



Scala

an overview

Alceste Scalas

Imperial College
London

Programming languages reading group

31 October 2017



Scala: what's the hype about?

- ▶ **Object oriented** *and* **functional** programming language
- ▶ Runs on the **JVM**...
 - ▶ **interoperability with Java** and other languages
 - ▶ **influence** on Java 8/9 (lambdas)
- ▶ ...and also compiles as:
 - ▶ **Javascript** (Scala.js)
 - ▶ **native code** (Scala Native)
- ▶ Powerful **types**, with **innovations** (path-dependent types)

Object-Oriented Programming

```
abstract class Pet(val name: String) {  
  val scratches: Boolean // Abstract, immutable  
}
```

Object-Oriented Programming

```
abstract class Pet(val name: String) {  
  val scratches: Boolean // Abstract, immutable  
}
```

```
trait Tricks {  
  var sitting = false // Concrete, mutable  
  def sit    = { sitting = true }  
  def stand = { sitting = false }  
  def speak: Unit // Abstract  
}
```

Object-Oriented Programming

```
abstract class Pet(val name: String) {  
  val scratches: Boolean // Abstract, immutable  
}
```

```
trait Tricks {  
  var sitting = false // Concrete, mutable  
  def sit     = { sitting = true }  
  def stand  = { sitting = false }  
  def speak: Unit // Abstract  
}
```

```
class Cat(name: String) extends Pet(name) {  
  override val scratches = true  
}
```

Object-Oriented Programming

```
abstract class Pet(val name: String) {  
  val scratches: Boolean // Abstract, immutable  
}
```

```
trait Tricks {  
  var sitting = false // Concrete, mutable  
  def sit     = { sitting = true }  
  def stand  = { sitting = false }  
  def speak: Unit // Abstract  
}
```

```
class Cat(name: String) extends Pet(name) {  
  override val scratches = true  
}
```

```
class Dog(name: String) extends Pet(name) with Tricks {  
  override val scratches = false  
  override def speak = println("Woof!")  
}
```

Value semantics

```
class Pumpkin(val ripeness: Int,  
              val weight: Int)
```

```
val p1 = new Pumpkin(10, 5)
```

```
val p2 = new Pumpkin(10, 5)
```

```
p1 == p2 // false
```

Value semantics

```
class Pumpkin(val ripeness: Int,  
              val weight: Int)
```

```
val p1 = new Pumpkin(10, 5)
```

```
val p2 = new Pumpkin(10, 5)
```

```
p1 == p2 // false
```

```
case class Pumpkin(ripeness: Int,  
                  weight: Int)
```

```
val p1 = Pumpkin(10, 5)
```

```
val p2 = Pumpkin(10, 5)
```

```
p1 == p2 // true
```


Algebraic data types

```
sealed abstract class Term
case class Const(value: Int) extends Term
case class Neg(t: Term) extends Term
case class Sum(t1: Term, t2: Term) extends Term

val t = Neg( Sum( Const(1), Const(2) ) )
```

Algebraic data types

```
sealed abstract class Term
case class Const(value: Int) extends Term
case class Neg(t: Term) extends Term
case class Sum(t1: Term, t2: Term) extends Term

val t = Neg( Sum( Const(1), Const(2) ) )
```

```
def eval(t: Term): Int = t match {
  case Const(value) => value
  case Neg(t)       => -eval(t)
  case Sum(t1, t2)  => eval(t1) + eval(t2)
}
eval(t) // Returns: -3
```

Algebraic data types

```
sealed abstract class Term
case class Const(value: Int) extends Term
case class Neg(t: Term) extends Term
case class Sum(t1: Term, t2: Term) extends Term

val t = Neg( Sum( Const(1), Const(2) ) )
```

```
def eval(t: Term): Int = t match {
  case Const(value) => value
  case Neg(t)       => -eval(t)
  case Sum(t1, t2)  => eval(t1) + eval(t2)
}

eval(t) // Returns: -3
```

```
def str(t: Term): String = t match {
  case Const(value) => f"${value}"
  case Neg(t)       => f"-(${str(t)})"
  case Sum(t1, t2)  => f"${str(t1)} + ${str(t2)}"
}

str(t) // Returns: "-(1 + 2)"
```

Functional programming [\(https://alvinalexander.com/scala/...\)](https://alvinalexander.com/scala/...)

```
val chars = 'a' to 'z'  
  
// All pairs of different chars  
val perms = chars.map { a =>  
  chars.filter { b =>  
    a != b  
  }.map { b =>  
    f"${a}${b}"  
  }  
}.flatten
```

Functional programming [\(https://alvinalexander.com/scala/...\)](https://alvinalexander.com/scala/)

```
val chars = 'a' to 'z'  
  
// All pairs of different chars  
val perms = chars.map { a =>  
  chars.filter { b =>  
    a != b  
  }.map { b =>  
    f"${a}${b}"  
  }  
}.flatten
```

```
// Syntactic sugar: for-comprehension  
val perms2 = for {  
  a <- chars  
  b <- chars if (a != b)  
} yield f"${a}${b}"  
  
perms == perms2 // true
```

Monads

 (<http://scabl.blogspot.co.uk/2013/02/monads-in-scala-1.html>)

```
sealed abstract class MMaybe[+A] {  
  // flatMap corresponds to >=> in Haskell  
  def flatMap[B](f: A => MMaybe[B]): MMaybe[B]  
  
  // map corresponds to >> in Haskell  
  def map[B](f: A => B): MMaybe[B] = {  
    flatMap { a => MJust(f(a)) }  
  }  
}  
  
case class MJust[+A](a: A) extends MMaybe[A] {  
  override def flatMap[B](f: A => MMaybe[B]) = f(a)  
}  
  
case object MNothing extends MMaybe[Nothing] {  
  override def flatMap[B](f: Nothing => MMaybe[B]) = MNothing  
}
```

Monads

 (<http://scabl.blogspot.co.uk/2013/02/monads-in-scala-1.html>)

```
sealed abstract class MMaybe[+A] {
  // flatMap corresponds to >=> in Haskell
  def flatMap[B](f: A => MMaybe[B]): MMaybe[B]

  // map corresponds to >> in Haskell
  def map[B](f: A => B): MMaybe[B] = {
    flatMap { a => MJust(f(a)) }
  }
}

case class MJust[+A](a: A) extends MMaybe[A] {
  override def flatMap[B](f: A => MMaybe[B]) = f(a)
}

case object MNothing extends MMaybe[Nothing] {
  override def flatMap[B](f: Nothing => MMaybe[B]) = MNothing
}

for { x <- MJust(1); y <- MJust(2) } yield (x,y) // MJust((1,2))
for { x <- MJust(1); y <- MNothing } yield (x,y) // MNothing
```

Implicit parameters

```
def pat(implicit d: Dog) = {  
  println(f"Good ${d.name}!")  
}  
  
implicit val dog = new Dog("Rex")  
pat  
pat  
// Prints: Good Rex! (2 times)
```


Implicit parameters and conversions

```
def pat(implicit d: Dog) = {  
  println(f"Good ${d.name}!")  
}  
  
implicit val dog = new Dog("Rex")  
pat  
pat  
// Prints: Good Rex! (2 times)
```

```
def says(d: Dog) = {  
  println(f"${d.name} says:"); d.speak  
}  
  
implicit def toDog(name: String) = new Dog(name)  
  
says("Rocky")  
// Rocky says:  
// Woof!
```

Path-dependent types [\(http://danielwestheide.com/blog/2013/02/13/...\)](http://danielwestheide.com/blog/2013/02/13/...)

```
case class Franchise(title: String) {  
  case class Character(name: String)  
  def friends(c1: Character, c2: Character) = (c1, c2)  
}
```

Path-dependent types [\(<http://danielwestheide.com/blog/2013/02/13/...>\)](http://danielwestheide.com/blog/2013/02/13/...)

```
case class Franchise(title: String) {  
  case class Character(name: String)  
  def friends(c1: Character, c2: Character) = (c1, c2)  
}
```

```
val starTrek = Franchise("Star Trek")  
val kirk = starTrek.Character("James T. Kirk")  
val spock = starTrek.Character("Spock")
```

```
val starWars = Franchise("Star Wars")  
val luke = starWars.Character("Luke Skywalker")  
val yoda = starWars.Character("Yoda")
```

Path-dependent types <http://danielwestheide.com/blog/2013/02/13/...>

```
case class Franchise(title: String) {  
  case class Character(name: String)  
  def friends(c1: Character, c2: Character) = (c1, c2)  
}
```

```
val starTrek = Franchise("Star Trek")  
val kirk = starTrek.Character("James T. Kirk")  
val spock = starTrek.Character("Spock")
```

```
val starWars = Franchise("Star Wars")  
val luke = starWars.Character("Luke Skywalker")  
val yoda = starWars.Character("Yoda")
```

```
starTrek.friends(kirk, spock) // OK  
starWars.friends(luke, yoda) // OK
```

```
starTrek.friends(kirk, yoda)  
// error: type mismatch;  
// found   : starWars.Character  
// required: starTrek.Character
```

The future: Dotty

A reimplementaion of Scala with:

- ▶ a cleaner **theoretical basis** (the DOT calculus)
- ▶ **advanced features** (e.g., intersection and union types)



<http://dotty.epfl.ch/>

Protocols as types: in research

[START]

A client program can request
either:

- ▶ the **Transactions**
for a certain date
 - ▶ then, the bank can
either:
 - ▶ reply with
a **Report**, and
the session goes
back to [START]
 - ▶ *or*, reply **Bye**
with the report,
and the session
ends
- ▶ or **Quit**:
 - ▶ then, the session ends

Protocols as types: in research

[START]

A client program can request *either*:

- ▶ the **Transactions** for a certain date
 - ▶ then, the bank can *either*:
 - ▶ reply with a **Report**, and the session goes back to [START]
 - ▶ or, reply **Bye** with the report, and the session ends
- ▶ or **Quit**:
 - ▶ then, the session ends

Scala + lchannels (ECOOP'16)

```
def client(b: Out[Start]): Unit = {  
  if (...) {  
    val b2 = b !! Trans("08-10-2017")_  
  
    b2 ? {  
      case m @ Report(txt) => client(m.cont)  
      case Bye(txt)       => return  
    }  
  } else {  
    b ! Quit()  
  }  
}
```

Protocols as types: in industry

akka

TRY AKKA DOCUMENTATION BLOG GET INVOLVED

Build powerful reactive, concurrent, and distributed applications more easily

- ✦ Simpler Concurrent & Distributed Systems
- 🛡️ Resilient by Design
- 🚀 High Performance
- ☁️ Elastic & Decentralized
- 🔄 Reactive Streaming Data

Proven in production

Organizations with extreme requirements rely on Akka and other Lightbend technologies.

iHeart MEDIA Capital One credit karma intel Hootsuite NORWEGIAN CRUISE LINE UPSIDE Walmart PayPal amazon.com zalando weightwatchers

Latest News

Akka Typed: New Cluster Tools API

Oct 04 2017

In previous post we looked at the the Cluster and Receptionist for Akka Typed. In this post you will be introduced to the new typed APIs for Distributed Data, Cluster...

Protocols as types: in industry

The screenshot shows the Akka.io website. At the top, there is a navigation bar with the Akka logo and links for 'TRY AKKA', 'DOCUMENTATION', 'BLOG', and 'GET INVOLVED'. Below this is a hero section with a blue background. On the left, it says 'Build powerful reactive, concurrent, and distributed applications more easily'. On the right, there are five features listed with icons: 'Simpler Concurrent & Distributed Systems', 'Resilient by Design', 'High Performance', 'Elastic & Decentralized', and 'Reactive Streaming Data'. Below the hero section is a 'Proven in production' section with the text 'Organizations with extreme requirements rely on Akka and other Lightbend technologies.' and logos for iHeart MEDIA, Capital One, credit karma, intel, Hootsuite, NORWEGIAN CRUISE LINE, UPSIDE, Walmart, PayPal, amazon.com, zalando, and weightwatchers. At the bottom is a 'Latest News' section with a highlighted article titled 'Akka Typed: New Cluster Tools API' dated Oct 04 2017. The article text is partially visible: 'In previous post we looked at the the Cluster and Receptionist for Akka Typed. In this post we will be introduced to the new typed APIs for Distributed'.

Conclusions

Scala combines **object-oriented** and **functional** programming

Scala is **interoperable with Java**, and reuses its software libraries

Interest on **type-safe concurrent programming**. See:

<http://alcestes.github.io/lchannels>

(Library for type-safe distributed applications)

Paper: **“Lightweight session programming in Scala”**

(by A. Scalas and N. Yoshida; 30th European Conference on Object-Oriented Programming, 2016)