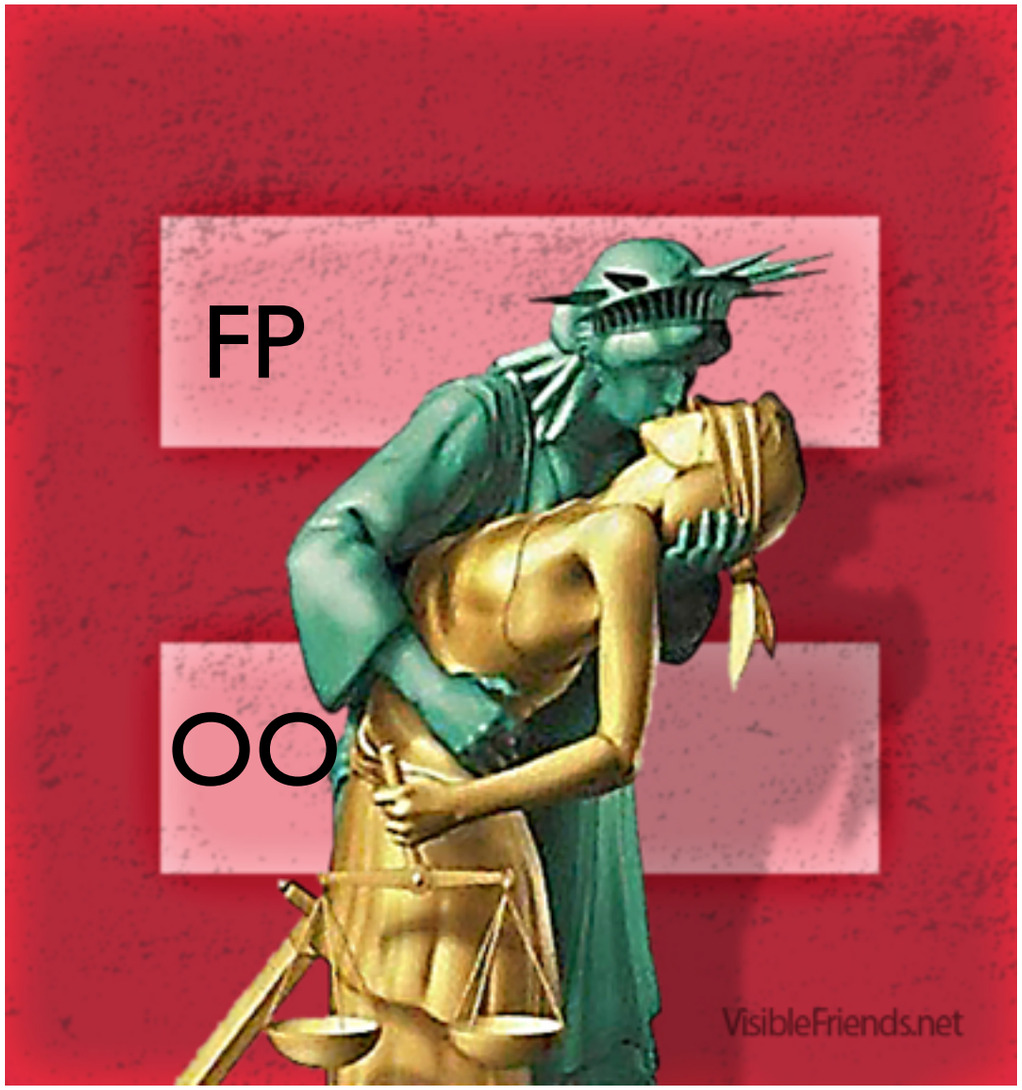
Scala, an equal marriage

classes/objects, functions and actors



Jacques.Noye@Mines-Nantes.fr

Introduction
Take 2

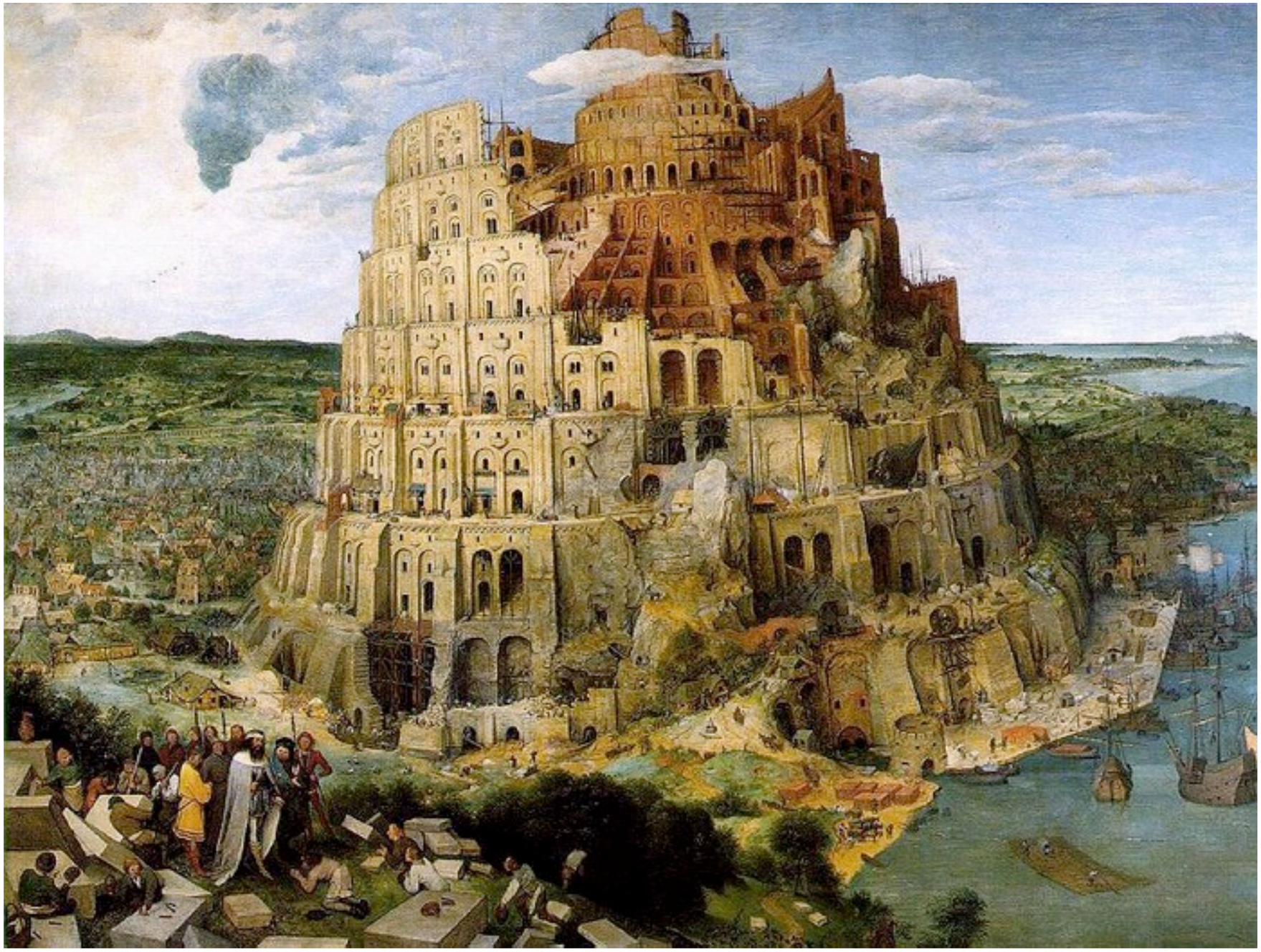
On the way to the glacier El Morado, Chili, November 2012

Jacques Noyé
École des Mines de Nantes

Programming languages history

- The long reign of imperative programming and lately of (imperative) object-oriented programming.
- Undercurrents: functional programming, logic and constraint programming
- A new major language every 10 years
- The end of Java's reign in 2007?

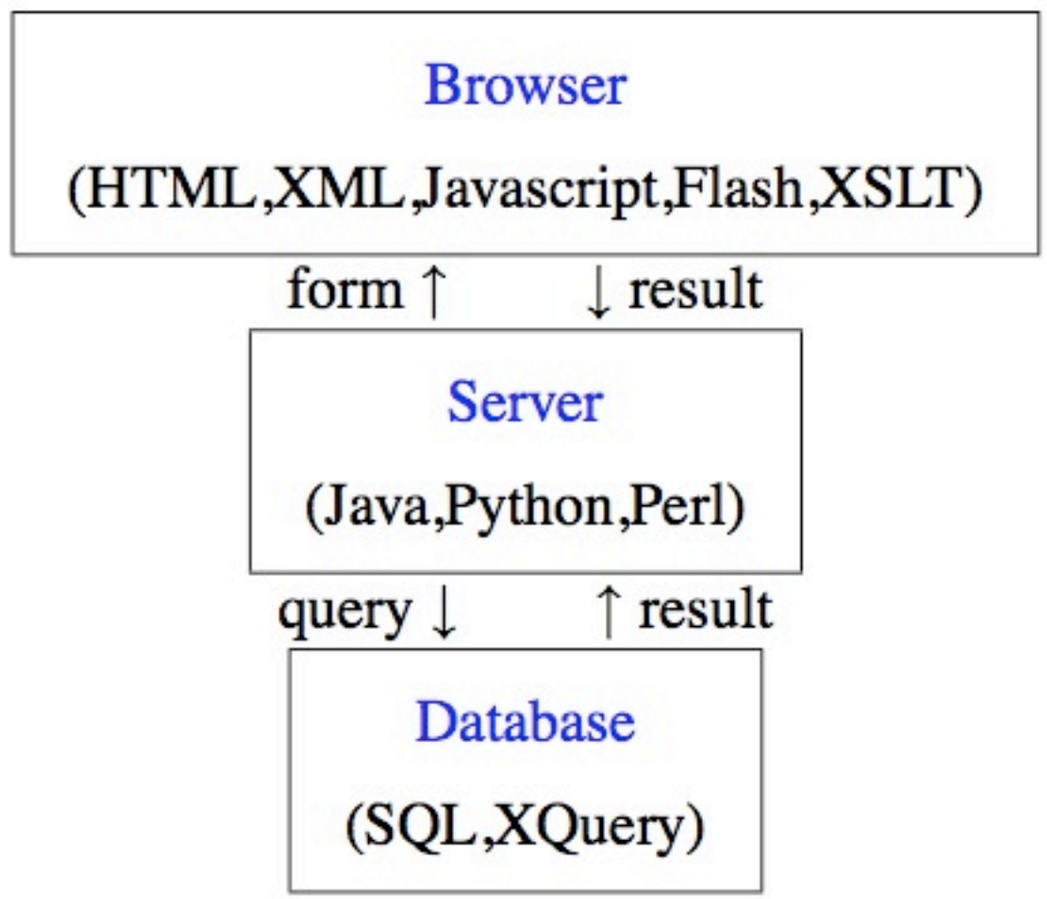
Today's landscape



Pieter Brueghel l'Ancien (1563). Wikimedia Commons

Today's landscape

Three-tier model



Présentation sur Links, Philip Wadler, février 2005

Pros and Cons

- Pros: each language is well-suited to its application domains.
- Cons: an issue when working with different languages. The individual components are fine. The assembly is fragile.
- Model-driven engineering adds to the complexity with another layer on top.

Another path: improve generic languages

- A path explored by Scala through support for
  - scalability, covering both small-scale and large-scale programming
  - extensibility, taking into account specific needs within a generic setting

Scala
(*Scalable Language*)

- Developed by the team of Martin Odersky at École Polytechnique Fédérale de Lausanne (Switzerland)
- Start: 2001, based on previous work on Funnel
- First distribution: end 2003
- The buzz: April 2009 (adoption by Twitter)
- Fall 2012: MOOC on Coursera

Martin Odersky, An Introduction to Functional Nets, Applied Semantics, LNCS 2395, 2002

Scala *in a nutshell*

- Multiparadigm : FP + OOP + COP (*Composition- or Component-Oriented*)
- Focus on scalability and extensibility
- Expressive type system with partial inference
- Concise smart syntax (beyond syntactic sugar)
- Interoperable with Java (virtual machine: the JVM)
- Many goodies: Read-Eval-Print Loop, concurrency...

The roots of Scala

- Syntax : Java, C#
- Implementation: Java
- Uniform object model: Smalltalk
- Universal nesting principle: Algol, Simula, Beta
- Uniform access principle: Eiffel
- Functional programming: ML family, Haskell
- Concurrency: Erlang
- OOP+FP : to compare with OCaml, Racket and O'Haskell

The heart of Scala

- The integration of OO and FP programming:
  - FP: safe assembly of closed components: higher-order functions, algebraic types and pattern matching.
  - OOP: flexible assembly of open components: dynamic object configuration, classes as partial abstractions with subtyping and inheritance.

Modules vs. classes: a summary

Things that work better with modules:

- Clear distinction between the values and the operations over them (better for n-ary operations).
- Stronger representation hiding.
- Functors as a general parameterization mechanism.
- Safety at the expense of flexibility.

Things that work better with objects and classes:

- Subtyping and row polymorphism at the level of values.
- Incremental programming via inheritance (open recursion, default implementations that can be overriden later).
- Flexibility at the expense of safety.

37

Xavier Leroy, Objects and Classes vs Modules in Objective Caml, 2003?

Modules and A.D.T.	Classes and objects
Protect against programmers	Trust programmers
Pessimistic mindset	Optimistic mindset
Educated terminology (signatures, structures, functors)	Common words (objects, classes, methods, to inherit)
Mathematical perspective (operations over abstract types; algebraic laws)	Physical/anthropomorphic perspective (send messages asking objects to do things)
Elitist discourse	Populist discourse
...	...

40

Xavier Leroy, Objects and Classes vs Modules in Objective Caml, 2003?

Two Worlds

Objects and modules have complementary strengths.

- Modules are good at *abstraction*.

  For instance: abstract types in SML signatures.
  (Object systems offer only crude visibility control through modifiers such as ***private*** or ***protected***).

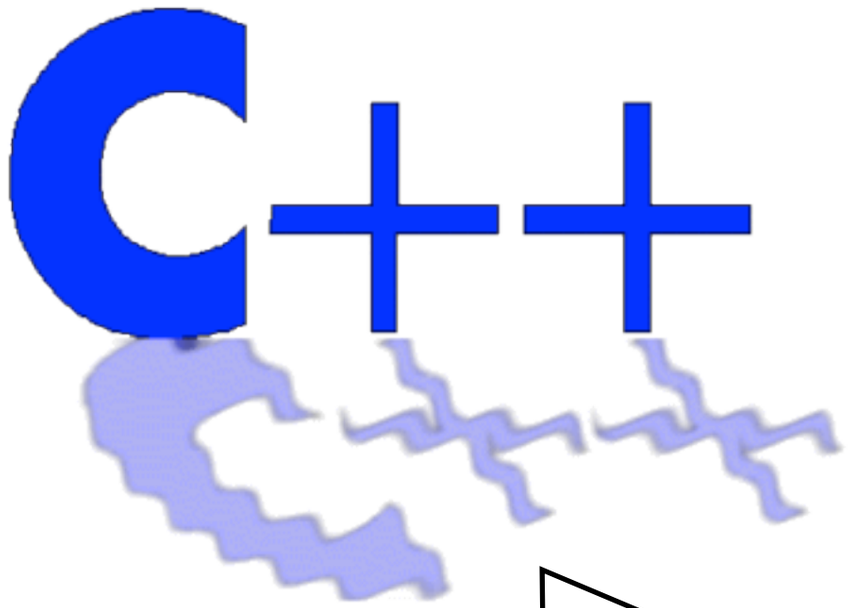
- Objects are good at *composition*.

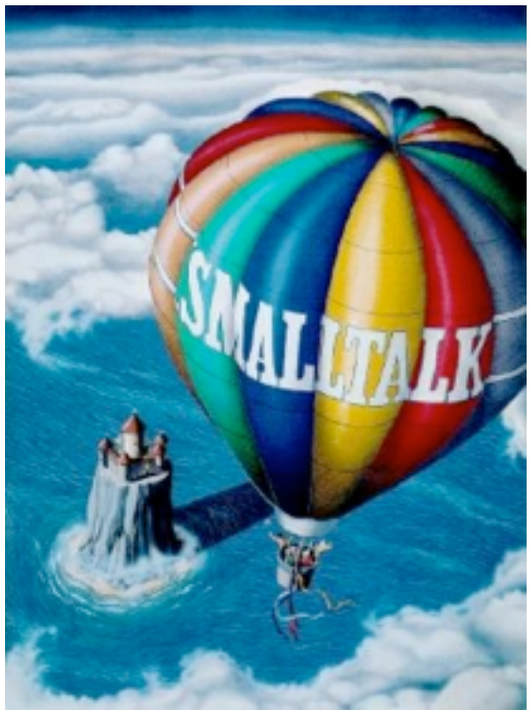
  For instance: Aggregation, recursion, inheritance, components as first-class values.
  (Only the first is supported by standard module systems).

Composition seems to be currently more popular than abstraction.

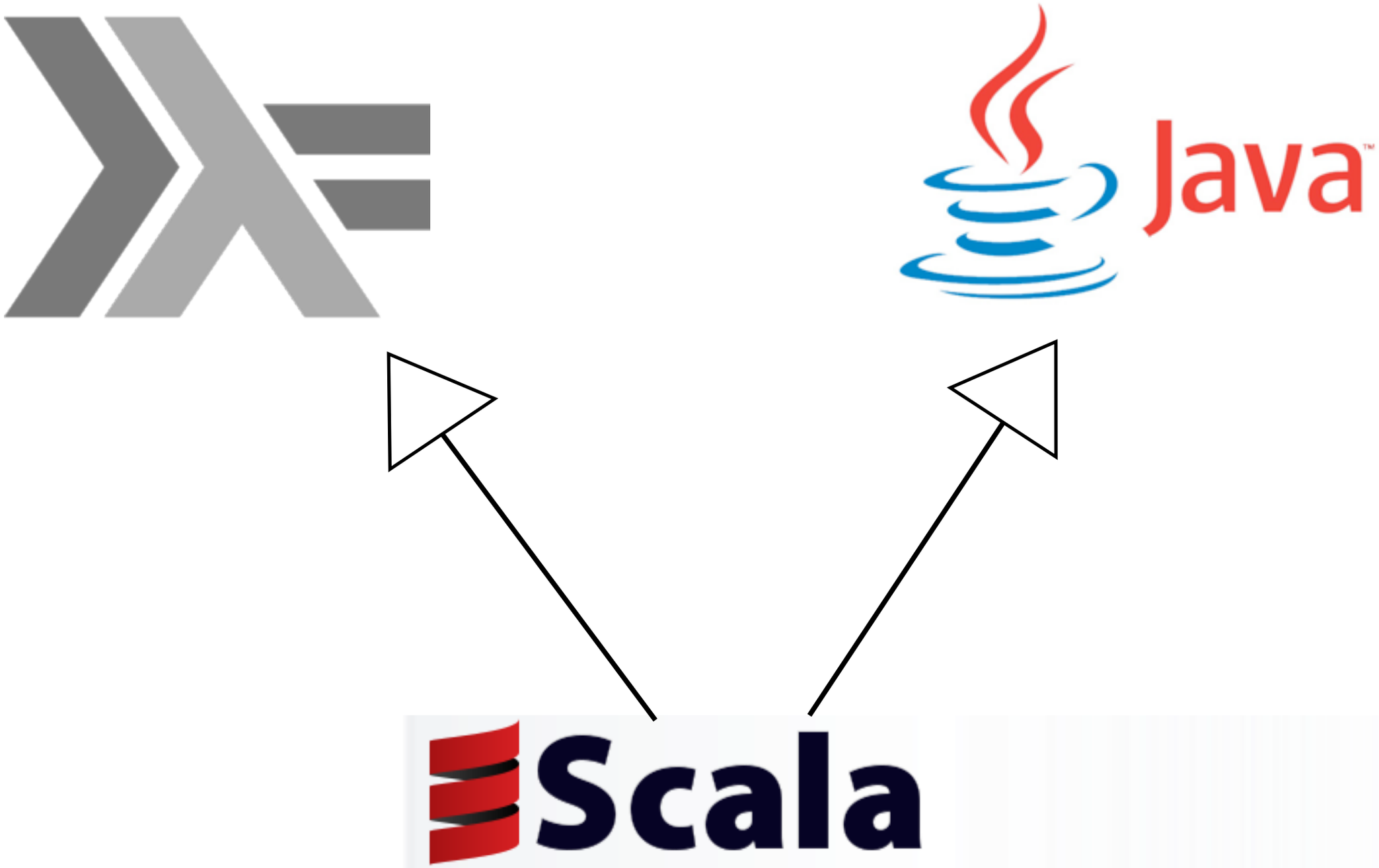
That's why most popular languages are based on object systems, even though it comes at a cost in the expressiveness of types.

2


C++


SMALLTALK


Java


Java
Scala

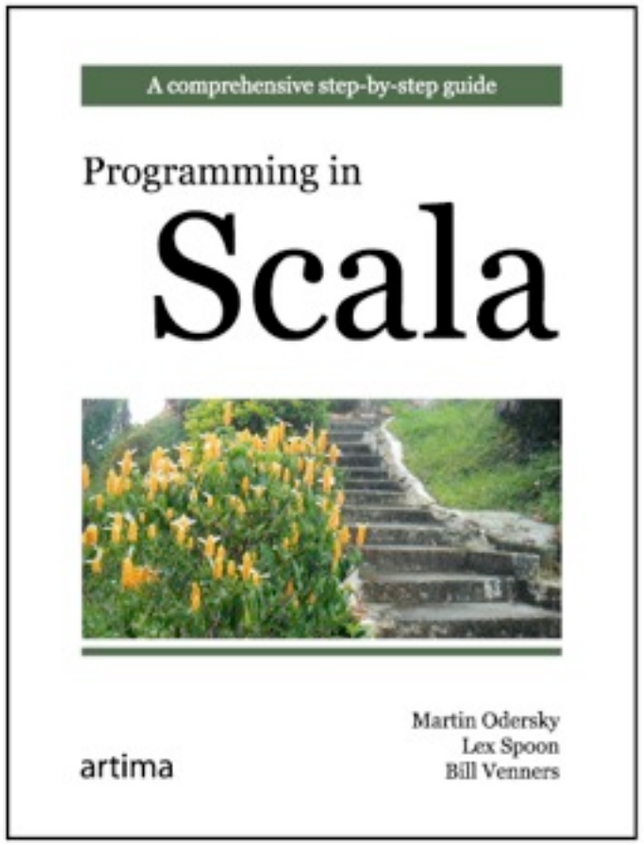

Java

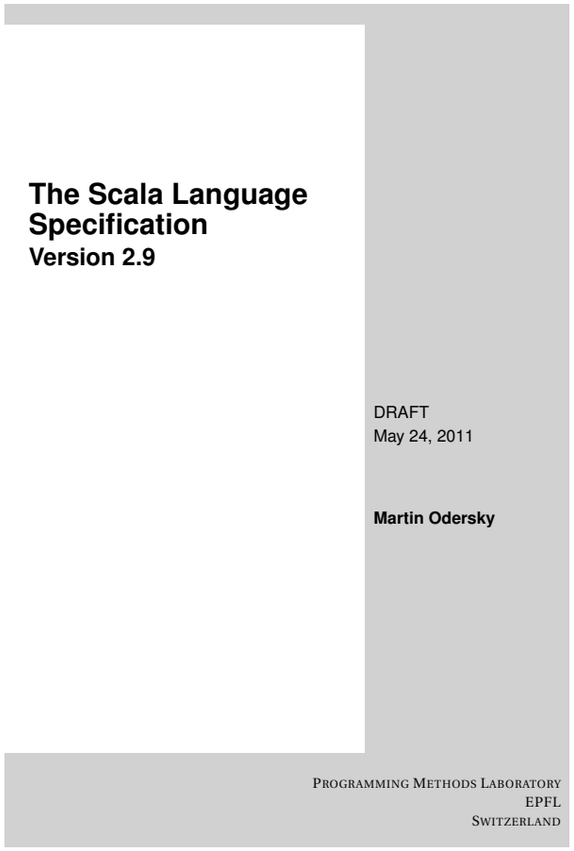

Scala

Outline

- Object-Oriented Programming
  - Functional objects
  - Imperative objects
  - Traits
- Functional (and OO) Programming
  - Functions and methods
  - Lists, pattern matching and types
  - Actors

My bookshelf





New: http://docs.scala-lang.org

(Functional) OO programming

Functional object

A functional (ie non-imperative) object is an object with immutable state

Simplest Example

r0 is immutable
(this does not mean
the object is)

```
scala> class Foo(){}
defined class Foo

scala> val r0 = new Foo(){};
r0: Foo = Foo@101873a6

scala> r0 = new Foo(){};
<console>:10: error: reassignment to val
       r0 = new Foo
          ^
```

Another example: Rationals

compare with Java:
`int n`

```
scala> class Rational(n: Int, d: Int)
defined class Rational

scala> val r1 = new Rational(2, 3)
r1: Rational = Rational@56781dd4
```

Primary constructor

parameters of
primary constructor

```
class Rational(n: Int, d: Int)
```

Method definition

```
scala> class Rational(n: Int, d: Int) {
  override def toString() = n + "/" + d
}
defined class Rational

scala> val r1 = new Rational(2, 3)
r1: main.Rational = 2/3
```

Overriding

mandatory

```
class Rational(n: Int, d: Int) {
  override def toString = n + "/" + d
}
```

- Avoids two problems:
  - Unfortunate *overriding* of inherited methods
  - Parameter modifications in a superclass: overriding is silently turned into *overloading*

Note: now current practice in Java, but using an annotation.

Uniform Access Principle

For a client:

```
scala> new Rational(2, 3).toString
res0: java.lang.String = 2/3
```

There is no difference between:

```
class Rational(n: Int, d: Int) {
  override def toString() = n + "/" + d
}
class Rational(n: Int, d: Int) {
  override val toString = n + "/" + d
}
```

When does the evaluation take place?

- `val`: initialization time
- `lazy val`: first use
- `def`: each use

Constructors

"constructor body"

```
scala> class Rational(n: Int, d: Int) {
  require(d != 0);
  def this(n: Int) = this(n, 1);
  // I could have put some expressions here
  override def toString = n + "/" + d
  // and here too
}
defined class Rational

scala> new Rational(1, 0)
java.lang.IllegalArgumentException: requirement failed
    at scala.Predef$.require(Predef.scala:221)
    at ...
```

Note: `java.lang`, `scala` and `Predef` are imported by default.

Semicolon inference

```
class Rational(n: Int, d: Int) {
  require(d != 0);
  def this(n: Int) = this(n, 1);
  override def toString = n + "/" + d;
}
```

An end of line is handled as a semicolon unless:

- The line ends with a token that can't terminate an instruction (period, infix operator...) or
- The following line starts with a token that can't start an instruction
- The line ends with unclose parentheses or brackets.

Secondary constructor

secondary constructor

calling the primary constructor

```
scala> class Rational(n: Int, d: In
  require(d != 0)
  def this(n: Int) = this(n, 1)
  override def toString = n + "/" + d
}

scala> new Rational(2)
res1: rational.Rational = 2/1
```

Instance variables

```
scala> class Rational(n: Int, d: Int) {
  require(d != 0)
  def this(n: Int) = this(n, 1)
  override def toString = n + "/" + d
  def add(that: Rational) =
    new Rational(n * that.d + that.n * d, d * that.d)
}
<console>:13: error: value d is not a member of Rational
          new Rational(n * that.d + that.n * d,
                                  ^
<console>:14: error: value d is not a member of Rational
                       d * that.d
                  ^
```

Instance variables

```
scala> class Rational(n: I
  require(d != 0)
  def this(n: Int) = this(n
  override def toString =
  def add(that: Rational) =
    new Rational(n * that.d + that.n * d, d * that.d)
}
<console>:13: error: value d is not a member of Rational
          new Rational(n * that.d + that.n * d,
                                  ^
<console>:14: error: value d is not a member of Rational
                       d * that.d
                 ^
```

Constructor parameters

≠

Instance variables

Instance variables
The verbose version

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

Note: instance variables must be initialized, otherwise they are considered as abstract.

Type inference

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

The types in grey are inferred

Instance variables
The concise version

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

Identifiers

Java:

- alphanumeric character sequences starting with a letter or underscore
- unicode characters

Scala: + operator character sequences

- ASCII characters: `!#$%&*+-/:<=>?@\^|~`
- unicode symbols (eg math symbols)
- sequences between ` (escaping keywords)

Infix operators

```
class Rational(val n: Int, val d: Int) {
  require(d != 0)
  def this(n: Int) = this(n, 1)
  override def toString = n + "/" + d
  def +(that: Rational) =
    new Rational(
      n * that.d + that.n * d,
      d * that.d)
}
...
// method call
new Rational(1).+(new Rational(1, 2))
// infix expression
new Rational(1) + new Rational(1, 2)
```

Infix operators

```
scala> 1 + 2
res0: Int = 3

scala> 1 .+(2)
res1: Int = 3

scala> 1.+(2) // deprecated
res2: Double = 3.0

scala> "Hello" indexOf 'o'
res3: Int = 4
```

+ is a method of class `Int`

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)

Unary postfix operators

```
// method call
new Rational(1, 2).toString
// postfix expression
import scala.language.postfixOps
new Rational(1, 2) toString
```

Prefix unary operators

+, –, !, ~ can be used as unary prefix operators.
The use of *op* is equivalent to a call to unary_*op*.

```
// method call
a.unary_-()
// postfix expression
-a
```

Precedence and associativity

- Precedence based on the first character of the operator (following usual rules on arithmetic and logic operators)
- Left-associativity except for operators terminating with a colon ":"
- The receiver of a right-associative operator is its *right*-hand side operand

Scala vs Java

```java
public class Rational {
  int n;
  int d;

  public Rational(int n, int d) {
    this.n = n;
    this.d = d;
  }
  public Rational(int n){
    this(n, 1);
  }
  @Override
  public String toString() {
    return n + "/" + d;
  }
  Rational add(Rational that) {
    return new Rational(n * that.d + that.n * d,
                        d * that.d);
  }
}
```

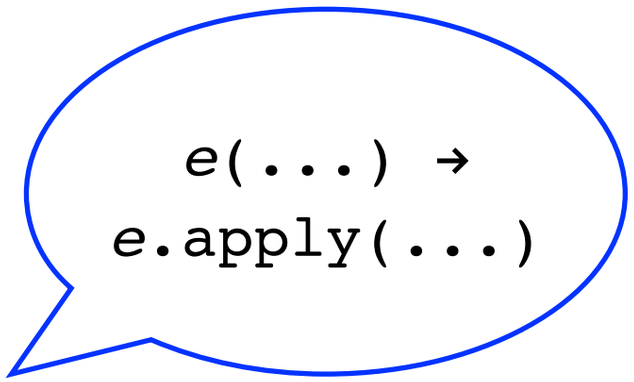
```scala
class Rational(val n: Int, val d: Int) {

  def this(n: Int) = this(n, 1)

  override def toString = n + "/" + d

  def +(that: Rational) =
    new Rational(n * that.d + that.n * d,
                 d * that.d)
}
```

Note: public is the default visibility.

Definition of a factory

Use a *companion object* (a singleton class) and a method `apply`

```
package rational

class Rational(val n: Int, val d: Int) {
  ...
}

object Rational {
  def apply(n: Int, d: Int) = new Rational(n, d)
  def apply(n: Int) = new Rational(n)
  def main(args: Array[String]) = {
    val r1 = Rational(1) + Rational(1, 2)
    println(r1)
  }
}
```



Notes: there is no "static" method; the method `main` can be used as an entry point; `Rational` (as a class) and `Rational` (as an object) belong to two different namespaces.

Implicits

your own type coercions

When type inference fails because

- the type provided is not ok (parameter, receiver)
- a curried method misses parameters (*implicit parameters*)

Which coercion?

- Method declared `implicit` taking the provided type and returning the required type
- whose name is in scope or associated to the provided or required type (via a companion object).
- There shouldn't be any ambiguity and a single conversion applies.

Example

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

object Rational {
  implicit def convert(that: Int) = new Rational(that)
}
...
  1 + new Rational(1) // Rational.convert(1) + new Rational(1)
```

Typing recursive methods

mandatory
return type

```
object Factorial {
  def fact(n: Int): Int =
    if (n <= 1) 1
    else n * fact(n-1)
}
```

Note: unlike in Java, (direct) tail recursion is optimized.

There are no statements

```
object Factorial {
  def fact(n: Int): Int =
    if (n <= 1) 1
    else n * fact(n-1)
}
```

a conditional is an expression

The evaluation of an expression returns a value

There are no statements

(The return of curly brackets)

```
object Factorial {
  def fact(n: Int): Int = {
    println("calling fac("+n+")");
    if (n <= 1) 1
    else n * fact(n-1)
  }
}
```

Evaluating a block/a sequence returns the value of its last expression

What does `println` return?

```
scala> val r = println("?")
?
r: Unit = ()
```

A side-effect returns `( )` of type `Unit` (not to be mistaken with `null` or `Nil`).

New members

- (singleton) classes and traits (*templates*)
- types, which can be abstract

```
class Abstract {
  type T
  def transform(x: T)
  val initial: T
  var current: T
}

class Concrete extends Abstract {
  type T = String
  def transform(x: String) = x + x
  val initial = "hi"
  var current = initial
}
```

The universal nesting principle

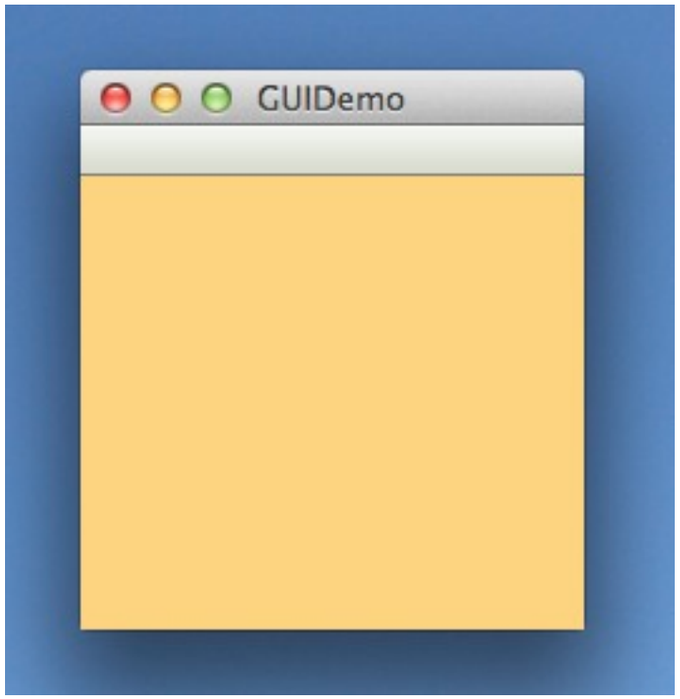
- Nesting a class in a class
- Nesting a package in a package
- Nesting a method in a method

```
def sqrt(x: Double) = {
  def sqrtIter(guess: Double, x: Double): Double =
    if (isGoodEnough(guess, x)) guess
    else sqrtIter(improve(guess, x), x)
  def improve(guess: Double, x: Double) =
    (guess + x / guess) / 2
  def isGoodEnough(guess: Double, x: Double) =
    abs(square(guess) - x) < 0.001
  sqrtIter(1.0, x)
}
```

Using Java in Scala

```
import javax.swing.{JFrame, JLabel, JMenuBar}
import java.awt.{BorderLayout, Color, Dimension}

object GUIDemo {
    def main(args: Array[String]) {
      val frame: JFrame = new JFrame("GUIDemo")
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)
      //Create the menu bar.  Make it have a green background.
      val greenMenuBar: JMenuBar = new JMenuBar()
      greenMenuBar.setOpaque(true)
      greenMenuBar.setBackground(new Color(154, 165, 127))
      greenMenuBar.setPreferredSize(new Dimension(200, 20))
      //Create a yellow label to put in the content pane.
      val yellowLabel: JLabel = new JLabel()
      yellowLabel.setOpaque(true)
      yellowLabel.setBackground(new Color(248, 213, 131))
      yellowLabel.setPreferredSize(new Dimension(200, 180))
      //Set the menu bar and add the label to the content pane.
      frame.setJMenuBar(greenMenuBar)
      frame.getContentPane().add(yellowLabel, BorderLayout.CENTER)

      frame.pack; frame.setVisible(true) }}
```

Note: it is also possible to use Scala in Java



Imperative Object-Oriented programming

Mutable state

Dropping a value

```
scala> class Counter {
  private var value = 0
  // def increment():Unit = { value += 1 }
  def increment() { value += 1 }
  def current() = value
}
defined class Counter

scala> val c = new Counter()
c: Counter = Counter@30479326

scala> c.increment()

scala> c.current()
res1: Int = 1
```

Reminder: default visibility is public and (concrete) fields must be initialized.

When to use parentheses?

- Recommendation:
  - Use parentheses for a *mutator* (state change)
  - Don't for an *accessor* (no state change)
- Can only be enforced on accessors (using a *parameterless method*)

```
class Counter {
  private var value = 0
  def increment() { value += 1 }
  def current = value
}
```

Defining a *property* in Java

```
public class Person {
  private int age;
  public int getAge() {
    return age;
  }
  public void setAge(int age) {
    this.age = age;
  }
}
```

Why not only a public field?

`getAge` and `setAge` define the *property* age of a person

Why?

- Modularization/evolution
- Encapsulation of constraints on the property

```
public class Person {
  private int age;
  public int getAge() {
    return age;
  }
  public void setAge(int newAge) {
    if (newAge > age)
      age = newAge;
  }
}
```

The age of a person cannot decrease

Note: in the absence of constraints/planned evolution, this is useless complication.

In Scala?

```
scala> class Person {
  var age = 0
}
defined class Person

scala> :javap Person
Compiled from "<console>"
public class Person extends
java.lang.Object{
    private int age;
    public int age();
    public void age_$eq(int);
    public Person();
}
```

getters and setters are implicitly defined

Implicit invocation of getters and setters

```
scala> val alice = new Person
alice: Person = Person@1ea3f672

scala> alice.age // invocation alice.age()
res10: Int = 0

scala> alice.age = 1 // invocation alice.age_=(1)
alice.age: Int = 1
```

Evolution

```
scala> class Person {
  private var privateAge = 0
  def age = privateAge // without parentheses
  def age_=(newAge: Int) =
    if (newAge > privateAge) privateAge = newAge
}
defined class Person

scala> val alice = new Person
alice: Person = Person@6f41ea57

scala> alice.age
res13: Int = 0

scala> alice.age = 1
alice.age: Int = 1

scala> alice.age = 0
alice.age: Int = 1
```

the interface hasn't changed

What happens with a private variable?

```
scala> class Person {
  private var age = 0
}
defined class Person

scala>:javap -private Person
Compiled from "<console>"
public class Person extends java.lang.Object{
    private int age;
    private int age();
    private void age_$eq(int);
    public Person();
}
```

the accessors are private

When it is a val?

```
scala> class Person {
 val age = 0
}
defined class Person
```

```
scala>:javap -private Person
Compiled from "<console>"
public class Person extends java.lang.Object{
    private int age;
    public int age();
    public Person();
}
```

there is no setter

And if there shouldn't be any access method?

```
scala> class Person {
  private[this] var age = 0
}
defined class Person

scala>:javap -private Person
Compiled from "<console>"
public class Person extends java.lang.Object{
    private int age;
    public Person();
}
```

private to the object

And if there shouldn't be any access method?

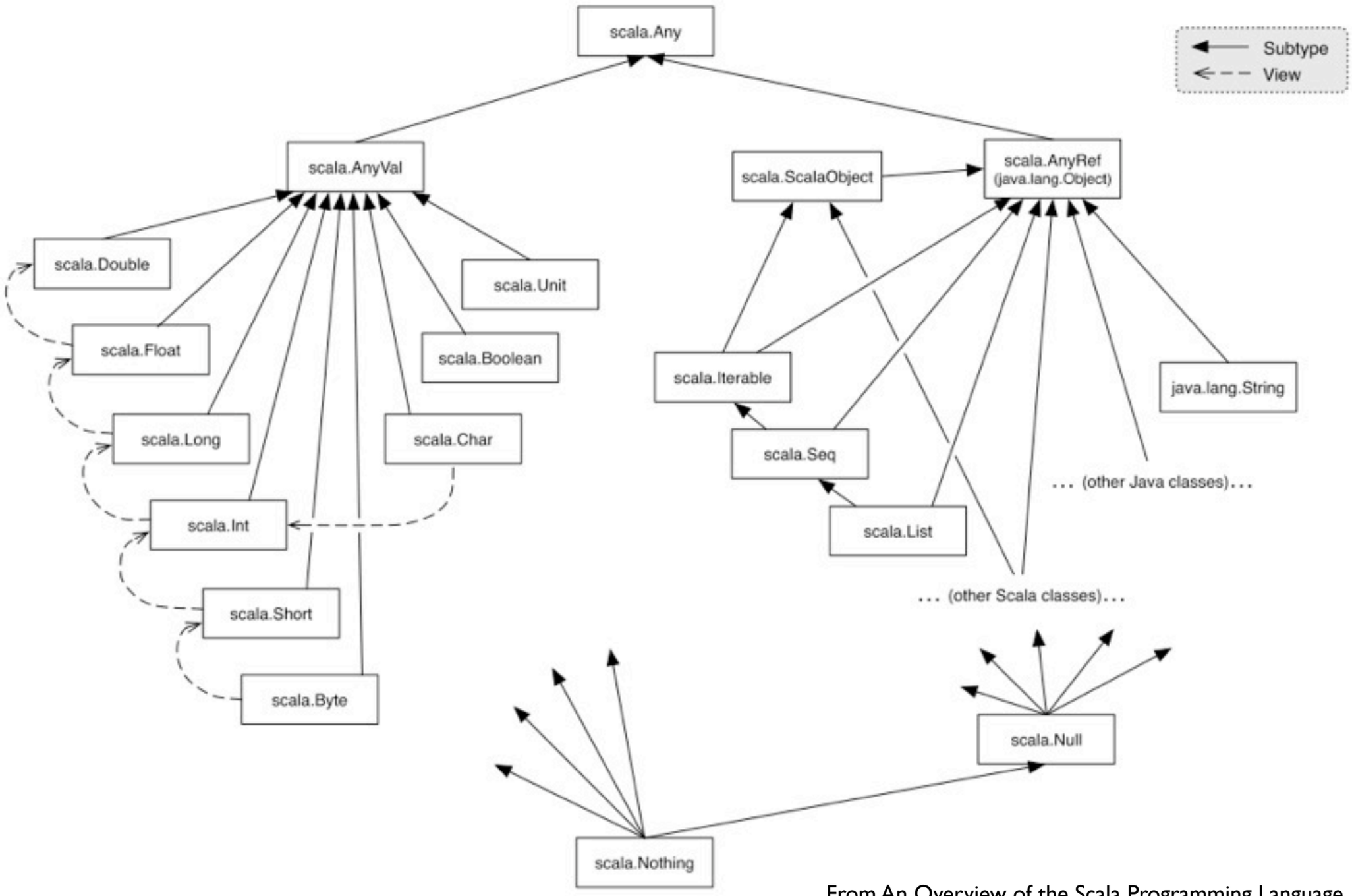
```
scala> class Person {
  private[this] val age = 0
}
defined class Person

scala>:javap -private Person
Compiled from "<console>"
public class Person extends java.lang.Object{
    private final int age;
    public Person();
}
```

Implementing a public property *p*

- Defining a mutable property:
  - **`var`** *`p`*
  - **`def`** *`p`*, **`def`** *`p_=`*
- Defining an immutable property:
  - **`val`** *`p`*
  - **`def`** *`p`*

Uniform access principle: seen from the client there is no difference between defining an instance variable and its accessors

Type hierarchy



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

- `Any` : the root, defines methods `isInstanceOf[`*T*`]`, `asInstanceOf[`*T*`]`, `toString`, equality (==) and hash codes.
- `AnyVal`: the root of predefined values (integrates primitive Java types and `Unit`)
- `AnyRef` is an alias for `java.lang.Object`
- `Null`: the type of `null` (a subtype of `AnyRef`, not `AnyVal`)
- `Nothing`: the smallest type, without any value

Horrors! `null` is back!

```
scala> var o: String = null
o: String = null

scala> o = “foo”
o: String = foo

scala> null
res0: Null = null

scala> null.isInstanceOf[AnyRef]
res1: Boolean = false
```

?!

Better not use it

```
scala> val area = Map("France" -> 551695, "Germany" ->
357021)
area: scala.collection.immutable.Map[String,Int] =
Map(France -> 551695, Germany -> 357021)

scala> area("United-Kingdom")
java.util.NoSuchElementException: key not found: United-
Kingdom
...
scala> area.get("United-Kingdom")
res1: Option[Int] = None

scala> area.get("Germany")
res2: Option[Int] = Some(357021)
```

Note: to be compared with the method `get` of `HashMap` in Java.

A use of `Nothing`

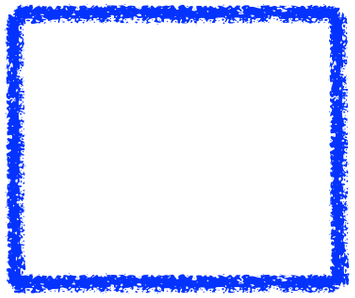
```
def divide(x: Int, y: Int): Int =
  if (y != 0) x / y // Int
  else throw new RuntimeException("division by zero") // Nothing
```

`Nothing <: Int`

- The method requires to return an integer, it is possible to provide a subtype (substitution principle).

- Replace `Int` by any type: `Nothing` must be the smallest type.

Traits



In French

What's a trait ?

- A program unit reusable through inheritance
- Combines features from other work: eg Bracha's *mixins* (Modula 3) and Schärli *et al.'* *traits* (Smalltalk).

A first simple trait (Java interface)

Java

```
public interface AbsIterator<T> {
  public boolean hasNext();
  public T next();
}
```

Scala

```
trait AbsIterator[T] {
  def hasNext: Boolean
  def next: T
}
```

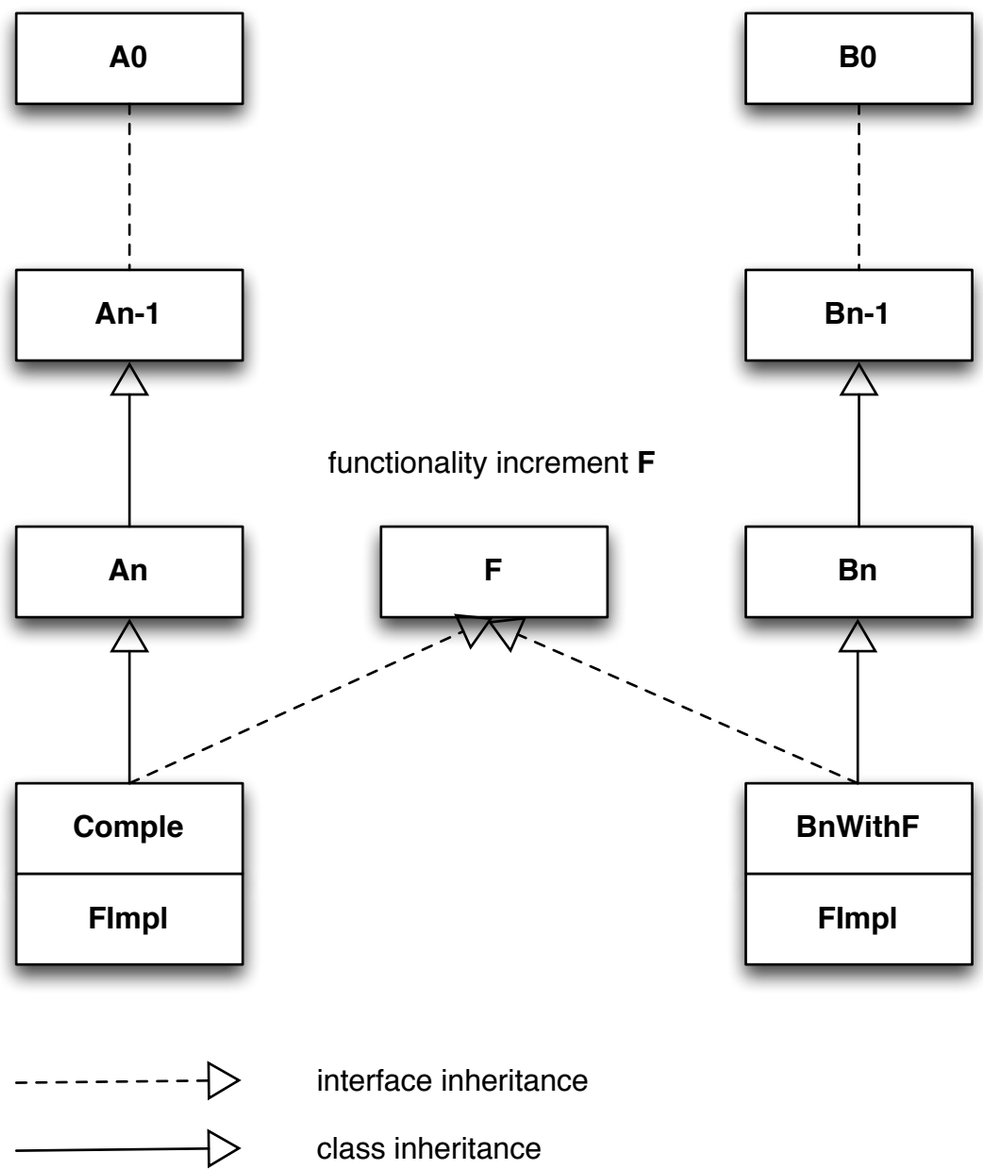
Example from An Overview of the Scala Programming Language
Tech. Report LAMP-REPORT-2006-001

Beyond a Java interface

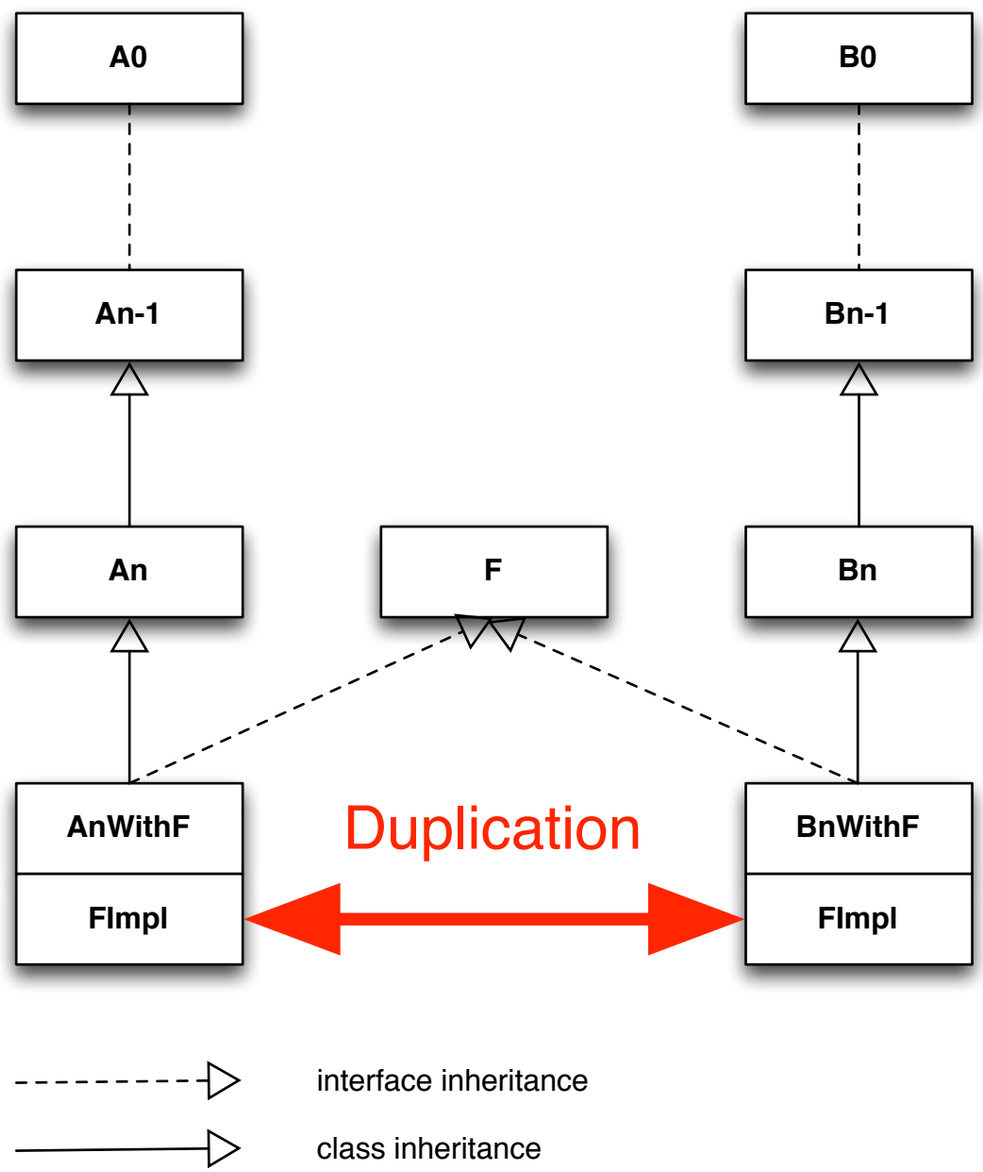
```
trait AbsIterator[T] {
  def hasNext: Boolean
  def next: T
}
trait RichIterator[T] extends AbsIterator[T] {
  def foreach(f: T => Unit): Unit =
    while (hasNext) f(next)
}
```

- Traits may include methods and fields (and maintain a state)
- But they can't be instantiated

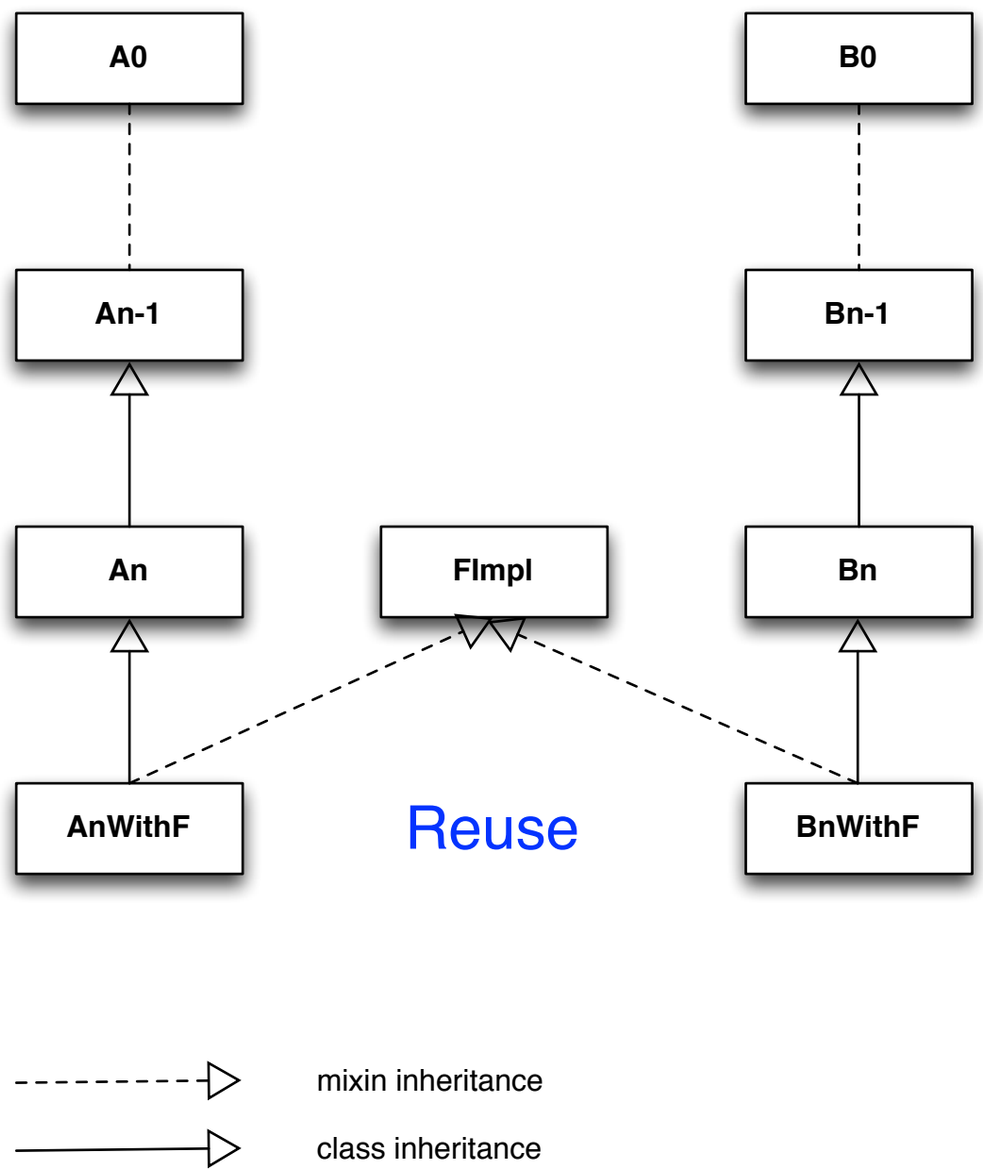
What's the idea?
Simple inheritance



What's the idea?
Simple inheritance



What's the idea?
Mixin composition



Trait compostion

```
class StringIterator(s: String) extends AbsIterator[Char] {
  private var i = 0
  def hasNext = i < s.length
  def next = { val x = s charAt i; i = i + 1; x }
}
class Iter(s: String) extends StringIterator(s) with
RichIterator[Char]
```

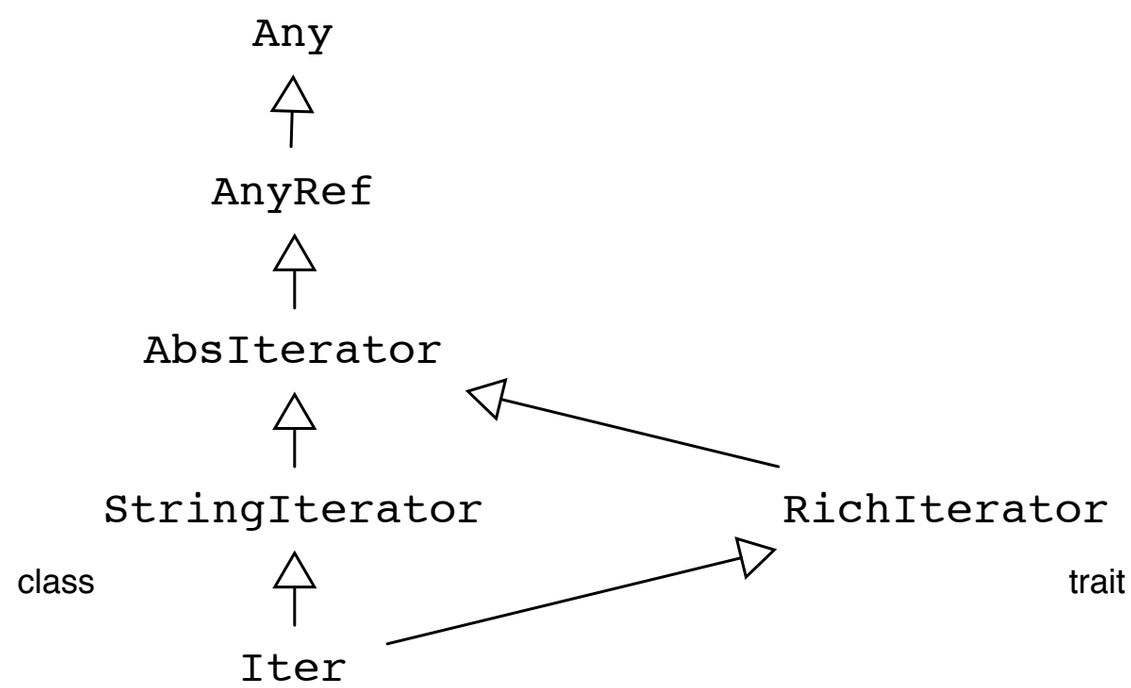
```
scala> new Iter("foo") foreach print
foo
```

`Iter` has 2 parents :

- a *superclass* `StringIterator`
- a *trait* `RichIterator`

Note: in grey, passing parameters from the constructor to the constructor of the superclass.

Inheritance hierarchy



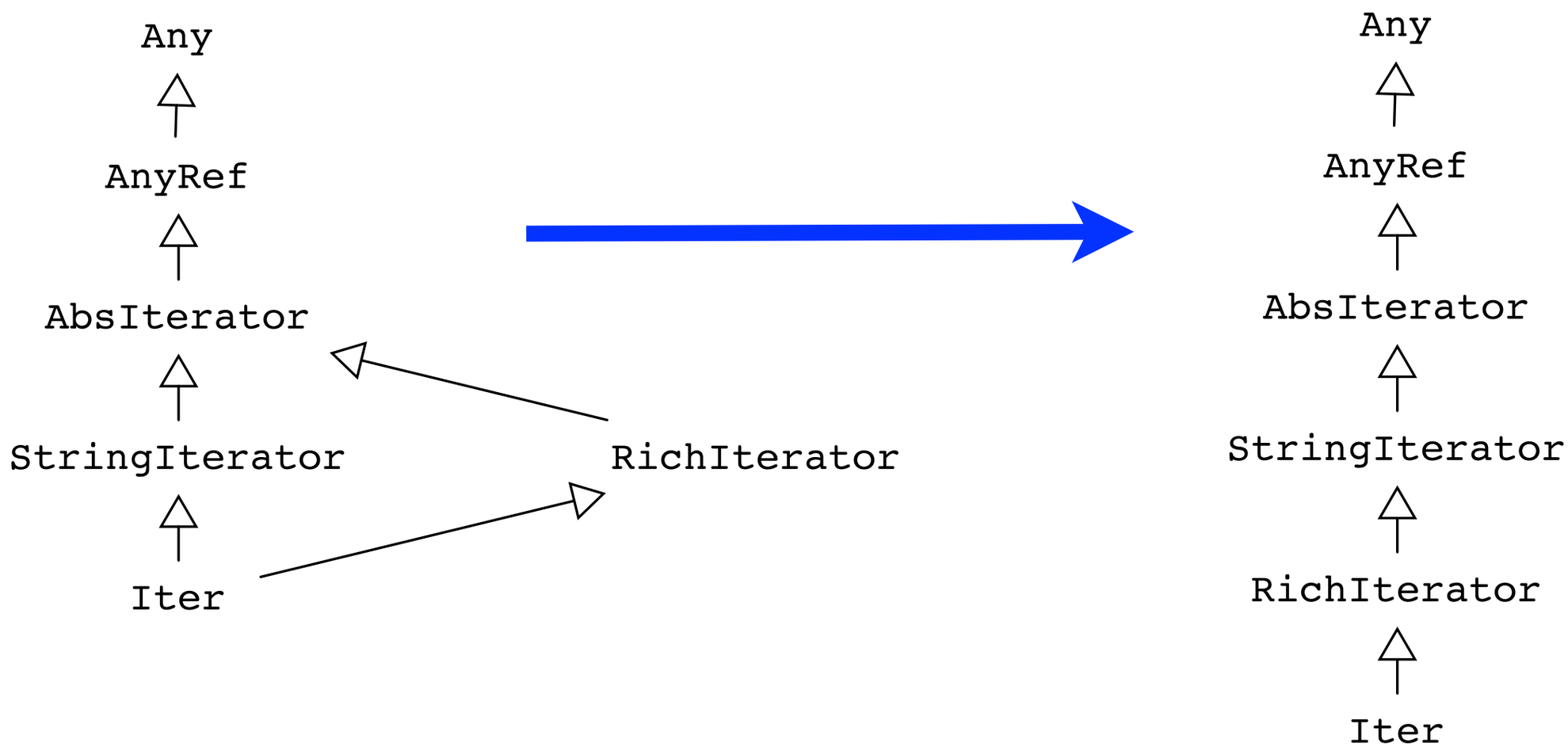
The inheritance relationship as a DAG (*Direct Acyclic Graph*) defining a partial order on types

Questions

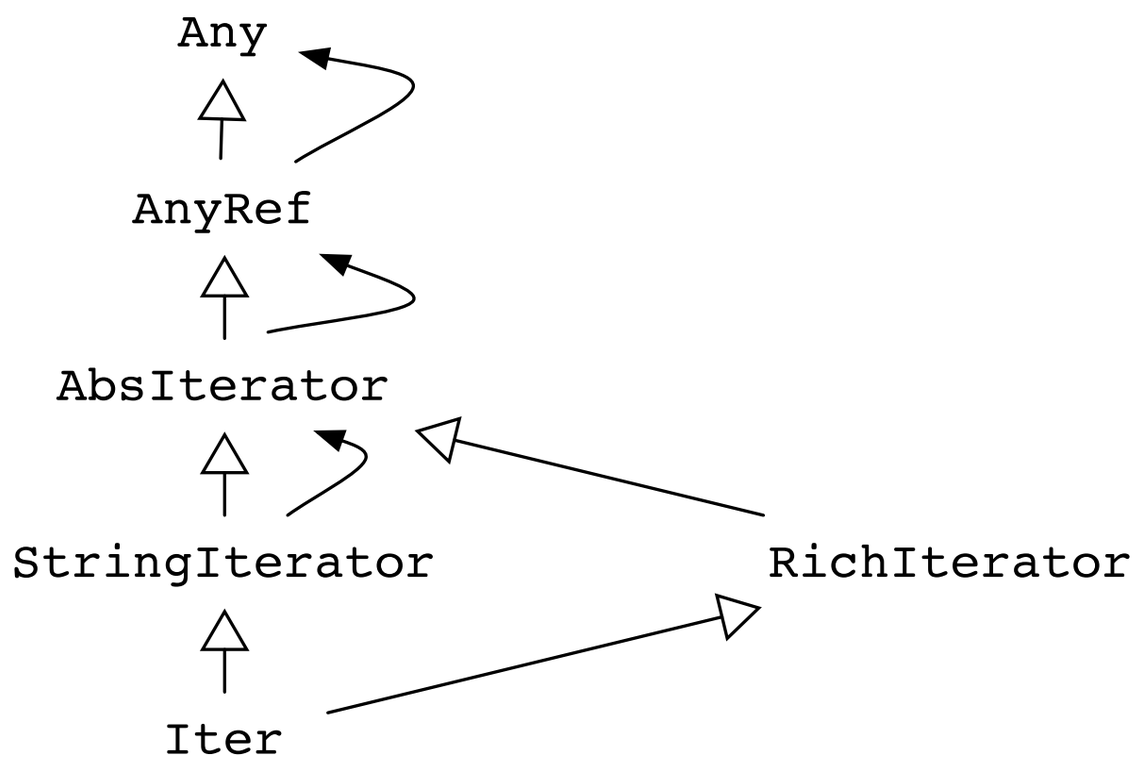
- What happens if the same parent is inherited via different paths?
- What happens if different parents define the same member?
- How to interpret supercalls?

Answer

A total order is defined through linearization

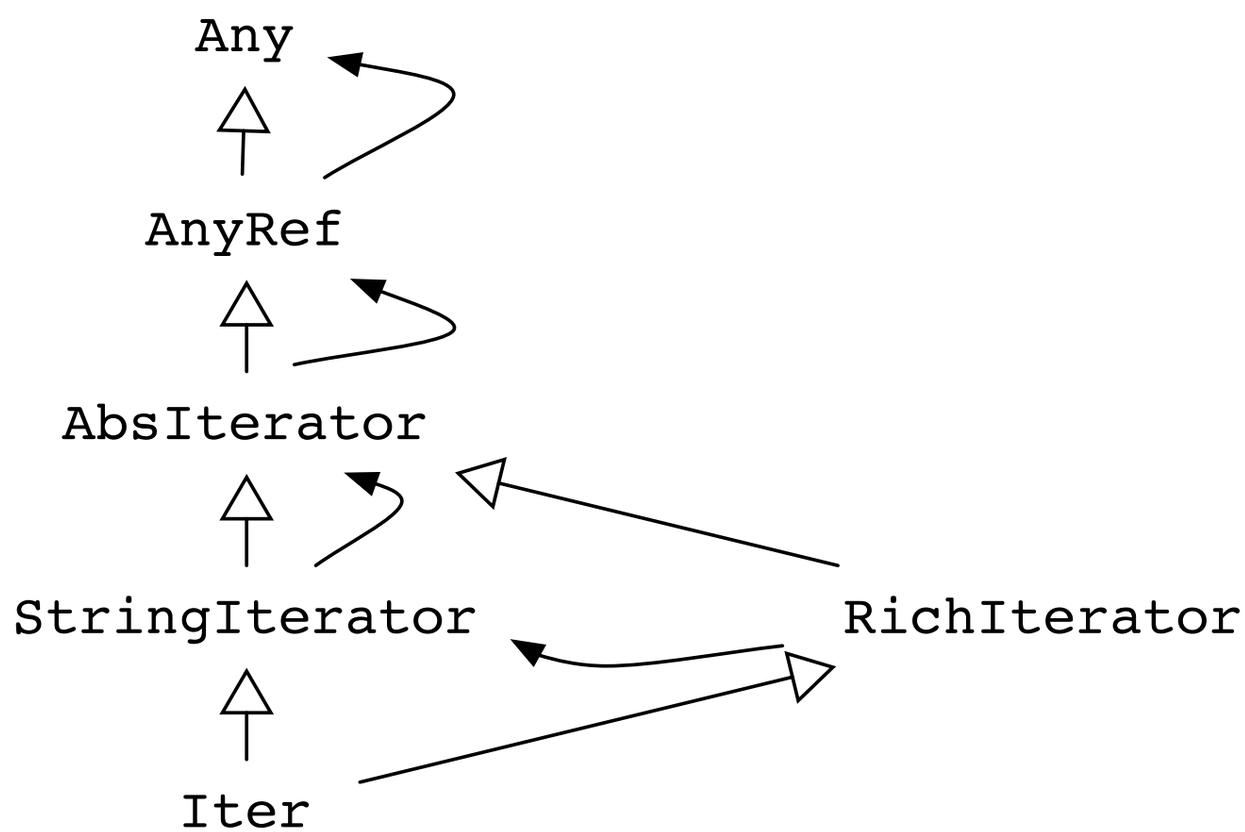


Principle of the linearization algorithm (1)



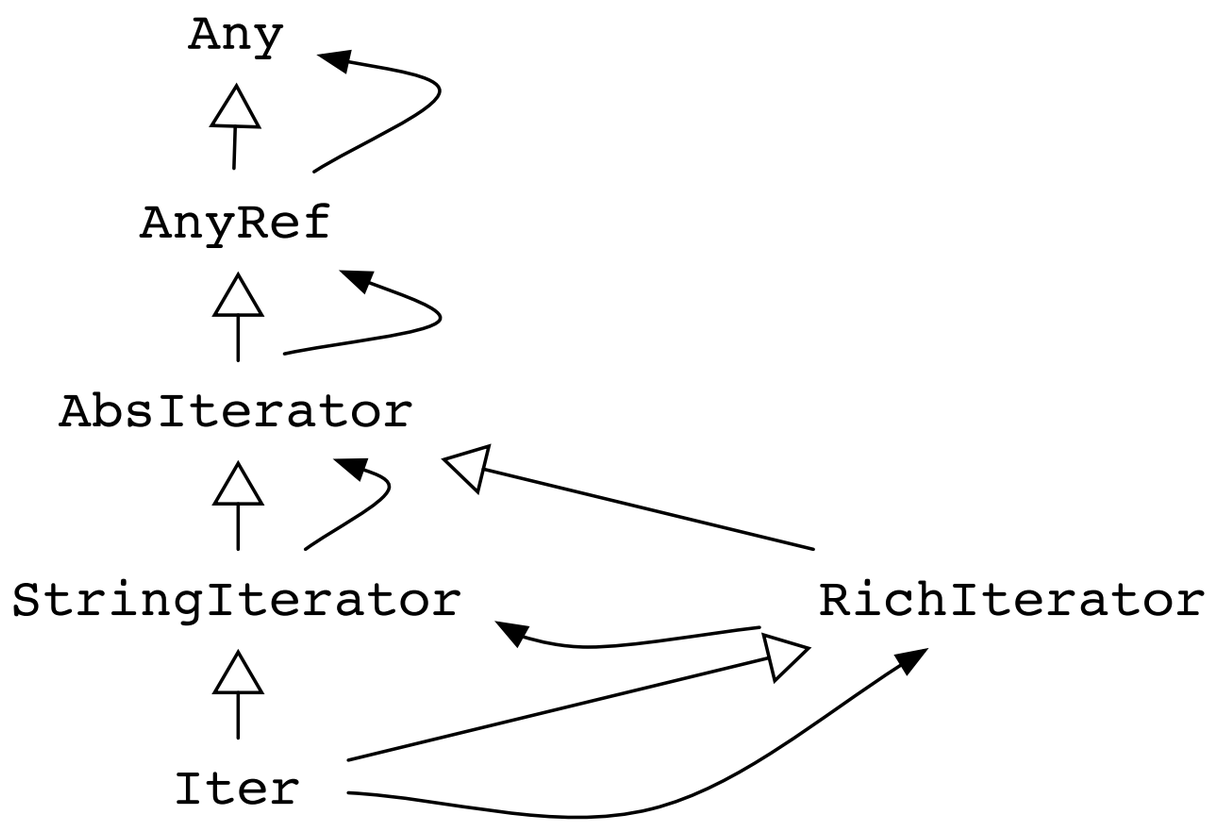
The superclass is linearized

Principle of the linearization algorithm (2)



Then the traits are linearized from left to right, ignoring the already linearized parents

Principle of the linearization algorithm (3)



Finally, the root of the DAG is added

Stacking modifications

```
trait SyncIterator[T] extends AbsIterator[T] {
  abstract override def hasNext: Boolean =
    synchronized(super.hasNext)
  abstract override def next: T =
    synchronized(super.next)
}
```

- Chain of supercalls (different from multiple inheritance)
- `super` is bound at composition time (hence the keyword `abstract`)
- when the existence of a supermethod is checked

Using traits as (genuine) interfaces

```
trait Queue[T] {
  def head: T
  def tail: Queue[T]
  def append(x: T): Queue[T]
}

object Queue {
  def apply[T](xs: T*): Queue[T] =
    new QueueImpl[T](xs.toList, Nil)
  private class QueueImpl[T](
    private val leading: List[T],
    private val trailing: List[T]
  ) extends Queue[T] {
    def head: T = mirror.leading.head
    ...
}
```

Alternative: use an abstract type.

Using traits as (genuine) interfaces

```
scala> val q = Queue(1)
q: Queue[Int] = Queue$QueueImpl@36e2c698

scala> val q1 = q append 2
q1: Queue[Int] = Queue$QueueImpl@56ccaefe


scala> import Queue.QueueImpl
import Queue.QueueImpl

scala> new QueueImpl
<console>:11: error: class QueueImpl
cannot be accessed in object Queue
       new QueueImpl
           ^
```

Rich interfaces

Example: `Traversable`

- Root of the collection hierarchy, one abstract method: `foreach[U](f: Elem => U)`
- The trait provides methods for: appending collections, map and fold operations, conversions, copying, size operations, element and subcollection retrieval, element tests, string operations, view operations