

Scala

Carlos Tomé Cortiñas, Renate Eilers and Matthew Swart



Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

Table of Contents

- ▶ Introduction
- ▶ Static semantics
- ▶ Dynamic semantics





Figure 1: Martin Odersky



History

From A Brief, Incomplete, and Mostly Wrong History of Programming Languages by James Iry

...

2003 - A drunken Martin Odersky sees a Reese's Peanut Butter Cup ad featuring somebody's peanut butter getting on somebody else's chocolate and has an idea. He creates Scala, a language that unifies constructs from both object oriented and functional languages. This pisses off both groups and each promptly declares jihad.



History

- ▶ Created by Prof. Martin Odersky at EPFL.
- ▶ Design started in 2001.
- ▶ Released to the public in 2003.
- ▶ But research on Scala is still going on nowadays.



Is Scala popular?

- ▶ Twitter
- ▶ LinkedIn
- ▶ The Guardian
- ▶ FourSquare
- ▶ Sony
- ▶ Many more...

	TIOBE	Redmonk
Scala	31	14
Java	1	2
Haskell	38	16



Projects

- ▶ SBT
- ▶ Squeryl
- ▶ PlayFramework
- ▶ Akka
- ▶ Finagle
- ▶ React
- ▶ Apache Kafka
- ▶ Apache Spark



Overview of features

- ▶ Scala is **F**unctional and **O**bject **O**riented.
- ▶ Scala is **S**tatically **T**yped.
- ▶ Scala compiles to Java bytecode and runs on the **JVM** (write once, run everywhere)
- ▶ Scala can seamlessly interoperate with Java written code.
- ▶ Scala is **not** a pure language.



The full picture

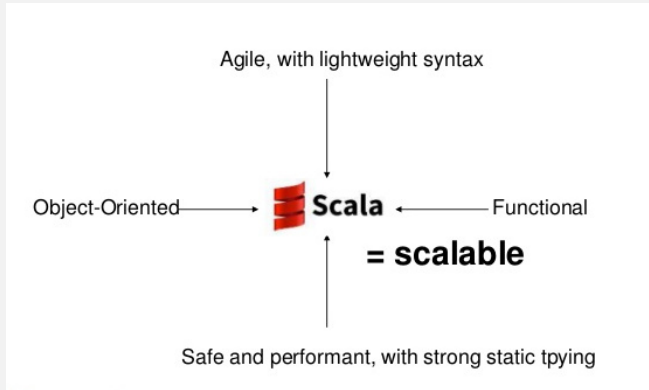


Figure 2



Object oriented but ...

- ▶ Scala approach to OO programming is quite different from Java.
- ▶ However, it has also classes and abstract classes as in Java.

```
abstract class Animal {  
    def shout : Unit  
}  
class Dog(name : String) extends Animal{  
    def shout : Unit = println("Woof!")  
}
```

- ▶ This is actually a keypoint for the interoperability between both languages.



Object oriented but ... (II)

- ▶ In Scala there are no interfaces like in Java.
- ▶ Scala supports out of the box type parameters (aka generics in Java).

```
abstract class Producer[A] {  
  def produce(x : A) : String  
}  
class IntProducer extends Producer[Int] {  
  def produce(x : Int) : String = x.toString()  
}
```

```
scala> new IntProducer().produce(666)  
res1: String = 666
```



What really makes Scala OO different?

- ▶ In Scala there is the notion of **singleton object**.
- ▶ There are also **case classes** which are a “special” kind of classes.
- ▶ **Traits** are the key construction in Scala, and can be seen as a mixture of Java abstract classes and interfaces.



Intermezzo: Covariance and contravariance

```
class GrandParent
class Parent extends GrandParent
class Child extends Parent
```

```
class CoBox[+A]
class ContraBox[-A]
```

```
def foo(x : CoBox[Parent]) : CoBox[Parent] =
  identity(x)
```

```
def bar(x : ContraBox[Parent]) : ContraBox[Parent] =
  identity(x)
```

```
foo(new CoBox[Child])           // success
foo(new CoBox[GrandParent])    // type error
```



Immutable vs Mutable

- Scala supports both immutable and mutable

```
scala> var x : Int = 1  
x: Int = 1
```

```
scala> x = 3  
x: Int = 3
```

```
scala> val y : Int = 1  
y: Int = 1
```

```
scala> y = 3  
<console>:12: error: reassignment to val  
      y = 3
```



Type hierarchy

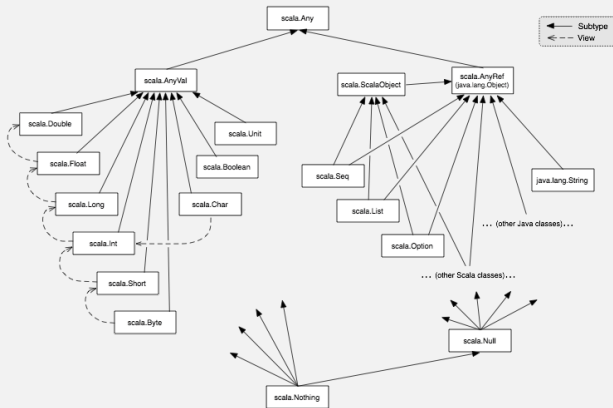


Figure 3



Singleton objects

- ▶ In Scala we are allow to define a class with only one object of such class.
- ▶ Instantiation of the class is done at the point of usage of the object.

```
object Singl {  
  val int_with_missile : Int = {  
    println("Throw missiles!");  
    1  
  }  
}  
  
scala> val s = Singl  
Throw missiles!  
s: Singl.type = Singl$@498057bb
```



Case Classes and Case Objects

Case classes

- ▶ Immutable by default
- ▶ Decomposable through pattern matching
- ▶ Compared by structural equality instead of by reference
- ▶ Succinct to instantiate and operate on

Case Object

- ▶ Does not take any arguments
- ▶ Singleton object
- ▶ Similar to the Case class(Except for the above bullets)



Case Classes and Objects - Example

```
case class Person(name : String) {  
  def noise = "I am a person"  
}
```

```
case object Frog{  
  def noise : String = "CROAK"  
}
```



Pattern Match

```
def whatDoAnySay(animal : Any) : Unit = {  
  animal match {  
    case Frog => println(Frog.noise)  
    case Person(name) => println(name)  
    case 1 => println("I am an Int")  
    case 'a' => println("I am a Character")  
    case "aaaa" => println("I am a String")  
    case x =>  
      println("Ahhh! No pattern match")  
  }  
}
```



Traits

- ▶ Traits are the fundamental unit of code reuse in **Scala**.
- ▶ A trait encapsulates method and field definitions that can be reused by any class through **mixin composition**.
- ▶ Unlike class inheritance, a class may **mixin** any number of traits.
- ▶ We will see how this works in practice.



Traits as stackable modifications

A simple Queue

```
import scala.collection.mutable.ArrayBuffer
```

```
trait Queue[A] {  
  def get() : A  
  def put(x : A) : Unit  
}
```

```
class BasicIntQueue extends Queue[Int] {  
  private val buf = new ArrayBuffer[Int]  
  def get() : Int = buf.remove(0)  
  def put(x : Int) : Unit = buf += x  
}
```



Traits as stackable modifications (II)

- ▶ Now suppose we want to modify the behaviour of the BasicIntQueue in different ways
 1. *Doubling* the integers that are inserted in the Queue
 2. *Incrementing* the integers that are inserted in the Queue
 3. *Filtering* out negative integers from the Queue
- ▶ How do we do that without modifying the existing code?



Traits as stackable modifications (III)

```
trait Doubling extends Queue[Int] {  
  abstract override def put(x : Int) : Unit = {  
    super.put(2 * x)  
  }  
}  
  
trait Incrementing extends Queue[Int] {  
  abstract override def put(x : Int) : Unit = {  
    super.put(1 + x)  
  }  
}  
  
trait Filtering extends Queue[Int] {  
  abstract override def put(x : Int) : Unit = {  
    if (x > 0) super.put(x)  
  }  
}
```



Traits as stackable modifications (III)

```
scala> val q1 =  
  | new BasicIntQueue with Incrementing  
                                with Filtering  
q1: BasicIntQueue with Incrementing with Filtering =  
$anon$1@f5a7226  
  
scala> q1.put(0)  
  
scala> q1.get()  
java.lang.IndexOutOfBoundsException: 0  
  at scala.collection.mutable.ResizableArray. ...  
  ...  
  at BasicIntQueue.get(<console>:15)  
  ... 31 elided
```



Traits as stackable modifications (IV)

```
scala> val q2 =  
  | new BasicIntQueue with Filtering  
                        with Incrementing  
q2: BasicIntQueue with Filtering with Incrementing =  
$anon$1@6f731759  
  
scala> q2.put(0)  
  
scala> q2.get()  
res7: Int = 1
```



Mixin composition & Linearization

- ▶ When resolving the calls to **super**, the mixins are resolved through process called linearization.
1. Queue \rightarrow AnyRef \rightarrow Any
 2. Filtering \rightarrow Queue \rightarrow AnyRef
 3. Incrementing \rightarrow Queue \rightarrow AnyRef
 4. BasicIntQueue \rightarrow Incrementing \rightarrow Filtering \rightarrow ...



Traits as ad-hoc polymorphism (with implicits)

- We can use *traits* to model Haskell's *type classes*

```
trait Monoid[A] {  
  def mempty : A  
  def mappend(x : A, y : A) : A  
}
```

- An instance for the type class represented by a trait is just a *value* implementing such trait

```
object IntMonoid extends Monoid[Int] {  
  def mempty : Int = 0  
  def mappend(x : Int, y : Int) : Int = x + y  
}
```



Traits as ad-hoc polymorphism (with implicits) (II)

- For example now we can define a function such as **foldMap**

```
def foldMap[A,M](f : A => M)
    (xs : List[A])
    (ma : Monoid[M]) : M = {
  xs.foldLeft(ma.empty)
    ((x : M, y : A) => ma.mappend(x, f(y)))
}
```

```
scala> foldMap((x : Int) => x)
      (List(1,2,3,4,5))
      (IntMonoid)

res1: Int = 15
```



Traits as ad-hoc polymorphism (with implicits) (III)

- ▶ Its a bit cumbersome to pass all around the instance for `Monoid[Int]`...
- ▶ Scala allows us to declare some parameters as **implicit**

```
implicit object IntMonoid extends Monoid[Int] ...
```

```
def foldMap[A,M](f : A => M)  
    (xs : List[A])  
    (implicit ma : Monoid[M]) : M = ...
```

```
scala> foldMap((x : Int) => x)(List(1,2,3,4,5))  
res1: Int = 15
```



Traits as ad-hoc polymorphism (with implicits) (IV)

- ▶ The compiler is in charge of figuring out the correct implementation for an **implicit** declared argument.
- ▶ If there are several that match the required type then apply some rules of preference. Or in the extreme case raises an error.
- ▶ Moreless works like Haskell class resolution (+ OverloadedInstances).
- ▶ But is nice that instances are first class objects because we can **explicitly** pass them as arguments.



Traits as ad-hoc polymorphism (with implicits) (V)

```
object ProdIntMonoid extends Monoid[Int] ...  
implicit object PlusIntMonoid extends Monoid[Int] ...
```

```
def foldMap[A,M](f : A => M)  
    (xs : List[A])  
    (implicit ma : Monoid[M]) : M = ...
```

```
scala> foldMap((x : Int) => x)(List(1,2,3,4,5))  
res1: Int = 15
```

```
scala> foldMap((x : Int) => x)(List(1,2,3,4,5))  
    (ProdIntMonoid)  
res2: Int = 120
```



Traits as Generalized Algebraic Datatypes (GADTs)

- With a combination of **case classes** and **traits** we can easily implement (Generalized) ADT.

(Well typed) Expression language

```
sealed trait Expr[A]

case class Val[A](x : A) extends Expr[A]
case class Add(x : Expr[Int], y : Expr[Int])
    extends Expr[Int]
case class If[A](c : Expr[Boolean])
    (x : Expr[A], y : Expr[A])
    extends Expr[A]
```



Traits as GADTs (II)

```
def eval[A](e : Expr[A]) : A = e match {  
  case Val(x) => x;  
  case Add(x,y) => eval(x) + eval(y)  
  case If(c)(x,y) =>  
    if (eval(c)) eval(x) else eval(y);  
}
```

```
scala> val ex1 =  
  | If(Val(true),Add(Val(1),Val(1)),Val(0))  
ex1: If[Int] =  
  | If(Val(true),Add(Val(1),Val(1)),Val(0))
```

```
scala> eval(ex1)  
res1: Int = 2
```



Traits and Higher-Kinded types

- ▶ Higher kinded types are types that take types as arguments
- ▶ For example in Haskell the type `Maybe :: * -> *`
- ▶ Scala supports Higher-kinded types

```
trait Functor[F[_]] {  
  def fmap[A,B](f : A => B)(F[A]) : F[B]  
}
```

- ▶ However, the support for type inference is somewhat limited and many times it cannot correctly typecheck the program.



Abstract type members

```
trait AbsCell {  
  type T  
  val init : T  
  var value : T = init  
  def get() : T = value  
  def set(x : T) : Unit = value = x  
}  
  
scala> val mc = new AbsCell { type T = Int;  
                               val init = 0 }  
mc: AbsCell{type T = Int} = $anon$1@7989d46  
  
scala> mc.set(99)  
  
scala> mc.get()  
res26: mc.T = 99
```



Path-dependent types

```
def resetCell(cell : AbsCell) = {  
  cell.set(cell.init)  
}
```

- ▶ Is this well typed?
- ▶ The expression `cell.init` has type `cell.T`
- ▶ The method `cell.set` has type `cell.T => Unit`
- ▶ `cell.T` is an example of a path-dependent type



Intermezzo: Implicit classes

- ▶ Scala also supports the definition of a class to be **implicit**.
- ▶ This makes the methods defined in the class to be available without ever instantiating the class explicitly.
- ▶ Of course, there are severe restrictions on how this classes are defined.

```
implicit class Incr(x : Int) {  
  def incr = x + 1  
}  
implicit class Decr(x : Int) {  
  def decr = x - 1  
}  
scala> 1.incr.incr.incr.decr  
res1: Int = 3
```



Intermezzo: Implicit classes (II)

- We can even add type parameters to the class so it works out of the box for any type.

```
implicit class Print[A](x : A) {  
  def print = println(x)  
}
```

```
scala> ((x : Int) => 1).print  
$line123.$read$$iw$$iw$$iw$$$Lambda$2891/...
```

```
scala> true.print  
true
```



Functional programming in Scala

- ▶ As we've seen before, Scala is defined as being a functional programming language.
- ▶ However, the core of the language are not functions but **Traits**
- ▶ While the approach of OCaml to OO+FP is to introduce an Object system on top of the core λ -**calculus**
- ▶ In Scala the approach is the opposite. Everything is an **Object**.



Functions as Objects

- ▶ In fact function values are treated as objects in Scala
- ▶ The function type $A \Rightarrow B$ is just an abbreviation for the class `scala.Function1[A,B]`, which is roughly defined as follows.

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

So functions are object that implement such **Trait** with `apply`

There are also traits `Function2`, `Function3`, ... for functions which take more parameters (Currently up to 22)



Functions as Objects (II)

- ▶ So are function values?
- ▶ Every object is a value in Scala, and functions are objects so ...
- ▶ But how about methods, are they values?
- ▶ Not really.

```
def id[A](x : A):A = x
```

- ▶ But we can turn any method into a function value using `_` (underscore) in the argument positions.

```
scala> id _
```



Functions as Objects (III)

```
scala> (id _)(1)
<console>:19: error: type mismatch;
 found   : Int(1)
 required: Nothing
      (id _)(1)
              ^
```

- What just happened?

```
scala> id _
res1: Nothing => Nothing = $$Lambda$2921/351794524
```

- Suspicious ...



Functions as Objects (IV)

- ▶ We need to actually provide the type of the parameter explicitly, because during the conversion to a function **Trait** Scala can't figure out the type parameters (they need to be concrete!).
- ▶ Now better.

```
scala> (id[String] _)("Hello")  
res1: String = "Hello"
```



Intermezzo: Apply method and Companion object

- ▶ Any class or trait can implement a method apply.
- ▶ And use it as if it was a function call (is just syntax sugar).

```
trait Dummy {  
  val value : Int  
}  
  
object Dummy {  
  class DummyImpl(x:Int) extends Dummy  
    { val value = x }  
  def apply(x : Int) = new DummyImpl(x)  
}
```

```
scala> val d = Dummy(1)  
d: Dummy.DummyImpl = Dummy$DummyImpl@71bce710
```



Expansion of Function values

An anonymous function such as

```
(x : Int) => x * x
```

is expanded to

```
{class AnonFun extends Function1[Int,Int] {  
    def apply(x : Int) = x * x  
}  
new AnonFun()}
```

or, shorter, using anonymous class syntax:

```
new Function1[Int,Int]{  
    def apply(x : Int) = x * x  
}
```



Expansion of Function Calls

A function call, such as $f(a,b)$, where f is a value of some class type, is expanded to

`f.apply(a,b)`

So the OO-translation of

```
val f = (x : int) => x * x
f(7)
```

would be

```
val f = new Function[Int,Int]{
  def apply(x : Int) = x * x
}
f.apply(7)
```



Functions and Methods

Note that a method such as

```
def f(x : Int) : Boolean = ???
```

is not itself a function value.

But if `f` is used in a place where a Function type is expected, it is converted automatically to the function value

```
(x : Int) => f(x)
```

or, expanded

```
new Function1[Int, Boolean]{  
  def apply(x : Int) = f(x)  
}
```



How does Scala evaluate expressions?

- ▶ Strict
- ▶ Lazy



Dynamic semantics - example

```
def cyclicList(x:Int): List[Int] = {  
  x :: cyclicList(x-1)  
}
```

```
scala> cyclicList(Int.MaxValue)  
java.lang.StackOverflowError  
at .cyclicList(<console>:11)
```



Dynamic semantics - example

```
def cyclicStream(x:Int): Stream[Int] = {  
  x #:: cyclicStream(x-1)  
}
```

```
scala> cyclicStream(Int.MaxValue)  
res1: Stream[Int] = Stream(2147483647, ?)
```



Dynamic semantics

```
def CallByValue(x: Int) = {  
  println("x1=" + x)  
  println("x2=" + x)  
}  
  
def callByName(x: => Int) = {  
  println("x1=" + x)  
  println("x2=" + x)  
}  
  
def zeroArityFunction(x: () => Int) = {  
  println("x1=" + x())  
  println("x2=" + x())  
}
```



Dynamic semantics

```
def something() : Int = {  
  println("calling something")  
  1  
}
```



Dynamic semantics - Call by value

```
def something() : Int = {  
  println("calling something")  
  1  
}  
  
def CallByValue(x: Int) = {  
  println("x1=" + x)  
  println("x2=" + x)  
}  
  
scala> CallByValue(something())  
calling something  
x1=1  
x2=1
```



Dynamic semantics - Call by name

```
def something() : Int = {  
  println("calling something")  
  1  
}  
  
def callByName(x: => Int) = {  
  println("x1=" + x)  
  println("x2=" + x)  
}  
  
scala> callByName(something())  
...
```



Dynamic semantics - Call by name

```
def something() : Int = {  
  println("calling something")  
  1  
}  
  
def callByName(x: => Int) = {  
  println("x1=" + x)  
  println("x2=" + x)  
}  
  
scala> callByName(something())  
calling something  
x1=1  
calling something  
x2=1
```



Dynamic semantics - 0-arity-function

```
def something() : Int = {  
  println("calling something")  
  1  
}  
  
def zeroArityFunction(x: () => Int) = {  
  println("x1=" + x())  
  println("x2=" + x())  
}  
  
scala> zeroArityFunction(something())  
<console>:14: error: type mismatch;  
found    : Int  
required: () => Int
```



Dynamic semantics - 0-arity-function

```
def something() : Int = {  
  println("calling something")  
  1  
}  
  
def zeroArityFunction(x: () => Int) = {  
  println("x1=" + x())  
  println("x2=" + x())  
}  
  
scala> zeroArityFunction(() => something())  
...
```



Dynamic semantics - 0-arity-function

```
def something() : Int = {  
  println("calling something")  
  1  
}  
  
def zeroArityFunction(x: () => Int) = {  
  println("x1=" + x())  
  println("x2=" + x())  
}  
  
scala> zeroArityFunction(() => something())  
calling something  
x1=1  
calling something  
x2=1
```



Dynamic semantics - Lazy keyword

```
scala> lazy val number1 = {  
  | println("I am a number ");  
  | 13  
  | }  
number1: Int = <lazy>
```

```
scala> val number2 = {  
  | println("I am a number: ");  
  | 20  
  | }  
I am a number:  
number2: Int = 20
```



Dynamic semantics - Lazy keyword

```
scala> lazy val number1 = {  
    | println("I am a number "); 13}  
number1: Int = <lazy>
```

```
scala> number1  
I am a number  
res0: Int = 13
```

```
scala> number1  
...
```



Dynamic semantics - Lazy keyword

```
scala> lazy val number1 = {  
    | println("I am a number "); 13 }  
number1: Int = <lazy>
```

```
scala> number1  
I am a number  
res0: Int = 13
```

```
scala> number1  
res2: Int = 13
```



Dynamic semantics - Lazy keyword

```
scala> lazy val number1 = {  
    | println("I am a number ");  
    | 13  
    | }  
number1: Int = <lazy>
```

```
scala> val number2 = {  
    | println("I am a number: "); 20 }  
I am a number:  
number2: Int = 20
```

```
scala> number2  
res1: Int = 20
```



Dynamic semantics - Lazy keyword

```
scala> lazy val number1 = {  
  | println("I am a number "); 13}  
number1: Int = <lazy>
```

```
scala> val number2 = {  
  | println("I am a number: "); 20}  
I am a number:  
number2: Int = 20
```

```
scala> number2  
res1: Int = 20
```

```
scala> number2  
res3: Int = 20
```



Dynamic semantics - summary

- ▶ Call by value
- ▶ Call by name
- ▶ Call by need



Recap

- ▶ Scala combines **O**bject **O**riented with **F**unctional.
- ▶ But its approach differs both from Java and Haskell quite a lot.
- ▶ It is a very interesting language.
- ▶ The different features we have shown are **composable** and allow to create awesome new patterns.
- ▶ Scala uses the Java Virtual Machine to execute the code.
- ▶ All standard libraries from Java are available.



Thank you!

Any questions?

