

Fonction anonymes

Jean-Luc Falcone

24 et 31 mars 2022

Génériques (1)

Types génériques

En Scala, les types génériques se notent avec des crochets:

```
//Classe Générique
```

```
case class Pair[T]( first: T, snd: T )
```

```
//Fonction générique
```

```
def flat[A,B,C]( x: (A,(B,C)) ): (A,B,C) = {  
    val (a,(b,c)) = x  
    (a,b,c)  
}
```

Rien à voir avec les tableaux

- En Java

```
java.util.ArrayList<String>  
java.util.HashMap<String,Date>  
int[]
```

- Mêmes types, en scala

```
java.util.ArrayList[String]  
java.util.HashMap[String,Date]  
Array[Int]
```

Fonctionne par *type erasure*

```
case class Box[T]( value: T )  
def toInt( b: Box[Int] ): Int = b.value  
def toInt( b: Box[String] ): Int = b.value.size
```

```
/* ERREUR COMPILATION  
  |Double definition:  
  |def toInt(b: Box[Int]): Int at line 2 and  
  |def toInt(b: Box[String]): Int at line 3  
  |have the same type after erasure.  
*/
```

Petites questions

Combien d'implémentations possibles pour cette fonction, supposée pure:

```
def foo( a: Boolean, b: Boolean ): Bool
```

Petites questions

Combien d'implémentations possibles pour cette fonction, supposée pure:

```
def foo( a: Boolean, b: Boolean ): Bool
```

et celle-ci ?

```
def foo( a: Int, b: Int ): Int
```

et celle-là ?

Petites questions

Combien d'implémentations possibles pour cette fonction, supposée pure:

```
def foo( a: Boolean, b: Boolean ): Bool
```

et celle-ci ?

```
def foo( a: Int, b: Int ): Int
```

et celle-là ? et celle-ci ?

```
def foo[T]( a: T, b: T ): T
```


Un peu d'OOP: Singletons et traits

- Interfaces en Java

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

- traits en scala

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

Utilisation de traits

- implements en java

```
class Infinite<T> implements Iterator<T> {  
    //...  
}
```

- extends en scala

```
case class Infinite[T]( value: T )  
    extends Iterator[T] {  
    def hasNext = true  
    def next = value  
}
```

Classe statique (java)

```
public class Water {  
  
    public final static double g = 9.81;  
    public final static double density = 1000;  
  
    public static double pressure( double h ) {  
        return waterDensity * g * h;  
    }  
  
}
```

Objet (scala)

Directement une instance, sans classe !

```
object Water {  
  val g = 9.81  
  val density = 1000.0  
  
  def pressure( h: Double ) = density * g * h  
}
```

//Utilisation

```
val p = Water.pressure( 16 ) //Pression d'eau à 16m
```

Singleton (java)

```
public class ItemPriceComparator
implements Comparator<Item> {
    public int compareTo( Item i1, Item i2 ) {
        if( i1.getPrice() < i2.getPrice() ){
            return -1;
        }
        if( i1.getPrice() > i2.getPrice() ) {
            return 1;
        }
        return 0;
    }
}
```

```
object ItemPriceComparator extends Comparator[Item]  
  def compareTo( i1: Item, i2: Item ) =  
    if( i1.price < i2.price ) -1  
    else if( i1.price > i2.price ) 1  
    else 0  
}
```

Static factory (java)

```
public class Username {  
    private String uname;  
    private Username( String u ) {  
        uname = u;  
    }  
    public static Username make( String u ) {  
        check(u);  
        return new Username( u );  
    }  
    private static void check( u ) { /* ... */ }  
}
```


Companion object (scala)

```
//Constructeur privé
case class Username private( username: String )

//Si même nom et même fichier, peut accéder
// aux membres private de Username
object Username {
  def make( u: String ): UserName = {
    check(u)
    new Username(u)
  }
  private def check( u: String ): Unit = ???
}

val u1 = Username.check( "foobar85" )
```

Fonctions anonymes

Fonctions anonymes en Scala

```
(u:User) => u.age >= 18
//type: (User)=>Boolean
(u: User) => u.emailAddress
//type: (User)=>Email
(e:Email) => e.tld == "ch"
//type: (Email)=>Boolean

(i:Int, j:Int) => i + 2*j
//type: (Int,Int)=>Int
```

Syntax alternative

```
(_:User).age >= 18  
(_:Email).tld == "ch"  
  
(_:Int) + 2 * (_:Int)
```

Déclaration

```
val isAdult = (u:User) => u.age >= 18
```

```
val mail = (u: User) => u.emailAddress
```

```
val isSwiss = (e:Email) => e.tld == "ch"
```

```
val f = (i:Int, j:Int) => i + 2*j
```

Utilisation

Déclaration

```
val isAdult = (u:User) => u.age >= 18
val mail = (u: User) => u.emailAddress
val isSwiss = (e:Email) => e.tld == "ch"

val f = (i:Int, j:Int) => i + 2*j
```

Utilisation

```
val alice = User( "Alice",
                  Email( "alice@a.ch" ), 19 )
```

```
isAdult(alice) // true
isSwiss( mail( alice ) ) // true
```

```
f( 2, 5 ) // 12
```

Combiner des fonctions

Combinateurs

$(A) \Rightarrow B$ andThen $(B) \Rightarrow C$: $(A) \Rightarrow C$

$(B) \Rightarrow C$ compose $(A) \Rightarrow B$: $(A) \Rightarrow C$

Exemple

```
val hasSwissMail1 = mail andThen isSwiss
```

```
val hasSwissMail2 = isSwiss compose mail
```

```
hasSwissMail1( alice ) // true
```

```
val g1 = {(i:Int) => i*2} andThen {i => 1.0/i}
```

```
val g2 = {(_:Int) * 2} andThen (1.0/_)
```

- Les fonctions anonymes Scala sont des *closures* (fermetures)
- Elles capturent l'environnement local

```
def adder(n: Int): Int=>Int =  
  i => i + n
```

```
val add2 = adder(2)  
add2(5)    //=> 7  
add2(-2)   //=> 0  
adder(3)(10) //=> 13
```


Implémentation

- Scala est un vrai langage orienté objet
- Donc les fonctions sont aussi des objets
- Classes dont la seule méthode abstraite est `apply`:
 - `()=>O: Function0[+O]`
 - `(A)=>O: Function1[-A,+O]`
 - `(A,B)=>O: Function2[-A,-B,+O]`
 - ...

Méthode `apply`

```
object f {  
  def apply( i: Int ): Double = 1.0 / i  
}  
f.apply( 100 ) // 0.01  
f( 10 )       // 0.1
```

Utilité

Dans presque tous les langages

Javascript

```
var f = function(x) {  
    return x*x + 2*x - 5;  
}
```

Python

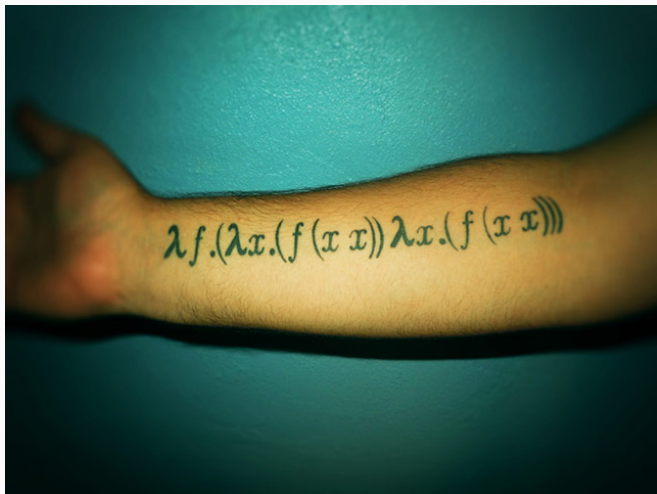
```
f = lambda x: x*x + 2*x - 5
```

C# 3.0

```
Func<int,int> f = x => x*x + 2*x - 5
```

et aussi Java 8, C++11, Perl 5, Matlab, R, etc.

Lambda calcul (Alonzo Church, 1936)



- Permet de représenter toutes les fonctions calculables, avec une algèbre très simple, et une évaluation par substitution.

Exemple d'expression (β -reduction)

- $\lambda x. \lambda f. fx$
- $(\lambda x. \lambda f. fx)y = \lambda f. fy$
- $(\lambda f. fy)\lambda x. xx = (\lambda x. xx)y = yy$

Définitions

- TRUE : $\lambda x. \lambda y. x$
- FALSE : $\lambda x. \lambda y. y$
- NOT : $\lambda p. \lambda a. \lambda b. pba$
- OR : $\lambda p. \lambda q. p p q$

Exemple (NOT TRUE)

$$\begin{aligned} & (\lambda p. \lambda a. \lambda b. pba)(\lambda x. \lambda y. x) \\ &= \lambda a. \lambda b. (\lambda x. \lambda y. x)ba \\ &= \lambda a. \lambda b. (\lambda y. b)a \\ &= \lambda a. \lambda b. b \end{aligned}$$

Fonctions d'ordre supérieur

Fonction dont les arguments et/ou le type de retour est une fonction.

```
val f = (i:Int) => (j:Int) => i*j
```

```
val g = f(3)
```

```
g(10) // 30
```

```
f(2)(5) //10
```

```
val checkTLD =
```

```
  (tld:String) => (e:Email) => e.tld == tld
```

```
val hasFrenchMail = mail andThen checkTLD("fr")
```

```
hasFrenchMail( alice ) //false
```

Fonctions d'ordre supérieur (2)

```
val log = (f: Int => Int) => { i: Int =>
    val j = f(i)
    println( s"INPUT: $i OUTPUT: $j" )
    j
}
```



```
def checkTLD( tld: String ): Email => Boolean =  
  _.tld == tld
```

```
def log[A,B]( f: A=>B ): A=>B = { a =>  
  val b = f(a)  
  println( s"INPUT: $a OUTPUT: $b" )  
  b  
}
```

Modelisation

Modéliser par des fonctions

- Serveur web: $(\text{HTTPRequest}) \Rightarrow \text{HTTPResponse}$
- Client web: $(\text{URL}) \Rightarrow \text{HTTPResponse}$
- Transformer une image: $(\text{Image}) \Rightarrow \text{Image}$
- Format binaires, par exemple PNG:
 - $(\text{BitVector}) \Rightarrow \text{Image}$
 - $(\text{Image}) \Rightarrow \text{BitVector}$
- Jeu d'échec (IA): $(\text{Board}) \Rightarrow \text{Move}$

Exemple conversion d'images (FP-1)

```
def fromJpeg(bits:BitVector): Image
def resize( height: Int, width: Int ): Image=>Image
def blur( radius: Int ): Image=>Image
def rotate( angle: Double ): Image=>Image
def toPng( img: Image ): BitVector
```

Exemple conversion d'images (FP-2)

```
val mod = rotate(90) andThen resize( 1024, 768 )
val pipeline =
    fromJpeg(_) andThen mod andThen toPng

def readBytes( f: File ): BitVector = ???
def writeBytes( f: File, bytes: BitVector ): Unit =
    ???

def convert( in: File, out: File,
    f: (BitVector)=>BitVector ): Unit = {
    val bytesIn = readBytes(in)
    val bytesOut = f( bytesIn )
    writeBytes(out, bytesOut )
}
```

Exemple conversion d'images (OOP-1)

```
trait Transform {  
  def apply( img: Image ): Image  
}  
class Resize( height: Int, width: Int )  
  extends Transform{  
  def apply( img: Image ): Image = ???  
}  
class Blur( radius: Int ) extends Transform {  
  def apply( img: Image ): Image = ???  
}  
class Rotate( angle: Double ) extends Transform {  
  def apply( img: Image ): Image = ???  
}
```

Exemple conversion d'images (OOP-2)

```
trait Decoder {  
  def apply( bs: BitVector ): Image  
}  
trait Encoder {  
  def apply( img: Image ): BitVector  
}  
case object FromJpeg extends Decoder {  
  def apply( bs: BitVector ): Image = ???  
}  
case object ToPNG extends Encoder {  
  def apply( img: Image ): BitVector = ???  
}
```

Exemple conversion d'images (OOP-3)

```
class Combiner( ts: List[Transform] )  
  extends Transform {  
    def apply( img: Image ): Image = {  
      var current = img  
      var rem = ts  
      while( rem.nonEmpty ) {  
        current = rem.head.apply( current )  
        rem = rem.tail  
      }  
      current  
    }  
  }  
  
val mod = new Combiner (   
  List( new Rotate(90), new Resize( 1024, 768 ) )  
)
```


Exemple conversion d'images (OOP-4)

```
class Pipeline( dec: Decoder, enc: Encoder,
               trans: Transform ) {
  def run( bytes: BitVector ): BitVector =
    enc.apply( trans.apply( dec.apply( bytes ) ) )
}

val pipeline =
  new Pipeline( FromJpeg, mod, ToJpeg )

def convert(in:File,out:File,p:Pipeline):Unit={
  val bytesIn = readBytes(in)
  val bytesOut = p.run( bytesIn )
  writeBytes(out, bytesOut )
}
```

Simplification



```
val pw = new PrintWriter( "file1.txt" )
try {
    dumpData( pw )
} finally {
    pw.close()
}
```

Gestion des IO (après- FP)

```
def withPrintWriter( name: String ) =  
  ( f: PrintWriter=>Unit ) => {  
    val pw = new PrintWriter( name )  
    try {  
      f( pw )  
    } finally {  
      pw.close()  
    }  
  }  
}
```

```
withPrintWriter( "file1.txt" )( dumpData )  
withPrintWriter( "file2.txt" ){ pw =>  
  pw.println( "Hello" )  
  pw.println( "world" )  
}
```

Quel est le type de retour de `withPrintWriter` ?

```
def withPrintWriter( name: String ) =  
  ( f: PrintWriter=>Unit ) => {  
    val pw = new PrintWriter( name )  
    try {  
      f( pw )  
    } finally {  
      pw.close()  
    }  
  }  
}
```

Gestion des IO (après- FP)

Quel est le type de retour de `withPrintWriter` ?

```
def withPrintWriter( name: String ) =  
  ( f: PrintWriter=>Unit ) => {  
    val pw = new PrintWriter( name )  
    try {  
      f( pw )  
    } finally {  
      pw.close()  
    }  
  }  
  
(PrintWriter=>Unit)=>Unit
```

Syntaxe pratique (1)

Scala permet de définir des méthodes qui prennent plusieurs listes d'arguments:

```
def add( i: Int )( j: Int ) = i+j  
def foo[A,B,C]( a: A )( f: (A,B)=>C ): B=>C =  
  b => f(a,b)
```

```
add(1)(2) //=> 3  
val twice = foo( 2 )( _ * _ )  
twice( 4 ) //=> 8
```


Syntaxe pratique (2)

Scala permet de remplacer les parenthèses par des accolades dans certains cas (1 seul argument):

```
def foo[A,B,C]( a: A )( f: (A,B)=>C ): B=>C =  
  b => f(a,b)
```

```
val bar = foo(true){ (b:Boolean,s:String) =>  
  if(b) s else ""  
}  
bar("hop") //=> "hop"
```

Gestion des IO (après- FP 2)

```
def withPrintWriter( name: String ) =  
  ( f: PrintWriter=>Unit ): Unit = {  
    val pw = new PrintWriter( name )  
    try {  
      f( pw )  
    } finally {  
      pw.close()  
    }  
  }  
}
```

```
withPrintWriter( "file1.txt" )( dumpData )  
withPrintWriter( "file2.txt" ){ pw =>  
  pw.println( "Hello" )  
  pw.println( "world" )  
}
```

Gestion des IO (après - OOP 1)

```
trait PWProcessor {  
  def process( pw: PrintWriter ): Unit  
}  
  
def withPrintWriter( name: String )  
  ( pwp: PWProcessor ): Unit = {  
  val pw = new PrintWriter( name )  
  try {  
    pwp.process( pw )  
  } finally {  
    pw.close()  
  }  
}
```

```
withPrintWriter( "file1.txt" )( new PWProcessor {  
    def process( pw: PrintWriter ) = dumpData(pw)  
})
```

```
withPrintWriter( "file2.txt" )( new PWProcessor {  
    def process( pw: PrintWriter ) = {  
        pw.println( "Hello" )  
        pw.println( "world" )  
    }  
})
```

Exemple: transactions de bases de données

```
trait Database {  
  def transaction: Transaction  
  def close: Unit  
}  
  
object Database {  
  def connect( host: Host ): Database  
}  
  
trait Transaction {  
  def execute( query: SQL ): List[Result]  
  def commit: Unit  
}
```

Example: Utilisation

```
val db = Database.connect( "machin.org" )

val tx = db.transaction
val res1 = tx.execute( query1 )
val res2 = tx.execute( query2(res1) )
tx.commit

val tx2 = db.transaction
val res3 = tx.execute( query3 )
val res4 = if( res3.isEmpty )
            tx.execute( query4 )
        else
            tx.execute( query5 )
tx2.commit
```

Exemple: Utilisation (debugé !)

```
val db = Database.connect( "machin.org" )
```

```
val tx = db.transaction
```

```
val res1 = tx.execute( query1 )
```

```
val res2 = tx.execute( query2(res1) )
```

```
tx.commit
```

```
val tx2 = db.transaction
```

```
val res3 = tx2.execute( query3 )
```

```
val res4 = if( res3.isEmpty )  
            tx2.execute( query4 )
```

```
    else
```

```
        tx2.execute( query5 )
```

```
tx2.commit
```

Exemple: transactions de bases de données (2)

```
object Database {  
  private def doConnect( host: Host ): Database  
  def connect[A]( host: Host )( body: Database=>A )  
    val db = doConnect( host )  
    val a = body( db )  
    db.close  
    a  
}
```


Exemple: transactions de bases de données (2)

```
trait Database {  
  protected def start: Transaction  
  def transaction[A]( body: Transaction=>A ): A = {  
    val tx = start  
    val a = body( tx )  
    tx.commit  
    a  
  }  
  private def close: Unit  
}
```

Exemple: utilisation (2)

```
Database.connect( "machin.org" ){ db =>
  val (r1,r2) = db.transaction { tx =>
    val res1 = tx.execute( query1 )
    val res2 = tx.execute( query2(res1) )
    (res1,res2)
  }
  val r4 = db.transaction { tx =>
    val res3 = tx.execute( query3 )
    val res4 = if( res3.isEmpty )
      tx.execute( query4 )
    else
      tx.execute( query5 )
    res4
  }
  (r1,r2,r4)
```