SCALA - INTRODUCTION

(WHAT IS, WHY IT'S IMPORTANT)

SCALA: INTRODUCTION

- Scala = scalable language
  - Epfl, Losanna (N. Virth)

  - Martin Odersky
    - prev: (TurboPascal,
      - Java1.5,…)

SCALA: INTRODUCTION

- Що особливого в scala (?)
  - Система типів (OO/FP)
  - Лаконічність; синтаксичний цукор (скорочення)
  - Метапрограмування
  - Широке використання в Академії та Індустрии
- Платформа
  - JVM (scala-native in progress, .net abandoned :( )
- Користувачі
  - Twitter, Paypal, Linked-In, Spotify, NY-Times …

Scala, case:

```
case class Person(firstName:String, lastName:String)
```

Java:

```
public class Person{
   String  firstName;
   String  lastName;

   Person(String firstName, String lastName) {
       this.firstName = firstName
       this.lastName = lastName
   }

   int hashCode()  {
       if (firstName==null) {
          secondName==null ? 0 else secondName.hashCode()
       } else {
         secondName==null ? firstName.hashCode()
              else firstName.hashCode() + secondName.hashCode()
       }
   }
```

```
int equals(Object other) {
   if (other==null) {
      return false
   } else {
      ..................
   }
}

String getFirstName()
                  { return firstName }

                     ..................
```

Scala, count words:

```
val lines = load(uri)
val count = lines.flatMap(_.split(" "))
                 .map(word => (word, 1))
                 .reduceByKey(_ + _)
```

// Same code, different execution

Java, count words:

```
@Override
 public void map(Object key, Text value, Context context
                ) throws IOException, InterruptedException {
   String line = (caseSensitive) ?
       value.toString() : value.toString().toLowerCase();
   for (String pattern : patternsToSkip) {
     line = line.replaceAll(pattern, "");
   }
   StringTokenizer itr = new StringTokenizer(line);
   while (itr.hasMoreTokens()) {
     word.set(itr.nextToken());
     context.write(word, one);
     Counter counter = context.getCounter(CountersEnum.class.getName(), CountersEnum.INPUT_WORDS.toString());
     counter.increment(1);
   }
 }
}

public static class IntSumReducer
     extends Reducer<Text,IntWritable,Text,IntWritable> {
 private IntWritable result = new IntWritable();

 public void reduce(Text key, Iterable<IntWritable> values,
                    Context context
                    ) throws IOException, InterruptedException {
   int sum = 0;
   for (IntWritable val : values) {
     sum += val.get();
   }
   result.set(sum);
   context.write(key, result);
 }
}
```

SCALA: INTRODUCTION

- Лаконічність - обернена сторона;

```
(n: Int) => (2 to n) |>
      (r=>r.foldLeft(r.toSet){
         (ps, x) => if (ps(x)) ps -- (x * x to n by x) else ps)
       }
```

```
trait Applicative[Z[_]] {
  def fmap[A, B](fa: Z[A], f: A => B): Z[B] = this(pure(f), fa)
  def apply[A, B](f: Z[A => B], a: Z[A]): Z[B] = liftA2(f, a, (_:A => B)(_: A))
  def liftA2[A, B, C](a: Z[A], b: Z[B], f: (A, B) => C): Z[C] =
                                                   apply(fmap(a, f.curried), b)
}
```

- К-ство ошибок <===> длинна программы.

SCALA: ОСОБЛІВОСТІ

- Pattern Matching (ATD)
- Implicit (typeclasses)
- Types
- Macros

Pattern Matching

SCALA

```
sealed trait List[+A]

class Nil extends List[Nothing]
class Cons[A](head:A, tail:List[A]) extends List[A]
```

```
alpha = nil
        ++ alpha::list alpha
```

```
def length[A](l: List[A]): Int =
  l match {
    case Nil  => 0
    case head :: tail  => 1+length(tail)
  }
```

```
dec length: list(alpha) -> int
— length nil <= 0
— length (a::l) <= length(l) + 1
```

Why Pattern Matching is better than sequence of IF-s ?

- Binding. (i.e. information from structure is extracted into variables)
- Exhaustive checking (if we miss something then we will see this)

We have not only algebraic, but object-oriented types.

- Views. (bridge, which represent object as algebraic type). Wadler, 1984

- Pattern objects. (Pattern-object method call return algebraic type)
- ODersky, 2006

We have not only algebraic, but object-oriented types.

Pattern-Object

```
x  match {
    case  A(x,y) =>  y1
    ....
}
```

extractor

```
object A {
    def unapply(x: X): Option[(A,B)]
}
```

Regular expressions:

```
final val StackElement =
              """\W+([^)]+)\(([^:]*):([^)]*)\)\W*""".r

line match {
    case StackElement(class,file,lineno) => ....
    ....
}
```

```
sealed trait Option[+A]

case class Some[A](a:A)  extends Option[A]
case object None  extends Option[Nothing]
```

ADT = { Algebraic Data Type }

HOPE (1970, Edinburg)

From today-s point of view: ADT ;)

Rod Burstall

David MacQueen

Some initial set of types: A, B, C, ….

Operations on types: Pair [A*B]
records (set of name-value-pairs)
discriminated unions (one of A or B)
// often (incorrectly): discr. union == ADT
functions: A=>B

```
data list alpha = nil
              ++ alpha :: list alpha
```

Equality by value

(A, B) == Pair[A,B]
(A+B) == emulated by sealed trait
(a:A,b:B)== case classes.

A | B ~~ partially emulated by traits
(will be implemented in dotty)

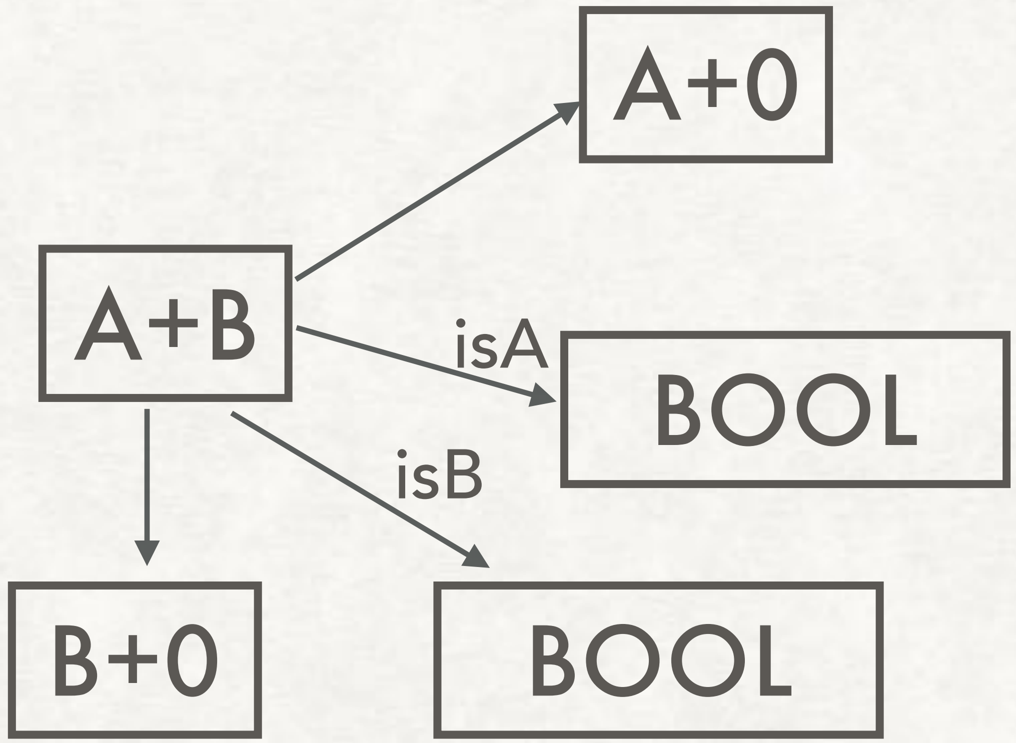
A & B ~~ partially emulated by A with B
(will be implemented in dotty)

A => B == Function[A,B]

ADT = { Algebraic Data Type }

discriminated unions (one of A or B)

// often (incorrectly): discr. union == ADT




```
data X = A ++ B
data A = 'A#integer#integer
data B = 'B#string
```

(a:A,b:B)== case classes [or objects].

```
sealed trait  X
case class A(x:Int,y:Int) extends X
case class B(s:String) extends X
```

Call by ……….

Value. // value is copied.

Reference: // reference is copied by value.
// reference in language must exists

Name // Algol 68 (today - call by closure)

Need // lazy one-time evaluation

Call by ……….

Name // Algol 68 (today - call by closure)

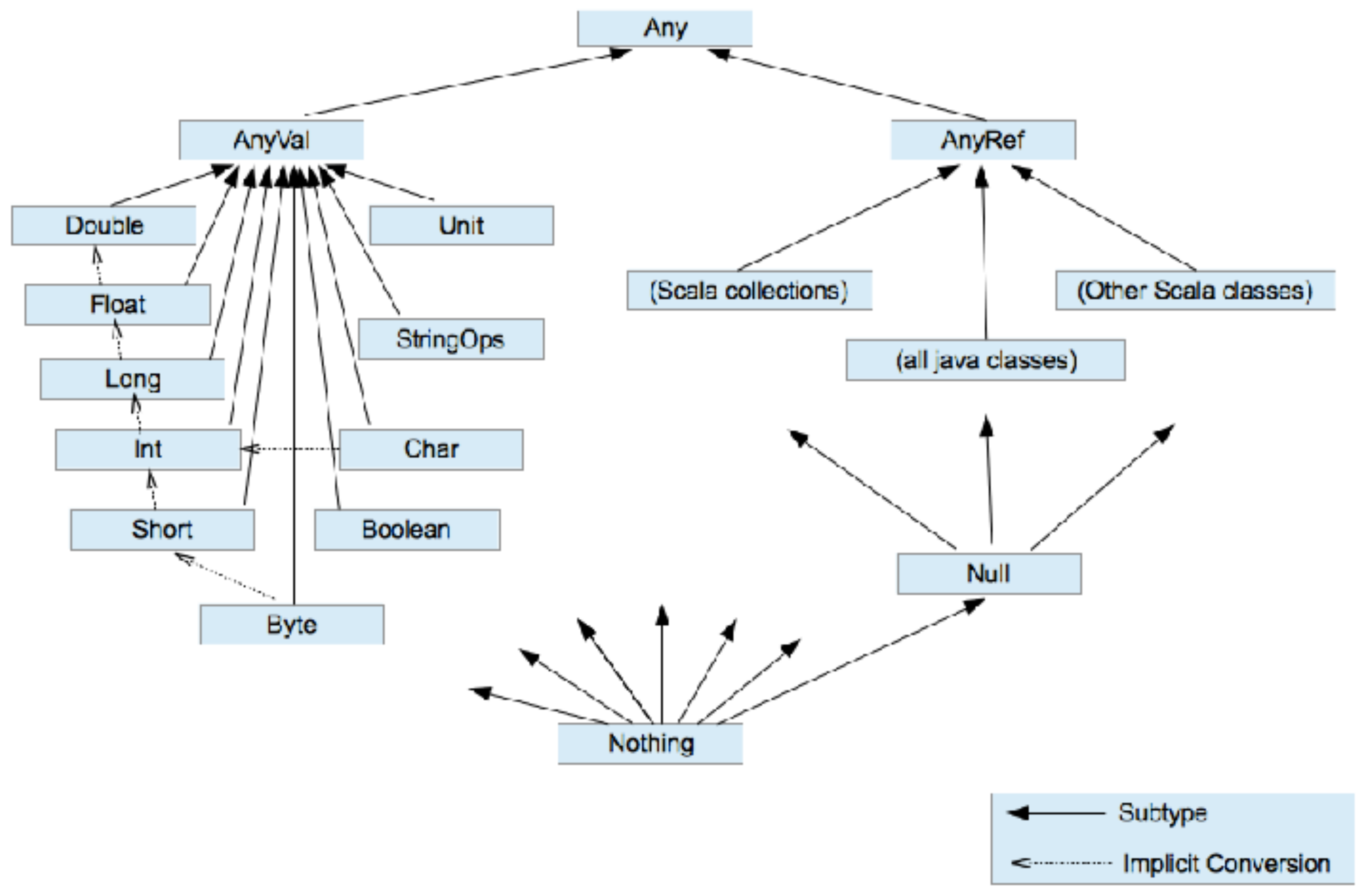
```
def  doWhile(x: => Boolean)(f : =>Unit)
{  while(x) {  f }  }
```

```
doWhile(x < 10)(x = x + 1)
```

Hieronymus Bosch
“A visual guide to the Scala language” oil on oak panels, 1490-1510

// http://classicprogrammerpaintings.tumblr.com/

Types: A <: B



Class Hierarchy

Nominative typing (type == name)

```
case class A(x: Int, y: Int)

       class B(x: Int, y: Int)
```

A != B   ~ (A <: B)   ~ (B <: A)

- Effective implementation in JVM
- Simula, Clu, C++, Java, …..

Structured typing (type == structure)

```
{ def x:Int ; def y: Int }
```

```
def f(p: { def x:Int ; def y: Int }):Int =
  p.x + p.y
```

```
val a = A(1,2)
val b = new B(1,2)

f(a) ==> 3
f(b) ==> 3
```

- implementation in JVM require reflection (can be better)
- ML, OCaml, Go
- theoretically have less corner cases than nominative

Refined type

```
B {
    def  z:  Int
}
```

- Structured type, based on nominative.
- Scala: structured types are refinement of AnyRef

Generics [Parametric Polymorphism]

```
F[T]
```

Existential types:

```
F[_]
```

```
F[X]  for  Some X
```

Bounded type parameters:

```
F[T <: Closeable]
```

//CLU where

```
F[T <: { def close(): Unit  }]
```

Type aliases

```
trait Expression[A]
{
    type Value = A
}
```

```
trait Expression
{
    type Value <: X
}
```

Undefined type alias

Scolem type

Traits:

```
trait  Interpeter {
   type Value
}
```

Flavours (Flavours, [LISP dialects]) 1980, MIT

// Howard Cannon, David Moor

Mixing (CLOS, OCaml, Groovy, Python ..)

```
trait  Show {
    this: Interpreter =>
    def  show
}
```

```
trait  Additive  {
    this:  Interpreter =>
    def plus(x:Value, y:Value): Value
}
```

```
trait BaseInterpreter[A]  extends Additive with Multiplicative with Show
{
    type Value = A
}
```

Traits:

```
trait  Additive  {
   this:  Interpreter =>
   def plus(x:Value, y:Value): Value
}
```

```
trait  LoggedAdditive extends Additive  {
   this => Logged
   def  plus(x:Value, y: Value) : Value =
    {
      log(s"(${x}+${y}")
      super.plus(x,y)
    }
}
```

// Flavours
: around
: before-next
: after-next

```
trait LoggedInterpreter[A]  extends BaseInterpreter[A]
                                        with  Logged with LoggedAdditive
```

// AOP   (aspect oriented programming)

implicit (val, def, classes)

- define rules of your world
- can be usable via implicit parameters
- implicit search <=> logical deduction
- can be dangerous.

```
implicit  def  stringToInt(s:String):Int = s.toInt

def f(x:Int):Int = x+1

f(“45”)
```

implicit (val, def, classes)

- define rules of your world
- can be usable via implicit parameters
- implicit search <=> logical deduction
- can be dangerous.

```
implicit   def  toJson(x:Int): Json = JsonNumeric(10)

    def printAsJson[A](x:A)(implicit convert:A=>Json): String =
          convert(x).prettyPrint
```

Extension methods.

//implicit-based technique (pimp my library pattern [obsolete name])

```
implicit class WithPow(x: Int) {
   def  pow(y: Int):  Int = Math.pow(x,y).toInt
}
```

```
scala> 2 pow 3
scala> res1: Int  = 8
```

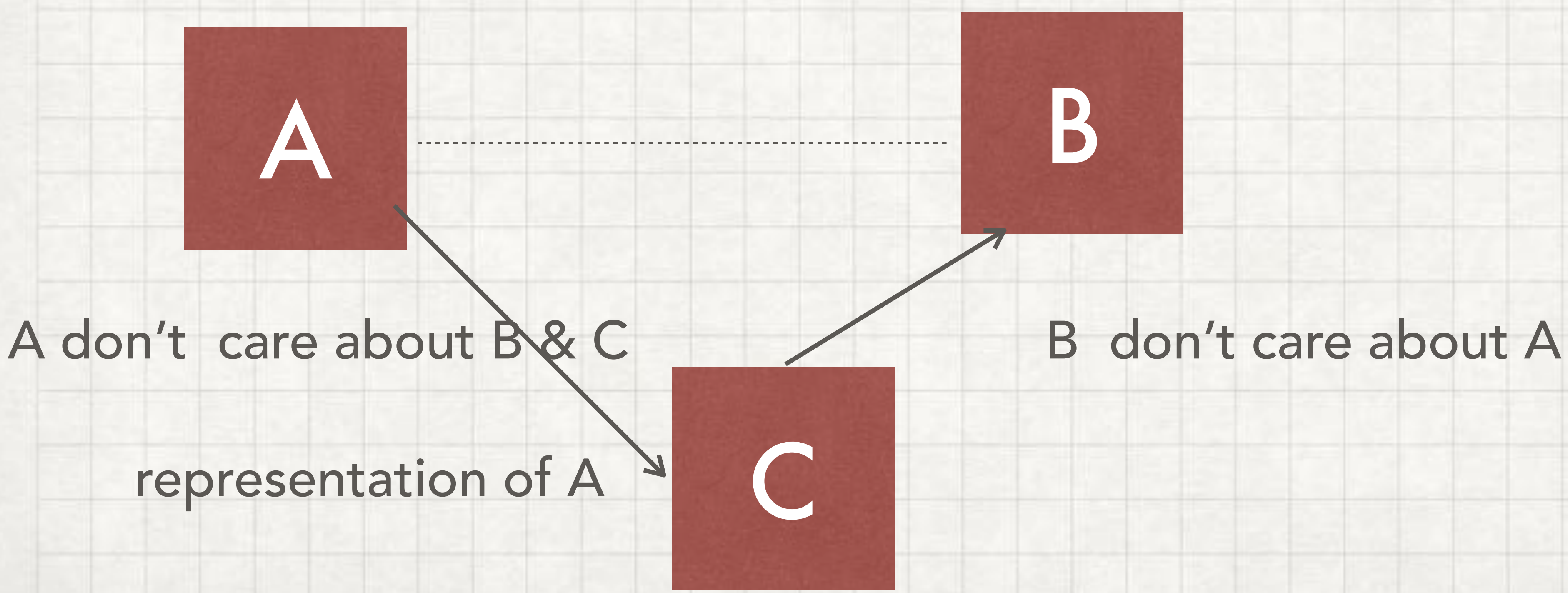
- pow — Int have no pow method
-           => compiler search for implicit with pow

Complexity

- Louse coupling   (can be build independently)
- Amount of shared infrastructure (duplication)
- Amount of location informations.

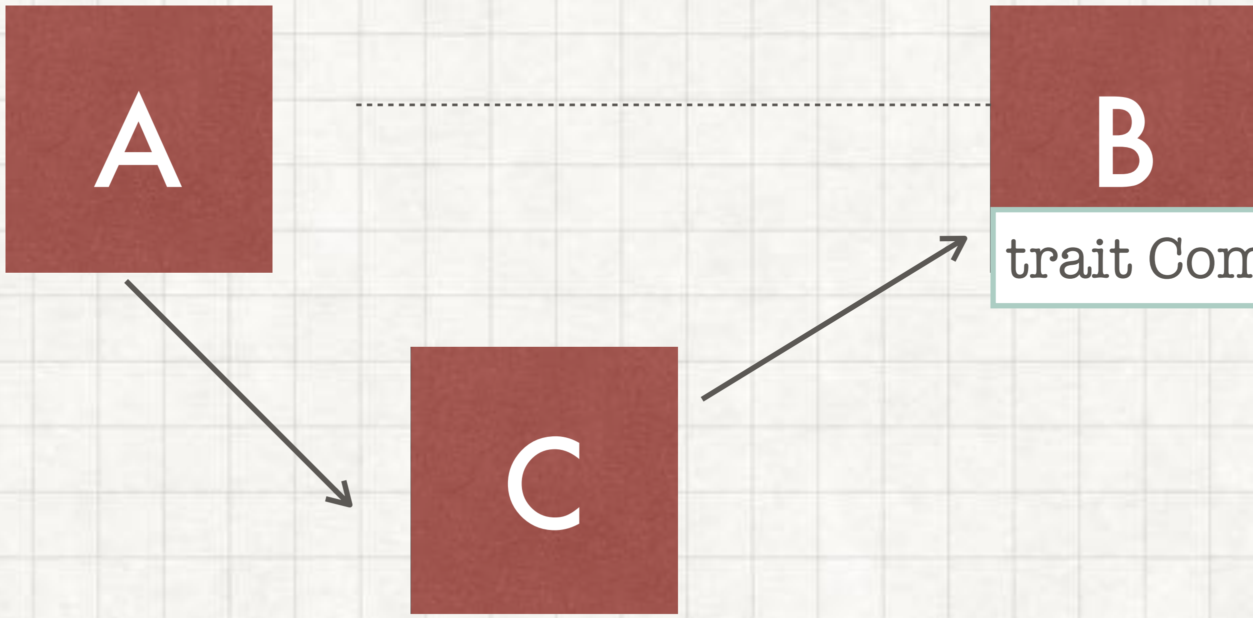
Typeclasses:

- typeclasses in Haskell
- implicit type transformations in scala
- concepts in C++14x   (WS, not ISO)
- traits in RUST



Typeclasses

```
class String
```




```
trait Comparable[A]
```

```
implicit object StringComparator extends Comparable[String]
```

RUST:

```
string
```

```
trait Ordered
{
  fn less(x:&self, y: &self) -> bool
}
```

```
imp Ordered  for  string
{

  fn less(x:&self, y: &self) -> bool
  {
   return ....
  }

}
```

Resources:

https://www.scala-exercises.org/

Book http://www.horstmann.com/scala/index.html

Courses https://www.coursera.org/learn/progfun1

Community:

UA FB:

https://www.facebook.com/groups/scala.ua/

gitter :

https://gitter.im/dev-ua/scala

World:

https://gitter.im/scala/scala