Fonctions Lambda

Jean-Luc Falcone

HEPIA - 2014

Fonctions anonymes

```
(i: Int) => i+1                 // (Int)=>Int

(i: Int, j: Int ) => i+j        // (Int,Int)=>Int

(_:Int) + 1                     // (Int)=>Int

(_:Int) + (_:Int)               // (Int,Int)=>Int
```

Utilisation

```
val inc = (i: Int) => i+1
val add = (i: Int, j: Int ) => i+j

val triple = (_:Int) * 3
val div = (_:Int) / (_:Int)

val f = inc.andThen(triple)
val g = (i:Int,j:Int) => add( f(i), div(i,j) )
```

Evaluation de fonction

```
f(3)          //=> 12
add(2,3)      //=> 5
div(12,2)     //=> 6
g(4,2)        //=> 17
```

Problème

- Pour calculer le montant total d'une commande
  - On multiplie le nombre d'unités par le coût
  - On applique un rabais
- Le rabais peut être:
  - Inexistant
  - Une somme fixe (pe: -25 CHF)
  - Un pourcentage (pe: -15%)
  - Un nombre d'unités offertes à partir d'un certain volume (pe: la onzième offerte)

Solution Orientée-Objet

```
trait Rebater {
  def apply( d: Double ): Double
}
object NoRebate extends Rebater{
  def apply( d: Double ) = d
}
case class Percent( p: Double ) extends Rebater {
  val factor = 1 - p/100
  def apply( d: Double ) = d * factor
}
def total( units: Int, cost: Double, rebate: Rebate ) = {
  val total = units*cost
  rebate.apply( total )
}
total( 3, 105.00, Percent(15) )
```

Solution Fonctionelle

```
def total( uts: Int, c: Double, rebate: Double=>Double ) =
  val total = units*cost
  rebate( total )
}

total( 3, 105.00, (d:Double) => d-25 )
total( 3, 105.00, (_:Double) * (1-0.15) )

total( 3, 105.00, d => d-25 )
total( 3, 105.00, _ * 0.85 )

total( 3, 105.00, identity  )
```

Solution Fonctionelle: Multi param lists

```
def total( uts: Int, c: Double)( reb: Double=>Double ) = {
  val total = units*cost
  rebate( total )
}

val customer: Customer = ...

total( 3, 105.00 ){ t =>
  val amount = customer.pastTotalOrders
  if( amount > 5000 ) t * 0.9
  else t
}
```

Solution Fonctionelle: exemples (2)

```
val noRebate = (d:Double) => d
total( 3, 105.00 )( noRebate )

def percent( p: Double ): Double=>Double = {
  val factor = 1 - p/100
  d => d*factor
}
total( 3, 105.00 )( percent(15) )
```

Solution Fonctionelle: exemples (3)

```
def whenHigh( treshold: Double,
              rebate: Double=>Double): Double=>Double =
  total => if( total >= threshold ) rebate(total)
           else total

val vipOnly = whenHigh( 500, percent(10) )

total( 3, 105.00 )( vipOnly )
```

Base de donnée (style impératif)

```
object DataBase {
  def connect(...): DataBase
}
trait DataBase {
  def transaction: Transaction
  def disconnect: Unit
}
trait Transaction {
  def query( sql: SQL ): Result
  def update( sql: SQL ): Boolean
  def commit: Unit
}
```

Utilisation (style impératif)

```
val db = DataBase.connect( ... )

val tx1 = db.transaction
val i = tx1.query(...) //READ
tx1.update( ... )      //WRITE
tx1.commit

val tx2 = db.transaction
val j = tx2.query(...)  //READ
tx2.update( ... )       //WRITE
tx2.commit

db.disconnect
```

Base de donnée (style hybride)

```
object DataBase {
  def connect(...): DataBase
  def withConnection(...)( body: DataBase=>Unit) {
    val db = connect(...)
    body(db)
    db.disconnect
  }
}
trait DataBase {
  def transaction: Transaction
  def disconnect: Unit
  def withTransaction( ops: Transaction=>Unit ) {
    val tx = transaction
    ops(tx)
    tx.commit
  }
}
```

Utilisation (style hybride)

```
DataBase.withConnection( ... ){ db =>

  db.withTransaction { tx =>
    val i = tx.query(...) //READ
    tx.update( ... )      //WRITE
  }

  db.withTransaction { tx =>
    val i = tx.query(...) //READ
    tx.update( ... )      //WRITE
  }

}
```