Scala on the spotlight

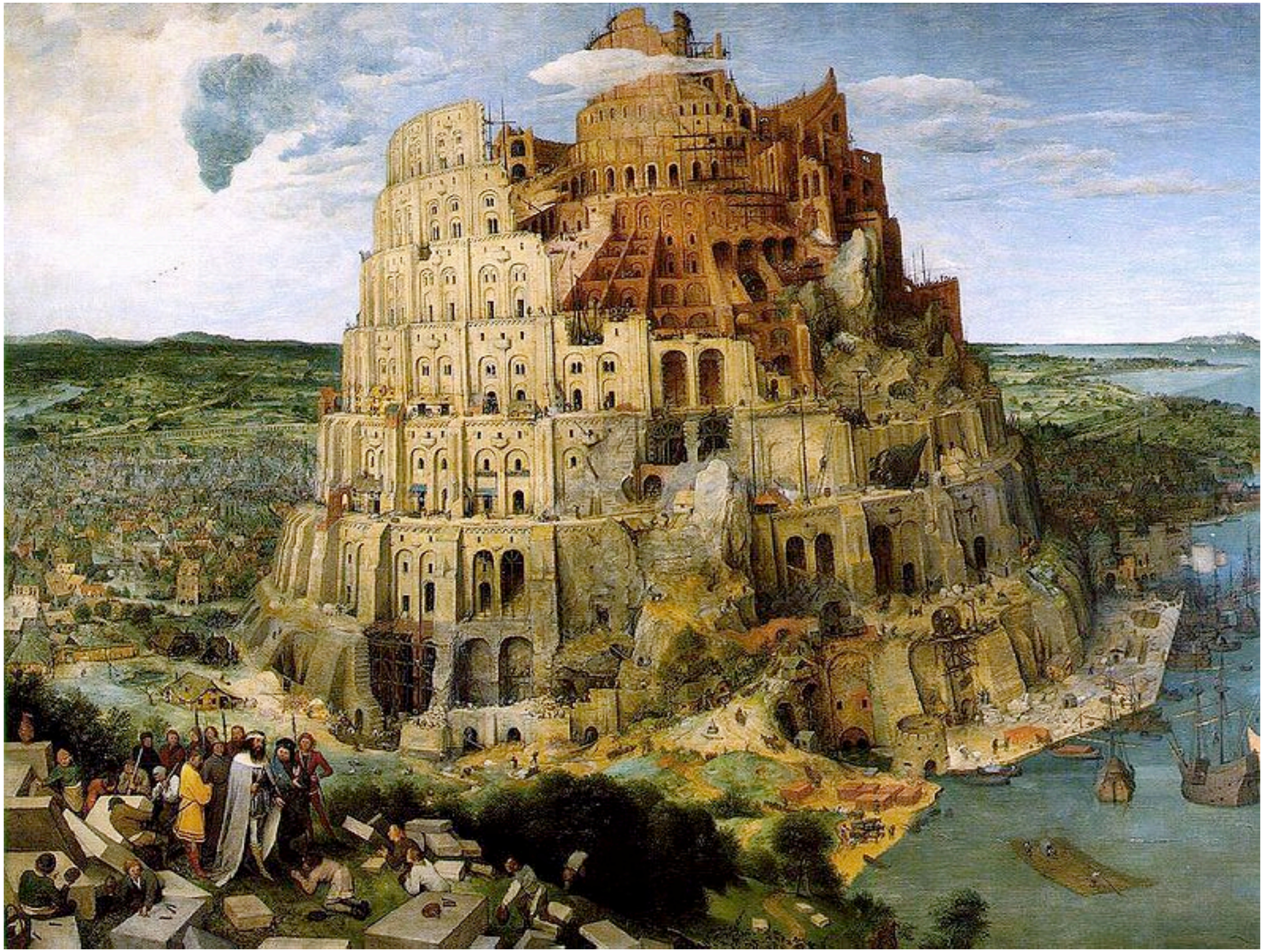
On the way to cerro Provincia, May 2009

Jacques Noyé
Ecole des Mines de Nantes

The history of programming languages

- The reign of imperative programming and (imperative) object-oriented programming
- Some underground streams: functionnal programming and logic/constraint programming
- A new major language every ten years
- The end of Java's reign: 2007?

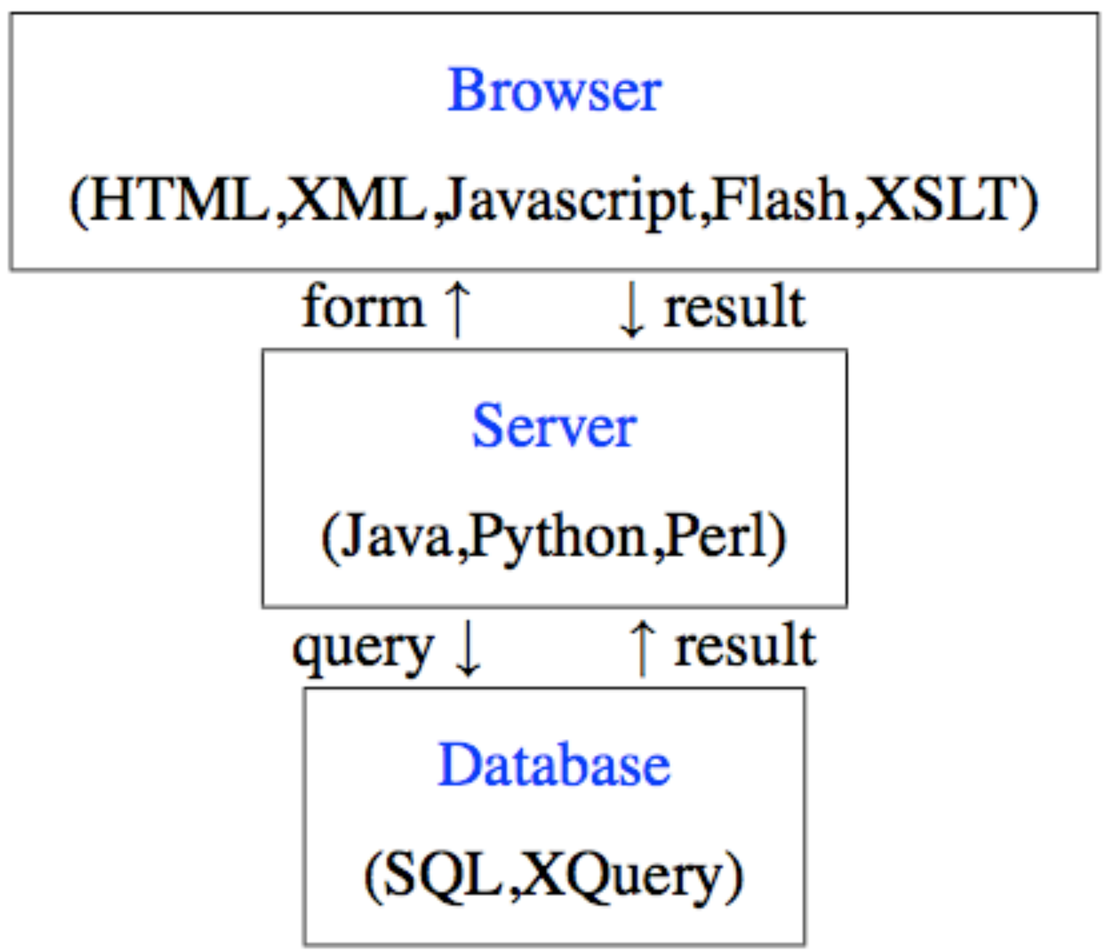
The current landscape



Pieter Brueghel the Elder (1563). Wikimedia Commons

The current landscape

Three-tier model



Talk on Links, Philip Wadler, Feb 2005

The good, the bad, and the ugly

- Good: every language is (hopefully) very well tuned to a specific domain.
- Bad: this is a major source of trouble as soon as one (person/program) has to work with several languages.
- Good individual parts, fragile whole.
- Model-driven engineering adds a layer of complexity on top of this.

The alternative: better general-purpose languages

- Support both programming in the small and programming in the large
- Support application-specific needs within the general purpose language (extensibility)
- Scala is a very interesting attempt at this
- Could it be the new general-purpose language of the decade?

Scala
(Scalable Language)

- Developped by Martin Odersky *et al.* at the Ecole Polytechnique Fédérale de Lausanne (Switzerland)
- Start: 2001
- First release: end 2003
- The buzz: April 2009 (adoption by Twitter)

Scala in a nutshell

- Multiparadigm: FP + OOP + COP (Composition- or Component-Oriented)
- Emphasis on scalability and extensibility
- Powerful type system, type inference
- Concise smart syntax (not mere syntactic sugar)
- Completely interoperable with Java
- A lot of goodies: top-level loop, XML, concurrency

Scala's roots

- Surface syntax: Java, C#
- Implementation: Java
- Uniform object model: Smalltalk
- Universal nesting: Algol, Simula, Beta
- Uniform access principle: Eiffel
- Fonctional programming: ML family, Haskell
- Concurrency: Erlang
- OOP+FP: OCaml, PLT-Scheme, O'Haskell

What makes Scala scalable?

- The main factor is the integration of FP and OO
  - FP safely composes (closed) parts: higher-order functions, algebraic types, and pattern matching
  - OO flexibly extends (open) parts: dynamic configurations of objects, classes as partial abstractions, subtyping and inheritance

The ν*Obj* Calculus [ECOOP2003]

Syntax

x, y, z	Name		
l, m, n	Term label	L, M, N	Type label
s, t, u ::=	Term	S, T, U ::=	Type
x	Variable	$p.\mathbf{type}$	Singleton
$t.l$	Selection	$T \bullet L$	Type selection
$\nu x \leftarrow t\ ;\ u$	New object	$\{x \mid \overline{D}\}$	Record type (=:: R)
$[x : S \mid \overline{d}]$	Class template	$[x : S \mid \overline{D}]$	Class type
$t\ \&_S\ u$	Composition	$T\ \&\ U$	Compound type
d ::=	Definition	D ::=	Declaration
$l = t$	Term definition	$l : T$	Term declaration
$L \preceq T$	Type definition	$L \preceq: T$	Type declaration
p ::=	Path	$\preceq:$ ::=	Type binder
$x \mid p.l$		$=$	Type alias
v ::=	Value	$\prec$	New type
$x \mid [x : S \mid \overline{d}]$		$<:$	Abstract type
		$\preceq$ ::=	Concrete type binder
		$= \mid \prec$	

Structural Equivalence α-renaming of bound variables x, plus

(extrude) $e\langle \nu x \leftarrow t\ ;\ u \rangle \equiv \nu x \leftarrow t\ ;\ e\langle u \rangle$ **if** $x \notin fn(e), bn(e) \cap fn(x, t) = \emptyset$

Reduction

(select) $\nu x \leftarrow [x : S \mid \overline{d}, l = v]\ ;\ e\langle x.l \rangle \rightarrow \nu x \leftarrow [x : S \mid \overline{d}, l = v]\ ;\ e\langle v \rangle$ **if** $bn(e) \cap fn(x, v) = \emptyset$

(mix) $[x : S_1 \mid \overline{d}_1]\ \&_S\ [x : S_2 \mid \overline{d}_2] \rightarrow [x : S \mid \overline{d}_1 \uplus \overline{d}_2]$

where evaluation context

$e ::= \langle\rangle \mid e.l \mid e\ \&_S\ t \mid t\ \&_S\ e \mid \nu x \leftarrow t\ ;\ e \mid \nu x \leftarrow e\ ;\ t \mid \nu x \leftarrow [x : S \mid \overline{d}, l = e]\ ;\ t$

Fig. 1. The νObj Calculus

FP in Scala

```
bash-3.2$ scala
Welcome to Scala version 2.7.4.final (Java HotSpot(TM)
64-Bit Server VM, Java 1.6.0_07).
Type in expressions to have them evaluated.
Type :help for more information.
scala> val x = 1
x: Int = 1
scala> val y = 2
y: Int = 2
scala> def add(x: Int, y: Int) = x + y
add: (Int,Int)Int
scala> val z = add(x, y)
z: Int = 3
```

First-class functions

```
scala> val inc = (x: Int) => x + 1
inc: (Int) => Int = <function>

scala> inc(10)
res20: Int = 11
```

First-class functions

```
scala> val add = (x: Int) => ((y: Int) => x + y)
add: (Int) => (Int) => Int = <function>

scala> val inc = add(1)
inc: (Int) => Int = <function>

scala> inc(10)
res6: Int = 11
```

Partially Applied Functions

```
scala> def add(x: Int)(y: Int) = x + y
add: (Int)(Int)Int

scala> val inc = add(1)_
inc: (Int) => Int = <function>

scala> inc(2)
res4: Int = 3
```

Partially Applied Functions

```
scala> def add(x: Int, y: Int) = x + y
add: (Int,Int)Int

scala> def inc(x: Int) = add(x, _: Int)
inc: (Int)(Int) => Int

scala> def inc(x: Int) = add(_: Int, x)
inc: (Int)(Int) => Int
```

Imperative features

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

scala> x = 2
<console>:5: error: reassignment to val
       x = 2
         ^

scala> var x = 1
x: Int = 1

scala> x = 2
x: Int = 2
```

Imperative features

No semicolon !

```
scala> val inc = (x: Int) => {
     | println("inc(" + x + ")")
     | x + 1
     | }
inc: (Int) => Int = <function>

scala> inc(10)
inc(10)
res21: Int = 11
```

Closures

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

scala> val addMore = (x: Int) => x + more
addMore: (Int) => Int = <function>

scala> addMore(10)
res26: Int = 11

scala> val more = 2
more: Int = 2

scala> addMore(10)
res27: Int = 11
```

Closures

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

scala> val addMore = (x: Int) => x + more
addMore: (Int) => Int = <function>

scala> addMore(10)
res23: Int = 11

scala> more = 2
more: Int = 2

scala> addMore(10)
res25: Int = 12
```

Lists and pattern matching

```
scala> val l = 1 :: 2 :: Nil
l: List[Int] = List(1, 2)

scala> val h :: tl = List(1)
h: Int = 1
tl: List[Int] = List()

scala> def append[T](xs: List[T], ys: List[T]): List[T]
=
  xs match {
    case List() => ys
    case x :: xs1 => x :: append(xs1, ys)
  }
```

Lists and pattern matching

```
scala> val l = 1 :: 2 :: Nil
l: List[Int] = List(1, 2)

scala> val h :: tl = List(1)
h: Int = 1
tl: List[Int] = List()

scala> def append[T](xs: List[T], ys: List[T]): List[T]
=
  xs match {
    case List() => ys
    case x :: xs1 => x :: append(xs1, ys)
  }
```

Lists and pattern matching

```
scala> val l = 1 :: 2 :: Nil
l: List[Int] = List(1, 2)

scala> val h :: tl = List(1)
h: Int = 1
tl: List[Int] = List()

scala> def append[T](xs: List[T], ys: List[T]): List[T]
=
  xs match {
    case List() => ys
    case x :: xs1 => x :: append(xs1, ys)
  }

scala> append(l, List(3, 4))
res11: List[Int] = List(1, 2, 3, 4)
```

A thrill

```
scala> val l = 1 :: 2 :: Nil
l: List[Int] = List(1, 2)

scala> def append[T](xs: List[T], ys: List[T]): List[T]
=
  xs match {
    case List() => ys
    case x :: xs1 => x :: append(xs1, ys)
  }

scala> append(l, List(3, 4))
res11: List[Int] = List(1, 2, 3, 4)

scala> append(List(1, 2, 3), List("4"))
res12: List[Any] = List(1, 2, 3, 4)
```

Higher-order functions

```
scala> def map[T, S](xs: List[T], f: T => S): List[S] =
  xs match {
    case Nil => Nil
    case x :: xs1 => f(x) :: map(xs1, f)
  }
map: [T,S](List[T],(T) => S)List[S]

scala> map(List(1, 2, 3), inc)
res20: List[Int] = List(2, 3, 4)
```

Maps

```
scala> val traduit = Map("j'ai" -> "tengo", "tu as" ->
"tienes", "il a" -> "tiene", "elle a" -> "tiene", "nous
avons" -> "tenemos", "vous avez" -> "tenéis", "ils ont"
-> "tienen")
traduit:
scala.collection.immutable.Map[java.lang.String,java.la
ng.String] = Map(elle a -> tiene, il a -> tiene, ils
ont -> tienen, nous avons -> tenemos, tu as -> tienes,
j'ai -> tengo, vous avez -> tenéis)

scala> traduit("elle a")
res21: java.lang.String = tiene
```

Local/Nested Functions

```
scala> def exists[T](xs: Array[T], p: T => boolean) = {
  var i: Int = 0
  while (i < xs.length && !p(xs(i))) i = i + 1
    i < xs.length
}
exists: [T](Array[T],(T) => boolean)Boolean

scala> def forall[T](xs: Array[T], p: T => boolean) = {
  def not_p(x: T) = !p(x)
  !exists(xs, not_p)
}
forall: [T](Array[T],(T) => boolean)Boolean
```

OO

```
scala> class Rational(n: Int, d: Int) {
  require(d != 0)
  val numer: Int = n
  val denom: Int = d
  def this(n: Int) = this(n, 1)
  override def toString = numer + "/" + denom
  def add(that: Rational): Rational =
    new Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom
    )
}
defined class Rational
```

OO

parameters of primary constructor

```
scala> class Rational(n: Int, d: Int) {
  require(d != 0)
  val numer: Int = n
  val denom: Int = d
  def this(n: Int) = this(n, 1)
  override def toString = numer + "/" + denom
  def add(that: Rational): Rational =
    new Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom
    )
}
defined class Rational
```

OO

precondition

```
scala> class Rational(n: Int, d: Int) {
  require(d != 0)
  val numer: Int = n
  val denom: Int = d
  def this(n: Int) = this(n, 1)
  override def toString = numer + "/" + denom
  def add(that: Rational): Rational =
    new Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom
    )
}
defined class Rational
```

OO

this is a functional object

```
scala>                 nt, d: Int) {
  requi
  val numer: Int = n
  val denom: Int = d
  def this(n: Int) = this(n, 1)
  override def toString = numer + "/" + denom
  def add(that: Rational): Rational =
    new Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom
    )
}
defined class Rational
```

OO

```
scala> class Rational(
  require(d != 0)
  val numer: Int = n
  val denom: Int = d
  def this(n: Int) = this(n, 1)
  override def toString = numer + "/" + denom
  def add(that: Rational): Rational =
    new Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom
    )
}
defined class Rational
```

secondary constructor

OO

```
scala> class Rational(n: Int, d: Int) {
  require(d != 0)
  val numer: Int = n
  val denom
  def this(                s(n, 1)
  override def toString = numer + "/" + denom
  def add(that: Rational): Rational =
    new Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom
    )
}
defined class Rational
```

mandatory

OO

```
scala> class Rational(n: Int, d: Int) {
  require(d != 0)
  val numer: Int = n
  val denom: Int = d
  def this(n: Int) = this(n, 1)
  override def toString = numer + "/" + denom
  def add(that: Rational): Rational =
    new Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom
    )
}
defined class Rational
```

parameterless method (UAP)

Keyword `override`

- Avoid accidental mistakes
  - Silent overriding of inherited method
  - Change of parameter in a superclass: overriding is silently turned in overloading

OO

```
scala> new Rational(1).add(new Rational(1, 3))
res14: Rational = 4/3

scala> new Rational(1, 0)
java.lang.IllegalArgumentException: requirement failed
   at scala.Predef$.require(Predef.scala:107)
   at ...
```

Using the compiler

- Defining a Scala entry point as a *standalone object*

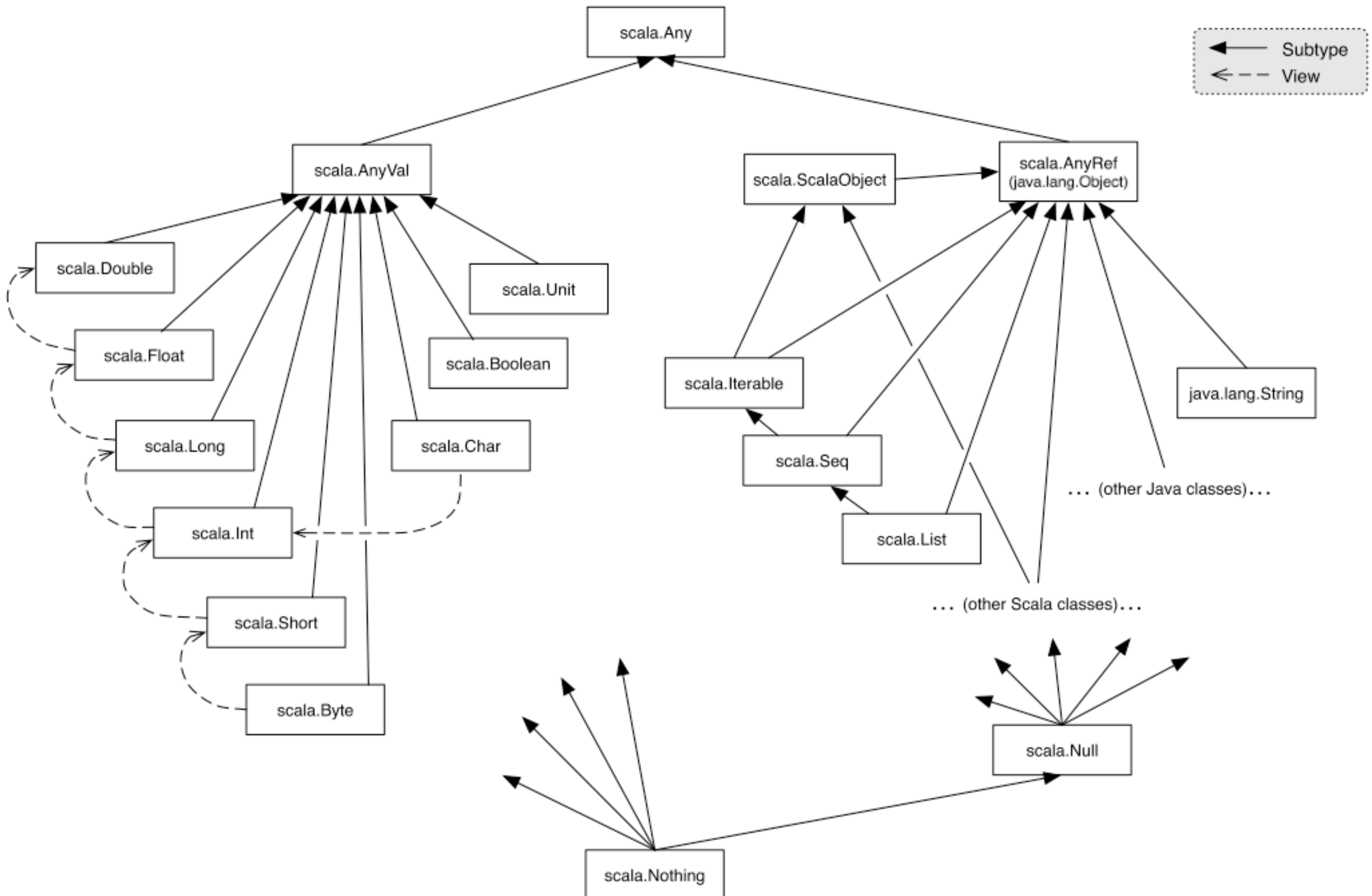
```
package rational

object Main {
  def main(args: Array[String]) {
    println(new Rational(1).add(new Rational(1, 3)))
  }
}
```

How does it blend?

- Scala is a pure OO language
  - Every value (eg numbers, functions) is an object
  - Every operation is a method call
  - There is no exception (eg no primitive types, no static methods)

The Scala hierarchy



From An Overview of the Scala Programming Language
Tech. Report LAMP-REPORT-2006-001

Numbers are objects

- `1 + 2` is equivalent to `(1).+(2)`
- `+` is a method of the class `Int`
- `+` is a legal identifier
- any identifier can be used as an operator: `"Hello" indexOf 'o'` is equivalent to `"Hello".indexOf('o')`

The example of `Rational`

```
class Rational(n: Int, d: Int) {
  ...
  def +(that: Rational): Rational =
    new Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom
    )
}
defined class Rational

scala> new Rational(1) + new Rational(1, 3)
res22: Rational = 4/3
```

Operators

- Simple rules to govern the precedence and priority of operators:
  - The precedence of an infix operator is determined by its first character (in accordance with the precedence of the usual operators)
  - Operators are left-associative except if they end with a colon (eg cons)
  - The receiver of a right-associative operator is its *right*-hand side operand

The example of cons

- `x :: y :: zs`

  is interpreted as

  `(zs.::(y)).::(x)`

- `::` is a method of the class `List`

The class `List`

- Defined as a *case* class

```
package scala
abstract class List[+T] {
  def isEmpty: Boolean
  def head: T
  def tail: List[T]
}
case object Nil extends List[Nothing] {
  ...
}
case class ::[T](h: T, tl: List[T]) extends List[T] {
  ...
}
```

Variance Annotations

- If `S <: T`, what do we want?
  - `List[S] <: List[T]`: covariance `List[+S]`
  - `List[S] :> List[T]`: contravariance `List[-S]`
  - `List[S]` and `List[T]` not comparable - nonvariance `List[T]` (default)
- Declaration-site variance, checked by the compiler

Variance made easy

- If a an apple is a fruit, what do we want to say?
  - covariance: a basket of apple is a basket of fruits
  - contravariance: a basket of fruits is a basket of apples
  - nonvariance: a basket of fruits and a basket of apples are not comparable

Function variance

- Function `S => T`, what are the annotations for `S` and `T`?
- Compare:
  - Here are some coins and buy some fruits.

  - Here is some money and buy some apples.

Function variance

- Function `S => T`, what are the annotations for `S` and `T`?
- Compare:
  - coins => fruits.

  - money => apples.

Case classes

- Syntactic convenience
  - Adds a factory method with the class (`new` not needed)
  - Parameters turned into fields
  - Creates methods `toString`, `hashCode`, and `equals`
- Supports pattern matching

The magic of cons

```
scala> 1 :: "1" :: '1' :: Nil
res13: List[Any] = List(1, 1, 1)
```

This is possible thanks to *bounded polymorphism*:

```
def ::[U >: T](x: U): List[U] = new scala.::(x, this)
```

The “constructor” `List`

- `List` cannot be the constructor of the class `List`
- `List(`*args*`)` is interpreted as a call `List.apply(`*args*`)` to the method `apply` of the *companion object* of the class `List` (works for any object):

```
apply [A](xs : A*) : List[A]
```

Functions are objects

- A function of type `S => T` is interpreted as an object of type `Function1[S, T]` with a method `apply`:

```
trait Function1[-S, +T]{
  def apply(x: S): T
}
```

- For instance, `(x: Int) => x + 1` is interpreted as:

```
new Function1[Int, Int]{
  def apply(x: Int) = x+1
}
```

Every operation is a method invocation

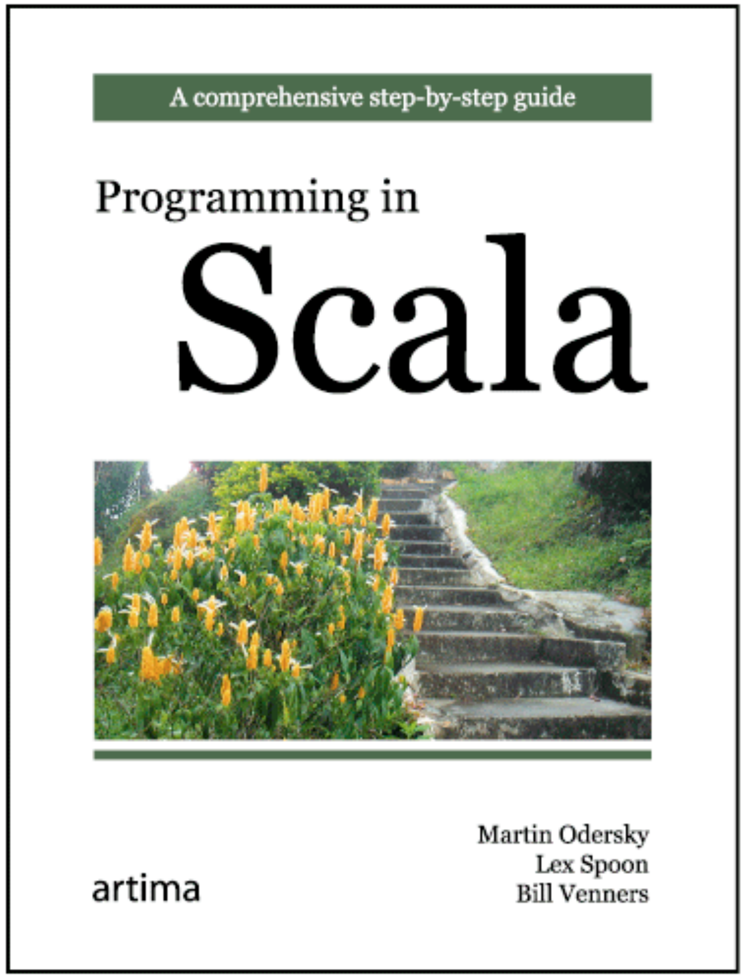
- The declaration of a variable **`var`** `x: T` defines a getter and a setter referencing a mutable memory cell not accessible directly from the source program:

```
def x: T
def x_= (newval: T): Unit
```

- A reference to `x` is interpreted as an invocation of the getter and an assignment of x as an invocation of setter.

Try it!

- http://www.scala-lang.org/
- On-line documentation
- Books
- Tools: emacs support, Eclipse plugin...



Source of most of the examples
The mistakes are mine

Scala as a composition language

Mount Everest North Face as seen from the path to the base camp, Tibet. Wikimedia Commons. GNU 1.2.