What The Hell Is This „Scala“ Anyway?

Sebastian Jackel

What's to come?

- What is Scala?
- Basic Language Features
- Collections
- Functional Code
- Concurrency and Actors

What is Scala?

- Scala is a hybrid OO/functional language
- It runs on the JVM
- It is statically typed
- It was designed by Martin Odersky

Hybrid? Like a pretentious Toyota?

- More like two programming paradigms complementing each other
- Many scripting langs have done this for ages
- Why not do this in a fast, statically typed language?

Hybrid? Like a pretentious Toyota?

- More like two programming paradigms complementing each other
- Many scripting langs have done this for ages
- Why not do this in a fast, statically typed language?

OO + FP + strong type system = Fun!

Martin Odersky? Who‘s that?

- Not the guy who invented peanut butter!
- But he co-wrote GJ which became javac of Java 1.3+.
- He also co-designed Java Generics
- He‘s currently a professor at EPFL in Lausanne, Switzerland

Why do we even need another language?

- Concurrency is more prevalent than ever
- In bigger environments, you want more powerful, higher level abstractions
- You want functional programming (oh yes, you do!)

```
10 BASICS
20 GOTO 10
```

The obligatory (and only) comparison to Java

```
public class Employee {
  private final String name;
  private double salary;

  public Employee(String name, double salary) {
    this.name = name;
    this.setSalary(salary);
  }

  public String getName() { return name; }
  public double getSalary() { return salary; }
  public void setSalary(double salary) {
    this.salary = salary;
  }
}
```

The obligatory (and only) comparison to Java

That code becomes

```
class Employee(val name: String, var salary: Double)
```

So the basic syntax is...?

```
class Employee(val name: String, var salary: Double) {

  def changeSalary(amount: Double) {
    salary += amount
  }

  def total = salary
}
```

So the basic syntax is...?

Class declaration

Fields & Constructor args

```
class Employee(val name: String, var salary: Double) {

  def changeSalary(amount: Double) {
    salary += amount
  }

  def total = salary
}
```

Added some method declarations for fun

What else is there to know?

- There are no static methods. You place those in singleton objects instead.
- In Scala everything REALLY is an object:

```
val x = 1 + 2
val y = 2.+(3)
```

What else is there to know?

- There are no static methods. You place those in singleton objects instead.
- In Scala everything REALLY is an object:

```
val x = 1 + 2
val y = 2.+(3)
```

Many things that seem to be keywords are just methods!

Inheritance & Traits

- Like Java, Scala does not do straight multiple inheritance
- Scala's *traits* are richer than interfaces though.
  - They may implement methods and fields.
  - They can be mixed in at runtime

Ye olde Logger example

```
trait Logging {
  val level: Level

  def log(msg: String) {
    preferredLogLib.log(level, msg)
  }
}
```

Ye olde Logger example

```
trait Logging {
  val level: Level

  def log(msg: String) {
    preferredLogLib.log(level, msg)
  }
}

val app = new ImportantApp(...) with
Logging { val level = DEBUG }

app.log("Shit -> Fan!!!")
```

Shamelessly nabbed from Dean Wampler's excellent slides!

```
["Coll", "ect", "ions"]
```

Let‘s have a look at collections

Lists are fun(ctional):

```
scala> val nums = 1 :: 2 :: 3 :: 4 :: 5 :: Nil
nums: List[Int] = List(1, 2, 3, 4, 5)

scala> nums head
res0: Int = 1

scala> nums tail
res1: List[Int] = List(2, 3, 4, 5)
```

Let‘s have a look at collections

```
scala> nums map {_ * 2}
res2: List[Int] = List(2, 4, 6, 8, 10)

scala> nums reduceLeft {_ + _}
res3: Int = 15

scala> nums filter {_ % 2 == 0}
res4: List[Int] = List(2, 4)
```

Let's have a look at collections

```
scala> nums map {_ * 2}
res2: List[Int] = List(2, 4, 6, 8, 10)

scala> nums reduceLeft {_ + _}
res3: Int = 15

scala> nums filter {_ % 2 == 0}
res4: List[Int] = List(2, 4)
```

The compiler does some awesome inference!

Function Literals instead of clunky anon inner classes

Don‘t go without a map

```
val salaries = Map(
  "Ralf" -> 2000.0,
  "Erik" -> 1500.0,
  "Don" -> 1500.0
)
```

```
val employees = for((name, salary) <- salaries)
    yield new Employee(name, salary)
```

The Option type

```
scala> salaries.get("Ralf")
res0: Option[Double] = Some(2000.0)
```

```
scala> salaries.get("Kurt")
res1: Option[Double] = None
```

- It is either None or Some(value)
- It is iterable via for-comprehension
- It helps prevent NPEs

We have more Options

```
for(kurt <- salaries.get("Kurt")) println(kurt)
```

won‘t print anything, but won‘t throw an NPE either,
because kurt has value „None“

We have more Options

```
for(kurt <- salaries.get("Kurt")) println(kurt)
```

won‘t print anything, but won‘t throw an NPE either, because kurt has value „None“

- Java collections return null all the time
- Those can be wrapped, because

We have more Options

```
for(kurt <- salaries.get("Kurt")) println(kurt)
```

won‘t print anything, but won‘t throw an NPE either, because kurt has value „None“

- Java collections return null all the time
- Those can be wrapped, because

```
scala> Option(null)
res0: Option[Null] = None
```

We have more Options

```
for(kurt <- salaries.get("Kurt")) println(kurt)
```

won‘t print anything, but won‘t throw an NPE either, because kurt has value „None“

- Java collections return null all the time
- Those can be wrapped, because

```
scala> Option(null)
res0: Option[Null] = None
```

Note: If you‘re using null and you‘re not working with Java libraries, you‘re doing it wrong!

While we're looking at for...

for can also accumulate results:

```
for(i <- 1 to 5) yield {i*2}
```

returns

```
Vector(2, 4, 6, 8, 10)
```

While we're looking at for...

for can also accumulate results:

```
for(i <- 1 to 5) yield {i*2}
```

returns

```
Vector(2, 4, 6, 8, 10)
```

```
for(i <- 1 to 5 if (i % 2 == 0)) yield i
```

returns

```
Vector(2, 4)
```

More Functional Code

Why would you want functional programming?

- You want to take away the pain from coding concurrent or parallel algorithms.
  - Means: You want to minimize shared mutable state.
- You want to keep your code concise.

An immutable Employee

```
case class Employee(name: String, salary: Double) {

  def modifySalary(modify: Double => Double) = {
    Employee(name, modify(salary))
  }
...
}
```

An immutable Employee

```
case class Employee(name: String, salary: Double) {

  def modifySalary(modify: Double => Double) = {
    Employee(name, modify(salary))
  }
...
}
```

Since we‘re immutable now, we return a new instance here

An immutable Employee

```
case class Employee(name: String, salary: Double) {

  def modifySalary(modify: Double => Double) = {
    Employee(name, modify(salary))
  }
...
}
```

Since we‘re immutable now, we return a new instance here

- Just adding the keyword „case“ to a class gives us:
  - A factory method for object construction
  - All fields are automatically considered as vals
  - Implementations of toString, hashcode and equals
  - Enables pattern matching

An immutable Employee

```
case class Employee(name: String, salary: Double) {

  def modifySalary(modify: Double => Double) = {
    Employee(name, modify(salary))
  }
...
}
```

Since we‘re immutable now, we return a new instance here

- Just adding the keyword „case“ to a class gives us:
  - A factory method for object construction
  - All fields are automatically considered as vals
  - Implementations of toString, hashcode and equals
  - Enables pattern matching

Allows for testing structural equality

Pattern Matching

Remember Option?

```
Option(dbResultSet.getString("name")) match {
  case Some(s: String) => s
  case None => ""
}
```

At first glance this looks like a mightier switch doesn‘t it?

Pattern Matching

Let‘s take a simple model of expressions:

```
sealed trait Expr
case class Num(value: Int) extends Expr
case class Plus(left: Expr, right: Expr) extends Expr
case class Minus(left: Expr, right: Expr) extends Expr
case class Mult(left: Expr, right: Expr) extends Expr
case class Div(left: Expr, right: Expr) extends Expr
```

Pattern Matching

Let‘s take a simple model of expressions:

```
sealed trait Expr
case class Num(value: Int) extends Expr
case class Plus(left: Expr, right: Expr) extends Expr
case class Minus(left: Expr, right: Expr) extends Expr
case class Mult(left: Expr, right: Expr) extends Expr
case class Div(left: Expr, right: Expr) extends Expr
```

To evaluate instances of those, we can just do this:

Pattern Matching

Let‘s take a simple model of expressions:

```
sealed trait Expr
case class Num(value: Int) extends Expr
case class Plus(left: Expr, right: Expr) extends Expr
case class Minus(left: Expr, right: Expr) extends Expr
case class Mult(left: Expr, right: Expr) extends Expr
case class Div(left: Expr, right: Expr) extends Expr
```

To evaluate instances of those, we can just do this:

```
def eval(e: Expr): Int = e match {
  case Num(n) => n
  case Plus(l, r) => eval(l) + eval(r)
  case Minus(l, r) => eval(l) - eval(r)
  case Mult(l, r) => eval(l) * eval(r)
  case Div(l, r) => eval(l) / eval(r) // May throw Exception
}
```

Pattern Matching

Let‘s take a simple model of expressions:

```
sealed trait Expr
case class Num(value: Int) extends Expr
case class Plus(left: Expr, right: Expr) extends Expr
case class Minus(left: Expr, right: Expr) extends Expr
case class Mult(left: Expr, right: Expr) extends Expr
case class Div(left: Expr, right: Expr) extends Expr
```

To evaluate instances of those, we can just do this:

```
def eval(e: Expr): Int = e match {
  case Num(n) => n
  case Plus(l, r) => eval(l) + eval(r)
  case Minus(l, r) => eval(l) - eval(r)
  case Mult(l, r) => eval(l) * eval(r)
  case Div(l, r) => eval(l) / eval(r) // May throw Exception
}
```

We can inspect the structure of case class instances

It‘s everywhere

Pattern matching doesn‘t just happen inside a `match` expression:

```
val employees = for((name, salary) <- salaries)
    yield new Employee(name, salary)
```

...and we‘ll see it again.

It's everywhere

Pattern matching doesn't just happen inside a `match` expression:

We had it here!

```
val employees = for((name, salary) <- salaries)
    yield new Employee(name, salary)
```

...and we'll see it again.

Function literals

You can declare anonymous functions („lambdas“) like this:

```
scala> val fun = (x: Int) => x * 2
fun: (Int) => Int = <function1>
```

Function literals

You can declare anonymous functions („lambdas“) like this:

```
scala> val fun = (x: Int) => x * 2
fun: (Int) => Int = <function1>
```

(You‘ve already seen it used with collections:)

```
scala> nums map {_ * 2}
res2: List[Int] = List(2, 4, 6, 8, 10)
```

Putting lambdas to use

```
class Employee(val name: String, var salary: Double) {

  def changeSalary(amount: Double) {
    salary += amount
  }

  def total = salary
}
```

- This `changeSalary()` method can only add or substract from the employee‘s salary.
- A 5% wage increase is unnecessarily complicated.
- Might as well use the field‘s Getter/Setter instead.

Putting lambdas to use

```
class Employee(val name: String, var salary: Double) {

  def modifySalary(modifier: Double => Double) {
    salary = modifier(salary)
  }

  def total = salary
}
```

Putting lambdas to use

```
class Employee(val name: String, var salary: Double) {

  def modifySalary(modifier: Double => Double) {
    salary = modifier(salary)
  }

  def total = salary
}
```

We can increase this employee‘s salary by 5% easily now:

```
scala> var simon = new Employee(”Simon”, 2000.0)
simon: Employee = Employee@f1d5566

scala> simon.modifySalary((x: Double) => x * 1.05)
```

Putting lambdas to use

```
class Employee(val name: String, var salary: Double) {

  def modifySalary(modifier: Double => Double) {
    salary = modifier(salary)
  }

  def total = salary
}
```

We can increase this employee‘s salary by 5% easily now:

```
scala> var simon = new Employee(”Simon”, 2000.0)
simon: Employee = Employee@f1d5566

scala> simon.modifySalary((x: Double) => x * 1.05)
```

Or even shorter: `_ * 1.05`

Putting lambdas to use

That‘s more like it:

```
class Employee(val name: String, var salary: Double) {

  def modifySalary(modifier: Double => Double) {
    salary = modifier(salary)
  }

  def total = salary
}
```

We can increase this employee‘s salary by 5% easily now:

```
scala> var simon = new Employee(”Simon”, 2000.0)
simon: Employee = Employee@f1d5566

scala> simon.modifySalary((x: Double) => x * 1.05)
```

Or even shorter: `_ * 1.05`

Currying

Currying means partially applying a function

```
scala> val fun = (x: Int) => (y: Int) => x * y
fun: (Int) => (Int) => Int = <function1>

scala> val timesTwo = fun(2)
bytwo: (Int) => Int = <function1>

scala> nums map timesTwo
res0: List[Int] = List(2, 4, 6, 8, 10)
```

Functions as objects

- Any object can be used like a function by implementing an `apply()` method.

- In fact, functions are just objects:

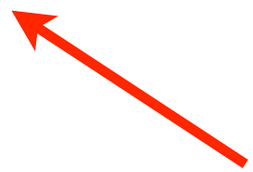
```
trait Function1[-T1, +R] {
  def apply(v1: T1): R
}
```

Functions as objects

- Any object can be used like a function by implementing an `apply()` method.

- In fact, functions are just objects:

```
trait Function1[-T1, +R] {
  def apply(v1: T1): R
}
```



We‘re of course eliding some convenience methods here

Con → | → cy

curren →

Hanging by a thread...

Just use Java threads if you want to:

```
scala> new Thread {println("Serious Business!")}
Serious Business!
res0: java.lang.Thread = Thread[Thread-4,5,main]
```

But if you do, you'll be facing the lock and synchronization issues that come with the paradigm.

Actors

- Small entities that communicate by sending each other messages
- side-effecting behaviour can be contained within one actor.
- Only the actors‘ inboxes need to be synchronized.
- In Scala, actors run on a threadpool that is expanded as needed.
  - not necessarily 1 actor == 1 thread

Actors

A Scala actor implements the Actor trait:

```
class UselessActor extends Actor {
  def act() = {
    println(„Look ma, I‘m acting!“)
  }
}
```

```
scala> val wahlberg = new UselessActor()
wahlberg: UselessActor = UselessActor@7bc8b313

scala> wahlberg.start()
res0: scala.actors.Actor = UselessActor@7bc8b313
Look ma, I'm acting!
```

Actors

Using `react()` or `receive()` and `loop()`, an actor can communicate via messages.

```
class SimpleActor extends Actor {
  def act() = loop {
    react {
      case s: String => println(„Received: “ + s)
    }
  }
}
```

Actor communication

```
case class Ping(sender: Actor)
case class Pong(count: Int, sender: Actor)
case object Stop
```

```
class PingActor(val max: Int) extends Actor {
   var count = 0

   def act() = loop {
      react {
         case Ping(sender) => if (count == max) {
            sender ! Stop
            self ! Stop
         } else {
            count += 1
            println(„Got Ping no. “ + count)
            sender ! Pong(count, self)
         }
         case Stop => {
            println(„Exiting!“)
            exit()
         }
      }
   }
}
```

```
class PongActor extends Actor {

   def replyTo(target: Actor) {
      val sender = self
      actor {
         Thread.sleep(1000)
         target ! Ping(sender)
      }
   }

   def act() = loop {
      react {
         case Pong(count, sender) => {
            println(„Got Pong no. “ + count)
            replyTo(sender)
         }
         case Stop => {
            println(„Exiting!“)
            exit()
         }
      }
   }
}
```

Also:

```
val billMurray = new GreatActor()
```

Advanced Features

Implicit Conversions

Let‘s specify a type for complex numbers:

```
case class Complex(real: Double, imag: Double) {

  def +(other: Complex) =
    Complex(real + other.real, imag + other.imag)
  ... // Further methods elided

}
```

What we‘d like to do:

```
scala> 1.0 + Complex(1.0, 1.0)
res0: Complex = Complex(2.0,1.0)
```

Implicit Conversions

Of course Double has no method

```
+: Complex => Complex
```

Implicit Conversions

Of course Double has no method

```
+: Complex => Complex
```

We can do something about that though:

```
implicit def doubleToComplex(d: Double) = Complex(d, 0.0)
```

Implicit Conversions

Of course Double has no method

```
+: Complex => Complex
```

We can do something about that though:

```
implicit def doubleToComplex(d: Double) = Complex(d, 0.0)
```

If the compiler can‘t resolve types, it tries to insert implicit conversions.

„Pimp My Library“ Pattern

Stuff like this is used a lot:
For example array-like access to characters in strings.

```
implicit def strToStringWrapper(s: String) =
  new RandomAccessSeq {
    def length = s.length
    def apply(i: Int) = s.charAt(i)
  }
```

```
scala> ”evil”(3)
res0: Char = l

scala> ”evil” map {_ toUpper}
res1: String = EVIL
```

The commandments of implicit conversions

- Only definitions marked as implicit are available.
- An implicit conversion must be in scope as single identifier or associated with source or target type of conversion.
- An implicit conversion is only inserted if there is no other possible conversion to insert.
- Only one implicit is tried.
- If the code type-checks, no implicit is invoked.

Implicit Parameters

The compiler will also insert parameter lists.

```
case class Drink(name: String)

implicit val russian = Drink(”White Russian”)

def favoriteDrink(person: String)(implicit drink: Drink) {
  println(person + ” likes to drink ” + drink.name)
}

favoriteDrink(”The Dude”)
```

Typeclass Pattern

The Haskell programmer rejoices!

```
trait Monoid[T] {
  def mzero: T
  def mappend(a: T, b: T): T
}

implicit object IntMonoid extends Monoid[Int] {
  def mzero = 0
  def mappend(a: Int, b: Int) = a + b
}

def sum[A](xs: List[A])(implicit m: Monoid[A]) {
  if (xs.isEmpty) m.mzero
  else m.mappend(xs.head, sum(xs.tail))
}
```

scalaz

Scalate

Akka

squeryl

Bonus Round: The Scala Environment

specs

Scala-IDE

Play

sbt

ScalaTest

Lift

How to manage Scala projects

- ant - if you‘re that masochistically inclined
- maven-scala-plugin
  - From the official scala-tools Maven Repo:

    http://scala-tools.org/repo-releases
- sbt - Simple Build Tool

Simple Build Tool

- Basically replaces Maven
- Uses Apache Ivy for dependency management
- Comfortably XML-free
- Lots of plugins

IDE Integration

- IntelliJ IDEA plugin
  - Great code completion, analysis and refactoring support
  - Sometimes a bit unstable
- Scala-IDE for Eclipse:
  - Has for a long time been so-so
  - On it‘s way for a complete overhaul for Scala 2.9 (looks promising)
- Bundles for vim, emacs, TextMate, Insert your favorite editor here...

Akka

- Platform for scalable, event-driven and fault tolerant applications
- Provides very lightweight actors (~600 bytes per Instance)
- Software Transactional Memory
- „Let-it-Crash“ philosophy
- Java and Scala API

Noteworthy Libs & Frameworks

- Lift - a View-First web application framework
- Play - If you prefer MVC instead
- scalaz - Pure functional data structures for Scala (inspired by Haskell)
- Scalate - Scala Template Engine
- Scalatest, Scalacheck and Specs2 - three TDD/BDD libraries

Oh the songs we didn‘t sing...

- XML literals
- combinator parsing
- continuations
- the power of the type system
  - advanced inheritance
  - type bounds
  - structural typing
  - type classes and higher kinded types
  - fluffy kittens

Scala Sources

Sites

www.scala-lang.org - main language site
www.implicit.ly - news about Scala software releases
www.scala-ide.org - home of the Scala Eclipse plugin
www.akka.io - Akka concurrency framework
www.liftweb.net - Lift web framework

Free online books

programming-scala.labs.oreilly.com - „Programming Scala“ by Dean Wampler and Alex Payne
www.artima.com/pins1ed - „Programming in Scala“ 1st ed. by Lex Spoon, Bill Venners and Martin Odersky
simply.liftweb.com - Simpy Lift by David Pollak
www.scala-lang.org/docu/files/ScalaByExample.pdf - „Scala By Example“ (Draft), a PDF tutorial on Scala by Martin Odersky

Sources

Seductions of Scala - by Dean Wampler - excellent presentation!

That's all folks!

Questions?

Thanks for advice and encouragement to:
Dean Wampler (@deanwampler)
Heiko Seeberger (@hseeberger)
Mario Gleichmann (@mariogleichmann)
Ralf Lämmel (@reallynotabba)