Scala, an equal marriage
Part 2

Jacques.Noye@Mines-Nantes.fr
Methods and functions
Abstract point of view

A method is a \textit{function}

class C {
    ...
    def m(al: A1, ..., an: An): B = ...
    ...
}

\textit{m} \text{ is a \textit{function}} (maths)

\[ m : C \times A_1 \times \ldots \times A_n \rightarrow B \]
\[ (\text{this}, a_1, \ldots, a_n) \rightarrow m(\text{this}, a_1, \ldots, a_n) \]
Concrete point of view

A method is not a (first-class) function

scala> def inc(x: Int) = x + 1
add: (Int)Int

scala> val z = inc(2)
z: Int = 3

scala> inc
<console>:7: error: missing arguments for method inc in object $iw;
follow this method with `_` if you want to treat it as a partially applied function
    inc
    ^

A method is only available as a name to be used in a method call. It is not available as a value.

Note: toplevel definitions are actually made members of an internal object. Try scala -Xprint:parser (scary).
First-class functions in Scala

scala> val inc = (x: Int) => x + 1
add: (Int) => Int = <function1>

scala> val z = inc(2)
z: Int = 3

scala> inc
res1: (Int) => Int = <function1>

scala> println(inc)
<function1>
Implementation Principle (in Java)

```java
public interface Function1<T1, T> {
    T apply(T1 e1);
}

public class Inc implements Function1<Integer, Integer> {
    public Integer apply(Integer e1) {
        return e1 + 1;
    }
}

public static void main(String[] args) {
    Function1<Integer, Integer> inc = new Inc();
    Integer z = inc.apply(2);
    System.out.println(z);
    System.out.println(inc);
}
```
Implementation Principle (in Java)

```java
public interface Function1<T1, T> {
    T apply(T1 e1);
}

public class Inc implements Function1<Integer, Integer> {
    public Integer apply(Integer e1) {
        return e1 + 1;
    }
}

public static void main(String[] args){
    Function1<Integer, Integer> inc = new Inc();
    Integer z = inc.apply(2);
    System.out.println(z);
    System.out.println(inc);
}
```

Note: this is not as easy in the general case (closure).
Using an anonymous Java class

```java
public interface Function1<T1, T> {
    T apply(T1 e1);
}

public class Test1 {
    public static void main(String[] args) {
        Function1<Integer, Integer> inc =
                new Function1<Integer, Integer>() {
            public Integer apply(Integer e1) {
                return e1 + 1;
            }
        };
        Integer z = inc.apply(2);
        System.out.println(z);
        System.out.println(inc);
    }
}
```

instantiation of the anonymous class, which implements `Function1`
Implementation Principle (in Scala)

scala> val inc = new Function1[Int, Int]{
    |     def apply(e1: Int) = e1 + 1
    |}
inc: Integer => Integer = <function1>

scala> inc.apply(2)
res4: Integer = 3

scala> inc(2)
res5: Integer = 3

scala> inc{2}
res6: Int = 2

Note: the anonymous class is instantiated, not the trait.
The trait \texttt{Function}_n

This is the usual rule: function types are covariant (+) in their result type and contravariant (−) in their argument type.
Variance

if $S <: T$ then

- **Covariance:** $U[\ldots, S, \ldots] <: U[\ldots, T, \ldots]$

- **Contravariance:** $U[\ldots, S, \ldots] :> U[\ldots, T, \ldots]$

- **Nonvariance:** $U[\ldots, S, \ldots]$ and $U[\ldots, T, \ldots]$ cannot be compared
Turning a method into a function

scala> def inc(x: Int) = x + 1
add: (Int)Int

scala> val incF = (x: Int) => inc(x)
incF: Int => Int = <function1>

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

scala> val addF = (x: Int, y: Int) => add(x, y)
addF: (Int, Int) => Int = <function2>

Note: compare the answers for def and val.
Using partial application

Explicit

scala> val incF = inc _
incF: Int => Int = <function1>

scala> val addF = add _
addF: (Int, Int) => Int = <function2>

Implicit

scala> val incF: Int => Int = inc
incF: Int => Int = <function1>

scala> val addF: addF: (Int, Int) = add
addF: (Int, Int) => Int = <function2>
Using partial application

scala> val incF = (_: Int) + 1
incF: Int => Int = <function1>

scala> val addF = (_: Int) + (_: Int)
addF: (Int, Int) => Int = <function2>

scala> val incF: Int => Int = _ + 1
incF: Int => Int = <function1>

scala> val addF: addF: (Int, Int) = _ + _
addF: (Int, Int) => Int = <function2>
Curried methods

scala> val addF = (x: Int, y: Int) => x + 1
   
addF: (Int, Int) => Int = <function2>

scala> val inc = addF(_:Int, 1)
   
inc: Int => Int = <function1>

scala> val inc = addF(1, _:Int)
   
inc: Int => Int = <function1>

scala> def addC(x: Int) (y: Int) = x + y // curried method
   
addC: (x: Int)(y: Int)Int

scala> val inc = addC(1)_
   
inc: Int => Int = <function1>

scala> val inc: Int => Int = addC(1)
   
inc: Int => Int = <function1>
Closures

scala> val more = 1
more: Int = 1

scala> val addMore = (x: Int) => x + more
addMore: (Int) => Int = <function1>

scala> addMore(10)
res26: Int = 11

scala> val more = 2
more: Int = 2

scala> addMore(10)
res27: Int = 11

The value of more is fixed here
Terminology

• Let us consider the expression $e$

  $(x: \text{Int}) \Rightarrow x + \text{more}$

• $x$ is \textit{bound} in $e$ (it appears as a parameter)

• $\text{more}$ is \textit{free} in $e$ (it is defined elsewhere)

• The value of $e$ is a \textit{closure} ($e$ is closed over its free variables)
Closures
with mutable free variables

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

scala> val addMore = (x: Int) => x + more
addMore: (Int) => Int = <function1>

scala> addMore(10)
res26: Int = 11

scala> more = 2
more: Int = 2

scala> addMore(10)
res27: Int = 12
```

The “value” of more is fixed here

The “value” of more is fixed here

Parameter passing

- By value, by default
- By name with by-name parameters: compare:

```scala
def assert(test: Boolean) =
  if (assertionEnable && !test)
    throw new assertion

def assert(test: () => Boolean) =
  if (assertionEnable && !test())
    throw new assertion

def assert(test: => Boolean) =
  if (assertionEnable && !test)
    throw new assertion
```

*test is a by-name parameter*
Lists
Typing, pattern matching and higher-order programming
Inductive definition

- A list is
  - either an empty list \texttt{Nil}
  - or a list \( x :: \! \! 1 \) consisting of an element \( x \), \textit{head} of the list, and of a list \( 1 \), \textit{tail} of the list (\( :: \) is the \textit{cons} operator).
Building lists

scala> Nil
res0: scala.collection.immutable.Nil.type = List()

scala> 1 :: Nil
res1: List[Int] = List(1)

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

scala> val tail = 2 :: Nil
tail: List[Int] = List(2)

scala> val l = 1 :: tail
l: List[Int] = List(1, 2)
Building lists with the constructor List

scala> val l = List(1, 2, 3)
l: List[Int] = List(1, 2, 3)

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

scala> val l = 1 :: List(1) :: Nil
l: List[Any] = List(1, List(1))

This uses the “apply trick”:

object List {
  def apply[A](xs: A*): List[A] = xs.toList
}

A variable nb of args.
xs is of type Seq[A]
Typing

- Lists are **homogeneous**: all the elements are considered to have the same type $T$. The list has type $\text{List}[T]$.

- Lists are **covariants**:

  ```java
  abstract class List[+T] {
      ...
  }
  ```
Covariance and subtyping

- If $S <: T$ then $\text{List}[S] <: \text{List}[T]$
- If an apple is a fruit then a basket of apples is a basket of fruits.
- If $S_1 <: T$ and $S_2 <: T$ then a list of $S_1$ and $S_2$ is a list of $T$
- Mixing cabbages and carrots in a basket results in a basket of vegetables.
The empty list

- Nothing is the smallest type: for all T, Nothing <: T
- List[Nothing] <: List[T]
- T = Int: the empty list is a list of integers

```scala
scala> val l = List()
l: List[Nothing] = List()

scala> val l: List[Int] = empty
l: List[Int] = List()
```
Accessors

scala> val numbers = List(1, 2, 3)
numbers: List[Int] = List(1, 2, 3)

scala> numbers(1)
res5: Int = 2

scala> numbers.head
res6: Int = 1

scala> numbers.tail
res7: List[Int] = List(2, 3)

scala> numbers.isEmpty
res8: Boolean = false

scala> Nil.head
java.util.NoSuchElementException: head of empty list
Computing with lists
basic idiom: follow the inductive definition

scala> def length[T](l: List[T]): Int =
   if (l.isEmpty) 0
   else 1 + length(l.tail)
length: [T](l: List[T])Int

scala> length[Int](List(1, 2, 3))
res0: Int = 3
Immutability

scala> var e = 1
e: Int = 1

scala> val l = List(e)
l: List[Int] = List(1)

scala> e = 2
e: Int = 2

scala> l
res0: List[Int] = List(1)

Note: compare with closures.
Modifying a list

Copying the list with the modifications!

scala> def append[T](l1: List[T], l2: List[T]): List[T] = 
   if (l1.isEmpty) l2 
   else l1.head :: append(l1.tail, l2)
append: [T](l1: List[T],l2: List[T])List[T]

scala> append(List(1, 2), List(3, 4))
res0: List[Int] = List(1, 2, 3, 4)

scala> append(List(1, 2), List("3", "4"))
res1: List[Any] = List(1, 2, "3", "4")

A thrill
Patterns

scala> val h :: t = List(1, 2)
h: Int = 1
t: List[Int] = List(2)

scala> val e1 :: e2 :: Nil = List(1, 2)
e1: Int = 1
e2: Int = 2

scala> val List(_, x) = List(1, 2)
x: Int = 2

scala> val List(1, x) = List(2, 1)
scala.MatchError: List(2, 1)
The construct `match`

```scala
def length[T](l: List[T]): Int = 
  l match {
    case Nil => 0
    case _ :: t => 1 + length(l)
  }

def append[T](l1: List[T], l2: List[T]): List[T] = 
  l1 match {
    case Nil => l2
    case h1 :: t1 => h1 :: append(t1, l2)
  }
```

Note: the compiler detects non exhaustive patterns.
A case sequence is a (partial) function literal

```
scala> val isEmpty: List[Int] => Boolean = 
   {case Nil => true; case _ :: _ => false}
isEmpty: List[Int] => Boolean = <function1>
```

**Partial functions** (instances of `PartialFunction[-A, +B]`) extend functions of one argument with a method:

```
def isDefinedAt(x: A): Boolean
```
Pattern Matching

The expression \[ \{ \text{case } p_1 \Rightarrow e_1; \ldots; \text{case } p_n \Rightarrow e_n \} \]
is a partial function:

- isDefinedAt returns true if one of the patterns matches, false otherwise
- apply returns the value of \( e \) for the first pattern \( p \) which matches, throws MatchError otherwise
Other basic patterns

• Typed Patterns: \texttt{varId: Type} (checks type of \texttt{varId}, as \texttt{varId.isDirectory[Type]})

• Pattern Binders: \texttt{varId@Pattern} (binds matched value to \texttt{varId})
The class `List` extends `SeqFactory` {
  def apply[A](xs: A*): List[A] = xs.toList
}
abstract class List[+T] {
  def isEmpty: Boolean
  def head: T
  def tail: List[T]

  def :::[U >: T](x: U): List[U] = new :::(x, this)
}

case object Nil extends List[Nothing] {
  ...
}

case class :::[T](h: T, tl: List[T]) extends List[T] {
  ...
}

There are also upper bounds: U <: Upper >: Lower
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 (:: and Nil for lists)
Typical AST example

abstract class Expr

case class BinExpr(op: String, e1: Expr, e2: Expr) extends Expr

case class UnExpr(op: String, e: Expr) extends Expr

case class Number(n: Int) extends Expr

object Eval {
    def eval(e: Expr): Int = e match {
        case Number(n) => n
        case BinExpr("+", e1, e2) => eval(e1) + eval(e2)
        case BinExpr("-", e1, e2) => eval(e1) - eval(e2)
        case UnExpr("-", e) => - eval(e)
    }

    def main(args: Array[String]) =
        println(eval(BinExpr("+", Number(1), UnExpr("-", Number(1))))))
}
The lower bound is not an option

scala> :paste
// Entering paste mode (ctrl-D to finish)

abstract class MyList[+T] {
  def ::(x: T): MyList[T] = new ::(x, this)
}
case object Nil extends MyList[Nothing]
case class ::[T](h: T, tl: MyList[T]) extends MyList[T]

// Exiting paste mode, now interpreting.

<console>:12: error: covariant type T occurs in contravariant position in type T of value x
def ::(x: T): MyList[T] = new ::(x, this)
  ^
detected by the compiler
Extractors

Un extractor is an object which provides a method apply (optional) and unapply (mandatory) to construct and destruct a pattern, respectively.

```scala
object Pair {
  def apply[A, B](x: A, y:B) = Tuple2(x, y)
  def unapply[A, B](x: Tuple2[A, B]): Option[Tuple2[A, B]] = Some(x)
}
```
Capturing recursion patterns

map

def mapInc(xs: List[Int]): List[Int] = xs match {
  case Nil => Nil
  case x :: xs => x + 1 :: mapInc(xs)
}
def map2String(xs: List[Int]): List[String] = xs match {
  case Nil => Nil
  case x :: xs => x.toString :: map2String(xs)
}
def mapInt[T](xs: List[Int], f: Int => T): List[T] = xs match {
  case Nil => Nil
  case x :: xs => f(x) :: mapInt(xs, f)
}

1st generalization

def map[S, T](xs: List[S], f: S => T): List[T] = xs match {
  case Nil => Nil
  case x :: xs => f(x) :: map(xs, f)
}
Example of use

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

scala> map(List(1, 2, 3), (_: Int) + 1)
res18: List[Int] = List(2, 3, 4)

scala> def map2String[S](xs: List[S]) =
  map[S, String](xs, (x: S) => x.toString)
map2String: [S](xs: List[S])List[String]

scala> map2String[Char](List('a', 'b'))
res19: List[String] = List(a, b)
Capturing recursion patterns

\textbf{foldRight}

\begin{verbatim}
def sum(xs: List[Int]): Int = xs match {
  case Nil => 0
  case x :: xs => x + sum(xs)
}
\end{verbatim}

\begin{align*}
def \text{foldRight}[S,T](xs: List[S], s: T, f: (S, T) \rightarrow T): T &= xs \text{ match } \{ \\
  \text{case Nil } &\Rightarrow s \\
  \text{case x :: xs } &\Rightarrow f(x, \text{foldRight}(xs, s, f))
\}\end{align*}
Capturing recursion patterns

foldLeft

def sum(soFar: Int, xs: List[Int]): Int = xs match {
  case Nil => soFar
  case x :: xs => sum(soFar+x, xs)
}

def foldLeft[S,T](s: T, xs: List[S], f: (T, S) => T): T = xs match {
  case Nil => s
  case x :: xs => foldLeft(f(s, x), xs, f)
}
for expressions
def myMap[A, B](f: A => B, xs: List[A]): List[B] = xs match {
  case Nil => Nil
  case x::xs => f(x) :: myMap(f, xs)
}
def myFilter[A](p: A => Boolean, xs: List[A]): List[A] = xs match {
  case Nil => Nil
  case x::xs => if (p(x)) x :: myFilter(p, xs) else myFilter(p, xs)
}
def myFlatMap[A, B](f: A => List[B], xs: List[A]): List[B] = xs match {
  case Nil => Nil
  case x::xs =>  f(x) ::: myFlatMap(f, xs)
}
def myForeach[A](f: A => Unit, xs: List[A]): Unit = xs match {
  case Nil => ()
  case x :: xs => f(x); myForeach(f, xs)
}
map & Co
as methods of List

scala> val words = List("The", "Dark", "Knight", "Rises")
words: List[java.lang.String] = List(The, Dark, Knight, Rises)

scala> words map (_.length)
res9: List[Int] = List(3, 4, 6, 5)

scala> words map (_.toList)
res13: List[List[Char]] = List(List(T, h, e), List(D, a, r, k), List(K, n, i, g, h, t), List(R, i, s, e, s))

scala> words filter (_.length > 3)
res10: List[java.lang.String] = List(Dark, Knight, Rises)

scala> words flatMap (_.toList)
res11: List[Char] = List(T, h, e, D, a, r, k, K, n, i, g, h, t, R, i, s, e, s)

scala> words foreach (print _)
TheDarkKnightRises
for expressions
loops and comprehensions

-- comprehension in Haskell, ZF
mySort :: Ord a => [a] -> [a]
mySort [] = []
mySort (x:xs) = mySort [e | e <- xs, e<x] ++
                [x] ++
                mySort [e | e <- xs, e>=x]

for (e <- xs if e < x) yield e
map & Co.  
using for expressions

// for comprehensions (yield)
def myMap[A, B](f: A => B, xs: List[A]): List[B] =
    for (x <- xs) yield f(x)

def myFilter[A](p: A => Boolean, xs: List[A]): List[A] =
    for (x <- xs if p(x)) yield x

def myFlatMap[A, B](f: A => List[B], xs: List[A]): List[B] =
    for (x <- xs; y <- f(x)) yield y

// for loop (no yield)
def myForeach[A](f: A => Unit, xs: List[A]): Unit =
    for (x <- xs) f(x)
Translating \texttt{for} loops

A single generator

(1) \texttt{for (} x \leftarrow \texttt{expr} \texttt{) body} \rightarrow \texttt{expr \ foreach (} x \rightarrow \texttt{body})

Several generators: one loop per generator

(2) \texttt{for (} x \leftarrow \texttt{expr1; y} \leftarrow \texttt{expr2 \ tail) body} \\
\rightarrow \texttt{expr1 \ foreach (} \texttt{for (} y \leftarrow \texttt{expr2 \ tail) body})
Translating for comprehensions

A single generator

(3) for (x <- expr1) yield expr2 \rightarrow expr1 map (x => expr2)

A sequence of generators

(4) for (x <- expr1; y <- expr2 tail) yield expr3
\rightarrow expr1 flatMap (x => for (y <- expr2 tail) yield expr3)
Eliminating filters and definitions

Filter

(5) \( x \leftarrow \text{expr1 if expr2} \)
\( \rightarrow x \leftarrow \text{expr1 filter } (x \Rightarrow \text{expr2}) \)

Definition

(6) \( x \leftarrow \text{expr1}; y = \text{expr2} \)
\( \rightarrow (x, y) \leftarrow \text{for } (x \leftarrow \text{expr1}) \text{ yield } (x, \text{expr2}) \)

Note: \( x, \text{expr1} \ldots \) are metavariables.
Eliminating patterns

Refutable pattern in a generator

\[(0)\] \(p \leftarrow expr1\)
\(\rightarrow expr1\) \text{filter} \{ \text{case } p \Rightarrow \text{true}; \text{case } _ \Rightarrow \text{false} \}

Irrefutable pattern in a generator requires variants of rules (1) to (6)

\[(3')\] \text{for } (p \leftarrow expr1) \text{ yield expr2}
\(\rightarrow expr1\) \text{map} \{ \text{case } p \Rightarrow expr2 \} \}

Note: full scheme is given in Scala Language Specifications.
Generalization

Translation applied by the compiler.

Applies to any class implementing map, flatMap, filter and foreach (possibly partially).

For instance:

```scala
scala> val a = Array(1, 2, 3)
a: Array[Int] = Array(1, 2, 3)
scala> for (i <- 0 to 2) print(a(i))
123
scala> 0 to 2
res1: scala.collection.immutable.Range.Inclusive = Range(0, 1, 2)
scala> for (x <- a) print(x)
123
scala> for (i <- 0 to 2) yield a(i)
res2: scala.collection.immutable.IndexedSeq[Int] = Vector(1, 2, 3)
scala> for (x <- a) yield x
res3: Array[Int] = Array(1, 2, 3)
```

the type of the result depends on the generator
Some more fun/headaches
A case for abstract types

scala> :paste
// Entering paste mode (ctrl-D to finish)
class Food
class Grass extends Food
abstract class Animal {
def eat(food: Food)
}
class Cow extends Animal {
  override def eat(food: Grass){}
}
// Exiting paste mode, now interpreting.

<console>:15: error: class Cow needs to be abstract, since method eat
  in class Animal of type (food: Food)Unit is not defined
  (Note that Food does not match Grass: class Grass is a subclass of
  class Food, but method parameter types must match exactly.)
    class Cow extends Animal {
      ^
Note: the super classes of class Cow contain the following, non final
members named eat:
def eat(food: Food): Unit
  override def eat(food: Grass){}
Abstract types to the rescue

class Food
class Grass extends Food
class Fish extends Food

abstract class Animal {
    type SuitableFood <: Food // abstract type with upper bound
def eat(food: SuitableFood)
}
class Cow extends Animal {
    type SuitableFood = Grass // concrete type
    override def eat(food: Grass) {}
}
Abstract types to the rescue

scala> val bessy = new Cow
bessy: Cow = Cow@6d643e7b

scala> bessy eat (new Grass)

scala> bessy eat (new Fish)
<console>:14: error: type mismatch;
  found    : Fish
  required: Grass
       bessy eat (new Fish)
     ^
Path-dependent types

scala> val bessy: Animal = new Cow
bessy: Animal = Cow@647a303a

scala> bessy eat (new Grass)
<console>:13: error: type mismatch;
  found   : Grass
  required: bessy.SuitableFood
    bessy eat (new Grass)

scala> val bessy = new Cow
bessy: Cow = Cow@575db0f5

scala> val lili = new Cow
lili: Cow = Cow@1416c1f4

scala> lili eat (new bessy.SuitableFood)
Path-dependent types and inner classes

scala> class Outer {
  class Inner
}
}
defined class Outer
scala> val o1 = new Outer
o1: Outer = Outer@2029a303
scala> val i1 = new o1.Inner
i1: o1.Inner = Outer$Inner@200069ed
scala> val o2 = new Outer
o2: Outer = Outer@badfba
scala> val i2 = new o2.Inner
i2: o2.Inner = Outer$Inner@6ec4786e
scala> val l = List(i1, i2)
l: List[Outer#Inner] = List(Outer$Inner@200069ed, Outer$Inner@6ec4786e)
(Simple) Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class or trait</td>
<td>class C ... , trait C ...</td>
</tr>
<tr>
<td>Tuple type</td>
<td>(T₁, ..., Tₙ)</td>
</tr>
<tr>
<td>Function type</td>
<td>(T₁, ..., Tₙ) ⇒ T</td>
</tr>
<tr>
<td>Annotated type</td>
<td>T @A</td>
</tr>
<tr>
<td>Parameterized type</td>
<td>A[T₁, ..., Tₙ]</td>
</tr>
<tr>
<td>Singleton type</td>
<td>value.type</td>
</tr>
<tr>
<td>Type projection</td>
<td>O#I</td>
</tr>
<tr>
<td>Compound type</td>
<td>T₁ with T₂ with ... with Tₙ { declarations }</td>
</tr>
<tr>
<td>Infix type</td>
<td>T₁ A T₂</td>
</tr>
<tr>
<td>Existential type</td>
<td>T forSome { type and val declarations }</td>
</tr>
</tbody>
</table>

Simple: not qualified.

From *Scala for the Impatient*
Compound types
structural subtyping

case class Bird(val name: String) {
  def fly(height: Int) = {}
}
case class Plane(val callsign: String) {
  def fly(height: Int) = {}
}
def takeoff(
  runway: Int,
  r: {val callsign: String; def fly(height: Int})
) = {
  r.fly(1000)
}
val bird = new Bird("Polly the parrot"){ val callsign = name }
val a380 = new Plane("TZ-987")
takeoff(42, bird)
takeoff(89, a380)

Note: uses reflection at runtime.
Existential types

\[ T \text{ forSome } \{ \text{decls} \} \]

- Mainly used for interoperability with Java
- **Java:**
  - `Iterator<?>`
  - `Iterator<?> \text{extends} \text{Component}>`
- **Scala:**
  - `Iterator<?> \text{forSome} \{ \text{type} T \} \text{or} \text{Iterator[_]}`
  - `Iterator[T] \text{forSome} \{ \text{type} T \text{ <: Component} \}`
Actors
Concurrency paradigms (asynchronous)

• Shared memory
  • Pessimistic protocols: locking, monitors (Java)
  • Optimistic protocols: transactional memory

• Shared nothing
  • Synchronous message passing (Ada’s rendez-vous)
  • Asynchronous message passing (Actor’s mailbox)
Bounded Buffer v1.0
à la Java

class Buffer(capacity: Int) {
    protected val data = new Array[AnyRef](capacity)
    protected var in, out, count = 0

    def put(o: AnyRef): Unit = { ... }
    def get(): AnyRef = { ... }

    def await(cond: => Boolean) = // parameter by name
        while (!cond) { wait() }
}
def put(o: AnyRef) =
  synchronized {
    await(count < capacity)
    data(in) = o
    count += 1; in = (in + 1) % capacity
    if (count == 1) notifyAll()
  }

def get() = {
  synchronized {
    await(count != 0)
    val o = data(out); data(out) = null
    count -= 1; out = (out + 1) % capacity
    if (count == capacity - 1) notifyAll()
    o
  }
}
Producers and Consumers

class Producer(buf: Buffer) extends Runnable {
    new Thread(this).start()

    def run() =
        while(true) buf.put(new Product())
}

class Consumer(buf: Buffer) extends Runnable {
    new Thread(this).start()

    def run() =
        while(true) buf.get()
}
Actors

[Hewitt-Bishop-Steiger:ijcai73, Agha:86]

• Actors are named concurrent autonomous entities with local state that interact with each other through asynchronous message passing (each actor has its own mailbox).

• Fair scheduling: Messages are eventually delivered. No actor can permanently starve. An actor can still misbehave!

• Location transparency and mobility.

Aitors in Scala

- Inherited from Erlang [Armstrong:86]
- Impure in many ways (bad and good)
- Thread-based and event-based actors
- Remote actors (no migration)

P. Haller and M. Odersky, Scala Actors: Unifying thread-based and event-based programming, Theoretical Computer Science, 410(2-3), 2009
Creating Actors

import scala.actors.Actor

class Acteur(name: String, surname: String) extends Actor {
  def act(): Unit = {
    while (true)
      println("Bonjour, je suis " + name + " " + surname)
  }
}

object Take1 {
  def main(args: Array[String]) = {
    new Acteur("Laura", "Smet").start()
    new Acteur("Louis", "Garrel").start()
  }
}

Think about Thread and run() (but Actor is a trait)
import scala.actors.Actor.actor

object Take2 {
    def main(args: Array[String]) = {
        private def loop(name: String, surname: String) = {
            while (true)
                println("Bonjour, je suis " + name + " " + surname)

            actor { loop("Laura", "Smet") }
            actor { loop("Louis", "Garrel") }
        }
    }
}
Sending and Receiving Messages

case object Bonjour

val louis = actor {
  receive {
    case Bonjour => sender ! Bonjour
  }
}

val laura = actor {
  laura ! Bonjour
  receive {
    case Bonjour =>
      case _ => sender ! Malotru
  }
}

sending a message
Syntactic Sugar for Replies

```scala
val louis = actor {
  receive {
    case Bonjour => reply(Bonjour)
  }
}
val laura = actor {
  louis ! Bonjour
  receive {
    case Bonjour =>
    case _ => reply(Malotru)
  }
}

sender ! message □ reply(message)
```
Two-way messages

```scala
val louis = actor {
  receive { case Bonjour => reply(Bonjour) }
}
val laura = actor {
  louis !? Bonjour match {
    case Bonjour =>
    case _ => reply(Malotru)
  }
}

sender ! message ; receive { case r => r}
```
There is more to it

• Forwarding: the expression \textit{actor} \textit{forward message} sends \textit{message} to \textit{actor} on the behalf of \textit{sender}

• Futures: non-blocking version of !?

• Receive with timeout: \texttt{receiveWithin}
Actors, Objects and Threads

• this and self: this denotes the current object and self the current actor (they are different when `actor { body }` is used).

• So far, actors are *thread-based* (each actor is associated a thread) and each thread can be seen as an actor.
Each Thread is an Actor

```scala
object SelfActor {
  def main(args: Array[String]) = {
    self ! Hello
    receive { case Hello => println("I am an actor!") }
  }
}
```
Each Actor is a Thread
(Not quite)

```scala
import scala.actors.Actor._

object Main4 {
    def main(args: Array[String]) =
        println(currentThread);
        actor { println(currentThread) }
}

Output:
Thread[main,5,main]
Thread[ForkJoinPool-1-worker-1,5,main]
```
actor vs actor

Scala shows off

• actor, !, receive are not specific Scala keywords!

• There is not a single mention of actors in The Scala Language Specification.

• Actors are implemented as a library: scala.actor.
The trait and object Actor

trait Actor {
  val mailbox = new Queue[Any]
  def !(msg: Any): Unit = ...
  def receive[R](f: PartialFunction[Any, R]): R = ...
  ...
}

object Actor {
  def self: Actor = ...
  def actor(Body: => Unit): Actor = ...
  ...
}
Message sending

- Enqueues the message in the receiving actor’s mailbox
- If the receiving actor is waiting for the message, the actor is resumed
Message reception

receive \{ f \}

- The mailbox is scanned for expected messages (messages \( m \) such that \( f.\text{isDefinedAt}(m) \) returns true).
- If there is such a message, it is removed from the mailbox and \( f \) is applied to it.
- If not, the actor is suspended.
Event-based vs Thread-Based Actors

- There are two ways to store continuations:
  - as frames on the (actor’s thread) stack: *thread-based actors*
  - as a term on the heap (referenced by an instance variable of the actor): *event-based actors*

- Event-based threads can be seen as event handlers: each execution of the actor’s body is executed by a worker taken from a thread pool.
Syntax

- **Thread-based**

```scala
val louis = actor {
  while (true) {
    receive {
      case Bonjour => sender ! Bonjour
    }
  }
}
```

- **Event-based**

```scala
val louis = actor {
  loop {
    react {
      case Bonjour => sender ! Bonjour
    }
  }
}
```

Huh?
react vs receive

trait Actor {
  def receive[R](f: PartialFunction[Any, R]): R = ...
  def react(f: PartialFunction[Any, Unit]): Nothing = ...
  ...
}

Calling react never returns: the rest of the computation (the continuation) is defined in f.
Message handling

• When the actor is suspended, the continuation is stored in the actor and the thread executing the actor is released.

• A message send checks whether the actor is thread-based or event-based.
  • Thread-based: the underlying thread is resumed.
  • Event-based: a new task (handled by a thread pool) is created with the current continuation as a parameter.
Bounded Buffer v2.0
(Based on example from Scala’s distribution)

class Buffer(size: Int) {
  private case class Put(x: AnyRef)
  private case object Get

  private val buffer = actor { ... }

  def put(o: Object) { buffer !? Put(o) }
  def get: Object = (buffer !? Get).asInstanceOf[AnyRef]
}
private val buffer = actor {
    val data = new Array[Array[AnyRef]](capacity)
    var in, out, count = 0
    loop {
        react {
            case Put(o) if count < capacity =>
                data(in) = o
                count += 1; in = (in + 1) % capacity
                reply()
            case Get if count > 0 =>
                val o = data(out); data = null
                count -= 1; out = (out + 1) % capacity
                reply(o)
        }
    }
}
Comments

• How is it that we don’t need to care any longer about notifications?

• What do we gain?

```java
public class Buffer {
    protected Object[] data;
    protected int in, out, count = 0;
    protected final int capacity;

    public Buffer(int capacity) {
        this.capacity = capacity;
        data = new Object[capacity];
    }

    public synchronized void put(Object o) {
        while (count == capacity)
            try { wait(); }
            catch (InterruptedException ex) {}
        data[in] = o;
        ++count;
        in = (in + 1) % capacity;
        if (count == 1) notify();
    }

    public synchronized Object get() {
        while (count == 0)
            try { wait(); }
            catch (InterruptedException ex) {}
        Object o = data[out];
        data[out] = null;
        --count;
        out = (out + 1) % capacity;
        if (count == 1) notify();
        return o;
    }
}
```

```scala
class Buffer(size: Int) {
  private case class Put(x: AnyRef)
  private case object Get
  private case object Stop

  private val buffer = actor {
    val data = new Array[AnyRef](size)
    var in, out, count = 0
    loop {
      react {
        case Put(o) if count < capacity =>
          data(in) = o
          count += 1; in = (in + 1) % capacity
          reply()
        case Get if count > 0 =>
          val o = data(out); data(out) = null
          count -= 1; out = (out + 1) % capacity
          reply(o)
      }
    }
  }

  def put(o: AnyRef) { buffer !? Put(o) }
  def get: Object = (buffer !? Get)
}```
Buffer as an actor (v3.0)

case class Put(o: Object)
case object Get

class Buffer(size: Int) extends Actor {
  val data = new Array[AnyRef](capacity)
  var in, out, count = 0

  def act() : Unit = {
    loop {
      react {
        // as before
      }
    }
  }
}
class Producer(buf: Buffer) extends Actor {
    def act(): Unit =
        loop { buf !? Put(new Product()) }
}

class Consumer(buf: Buffer) extends Actor {
    def act(): Unit =
        loop { (buf !? Get) }
}
Producers and consumers

v4.0 (fully asynchronous)

case class Put(o: Object)
case object Reply // new
case object Get
case class Reply(o : Object) // new

class Producer(buf: Buffer) extends Actor {
  def act(): Unit = loop {
    buf ! Put(new Product())
    react { case Reply => }
  }
}
class Consumer(buf: Buffer) extends Actor {
  def act(): Unit = loop {
    buf ! Get
    react { case Reply(o) => }
  }
}
Conclusion

• Scala’s extensibility (functions, syntax)

• Scala’s actors are nice but not that easy to use (new programming patterns)

• Possibility of mixing shared memory with threads (eg to deal with blocking IO) and shared nothing with actors (eg to create a large number of concurrent entities)

• Quizz: what about sending/receiving mutable objects?
Conclusion
Some other stuff
to look at

• Scalaz, a library for putting more Haskell in your Scala

• The Typesafe platform: Scala + Akka (distribution) + Play (web programming)
Scala as

- Food for thought
- Food for action (i.e., programming)
It is up to you

• to go farther and higher

• to design and implement even better languages
Mount Everest North Face as seen from the path to the base camp, Tibet. Wikimedia Commons. GNU 1.2.