write

```
f apply x
```

Same thing for computing e.g. the maximum of two integers

`x max y` vs `x.max(y)`

Collections + Functions = fun

map

- Map each element of a collection to a new element, according to a well defined function

```
trait Collection[A] {
  def map[B](f: A => B): Collection[B]
}
```

map

- Map each element of a collection to a new element, according to a well defined function

```
trait Collection[A] {
  def map[B](f: A => B): Collection[B]
}
```

```
scala> List("a", "bb").map(x => x.length)
res2: List[Int] = List(1, 2)
```

map

- Map each element of a collection to a new element, according to a well defined function

```
trait Collection[A] {
  def map[B](f: A => B): Collection[B]
}
```

```
scala> List(1, 2, 3).map( n => isOdd(n) )
res3: List[Boolean] = List(true, false, true)
```

map

- Map each element of a collection to a new element, according to a well defined function

```
trait Collection[A] {
  def map[B](f: A => B): Collection[B]
}
```

```
alist.map(x => isOdd(x))

alist.map( isOdd(_) )
alist.map( isOdd  )
alist map  isOdd
```

... from map to Map

- The generic type is `Map[A, B]`
- The default implementation is immutable

```
val numbers: Map[Int, String] = Map()
```

or

```
val numbers = Map[Int, String]()
```

... from map to Map

- Initialization is not verbose

```scala
val numbers = Map[Int, String](
  1 -> "one",
  2 -> "two",
  3 -> "three"
)
```

... and then to groupBy

- Now that we have Lists and Maps

```
scala1> val list = List(1, 2, 3)
list: List[Int] = List(1, 2, 3)

scala> val isOdd = (x: Int) => x % 2 == 1
isOdd: Int => Boolean = <function>

scala> val oddeven = list.groupBy(isOdd)
oddeven: scala.collection.immutable.Map[Boolean,List[Int]]
     Map(false -> List(2), true -> List(1, 3))
```

What is the type of groupBy?

What is the type of groupBy?

```
class List[A] {
  def groupBy ...
}
```

What is the type of groupBy?

```scala
class List[A] {
  def groupBy[B](f: A => B): ???
}
```

What is the type of groupBy?

```scala
class List[A] {
  def groupBy[B](f: A => B): Map[B, List[A]]
}
```

null = Billion dollar mistake

Option

```scala
trait Option[+T]
case class Some[T](t: T) extends Option[T]
case object None extends Option[Nothing]
```

`Nothing` is a subtype of every other type but it has no instances

Option

```
scala> val xOpt: Option[Complex] = Option(null)
xOpt: Option[Complex] = None

scala> val x = xOpt.getOrElse(Complex(0, 0))
x: Complex = Complex(0.0,0.0)
```

Option (pattern match)

```
xOpt match {
  case Some(Complex(re, im)) => ...
  case None =>
}
```

Option (for comprehension)

```
for {
  x <- xOpt
} { ... }
```

- If we only care about the `Some()` case.
- In effect, you can view `Option` as a simple collection of at most one item.

For comprehension

```
for {
  item <- List(1, 2, 3)
} yield item * 2
```

- Computes a new list with each item doubled.
- This is a `map` in disguise.

For comprehension

```scala
for {
  item <- List(1, 2, 3)
} yield item * 2
```

same as

```scala
List(1, 2, 3).map(_ * 2)
```

For comprehension

```
for {
  item <- List(1, 2, 3)
} yield item * 2
```

same as

```
List(1, 2, 3).map(_ * 2)
```

Remember that

```
List(1, 2, 3).map(x => x * 2)
```

Add some more ingredients, and then you get

?

Monads! (in another talk)

```
for {
  x <- Some(Complex(0.0, 1.0))
  m <- List(1.0, 2.0, 3.0)
} yield Complex(x.re * m, x.im * m)
```

Monads! (in another talk)

```
for {
  x <- Some(Complex(0.0, 1.0))
  m <- List(1.0, 2.0, 3.0)
} yield Complex(x.re * m, x.im * m)
```

What about this?

```
for {
  x <- None
  m <- List(1.0, 2.0, 3.0)
} yield Complex(x.re * m, x.im * m)
```

And since we are talking about optional things ...

Optional arguments (class)

```
case class Collector(
  name: String,
  coins: List[Coin] = List()
  books: List[Book] = List() // = Nil
)
```

Optional arguments (class)

```scala
case class Collector(
  name: String,
  coins: List[Coin] = List()
  books: List[Book] = List() // = Nil
)
```

```scala
val collector1 = Collector(
  "John Smith"
)
```

Optional arguments (class)

```
case class Collector(
  name: String,
  coins: List[Coin] = List()
  books: List[Book] = List() // = Nil
)
```

```
val collector2 = Collector(
  "John Smith",
  List(OneEuroCoin),
  List(Book("Lord of the rings"))
)
```

Optional arguments (class)

```
case class Collector(
  name: String,
  coins: List[Coin] = List()
  books: List[Book] = List() // = Nil
)
```

```
val collector2 = Collector(
  name = "John Smith",
  coins = List(OneEuroCoin),
  books = List(Book("Lord of the rings"))
)
```

Not covered

Easy stuff

- Try[T] data type
- Future[T] data type
- lazy values
- by-name parameters
- ...

Not so easy stuff

- Variance (covariance, contravariance)
  - `List[+T]`
- Type constructors, higher kinds
  - *(very informally)*
  - `List`: `T` → `List[T]`
- Implicits
  - real power (cf. Haskell type classes).

Epilogue

Opinions circa 2009

- *"If I were to pick a language to use today other than Java, it would be Scala"*
  James Gosling, creator of **Java**

- *"If someone had shown me the 'Programming in Scala' book back in 2003, I'd probably have never created Groovy"*
  James Strachan, creator of **Groovy**

Resources

- scala-lang.org
- google.com/search?q=awesome-scala
- github.com/trending/scala

Thank you!

Slides that did not survive

BEGIN warm-up

END warm-up

BEGIN war map

END war map

Where to find me

```sql
select username
from   venue
where  username = 'loverdos'
and    venuename in ('gmail', 'twitter', 'github')
```