LA PROGRAMMATION FONCTIONNELLE EN JAVA

LES LAMBDA-EXPRESSIONS

Younès EL AMRANI

2015-2019

Introduction

- La programmation fonctionnelle en JAVA a fait initialement son apparition en JAVA en 2015. Elle apporte avec elle d'immenses possibilités en termes de flexibilité du langage et de modélisation des programmes. Cependant, il faut noter que JAVA est né dedans : c'est à l'origine une implémentation de OAK Lisp, bien qu'ayant relégué son patrimoine génétique aux oubliettes ; force est de constater qu'après 20 ans (de 1995 à 2015) les traits caractéristiques de la programmation fonctionnelle sont revenues tracer sur la face de JAVA les stigmates d'un héritage codé au plus profond de sa machine virtuelle.

Définition de la programmation fonctionnelle

- La programmation fonctionnelle dénote les langages capables de traiter les fonctions comme des citoyennes de première zone.
- La définition d'un élément du langage comme citoyen de première zone signifie qu'il peut se trouver et se retrouver dans n'importe qu'elle structure du langage :
  - Dans une affectation : aussi bien à gauche qu'à droite
  - Dans une liste de paramêtres
  - Dans un retour de fonction : comme valeur de retour

Comment faire de la programmation fonctionnelle

Pour faire de la programmation fonctionnelle, il faut être capable d'utiliser les fonctions comme des paramêtres, ou des valeurs de retour d'une autre fonction : des langages insoupconnés sont fonctionnels : cas de C !

// Illustration en C d'une fonction en right-hand side :

```
typedef int (*Tii)(int) ;
int f(int x){…}
int g(int x){ Tii F=f ;  F(10) ;}                    // f est en rhs
// Illustration en C d'une en paramêtres
int h(Tii F, int n){return F(n) ;}
// pour appel dans le main on écrit h(f,10)
int main( ) { h(f,10) ; }
// h est ici une fonction d'ordre 2 ! (en C!)
```

Quels types de fonctions en JAVA ?

- Depuis 1995, nous avons disposé en JAVA UNIQUEMENT des méthodes dans les classes pour réaliser des opérations.
- Les méthodes ne peuvent pas voir leur valeur (en tant que fonction) modifiée.
- Une méthode est définie par une séquence d’opérations qui ne peut être changée.
- Si on considère la classe comme une produit cartésien de valeurs scalaires et fonctionnelles, la méthode est une valeur fonctionnelle constante !

Que nous apportent les lambdas de java ?

Les lambdas en JAVA SONT DES FONCTIONS inspirées des langages fonctionnelles et introduisent au sein de JAVA, plusieurs possibilités :

- Affecter une lambda en partie droite de l'affectation
- Passer en paramêtre une lambda
- Retourner une lambda en valeur d'une autre fonction/méthode/lambda-expression
- Modifier la valeur d'une lambda après sa définition
- Passer en paramêtre une lambda à une autre lambdas : ce sont des fonctions d'ordre supérieur !

Pour définir une lambda expression, on utilise une interface fonctionnelle

- Les interfaces fonctionnelles sont des interfaces de JAVA qui donne la signature, prototype de la lambda expression : UNE ET UNE SEULE signature doit être donnée dans une interface fonctionnelle, exemple :

```
interface iddrd {double call(double x, double y) ;}
```

Pour définir une lambda expression, on peut utiliser une interface fonctionnelle

```
interface iddrd {double call(double x, double y) ;}
```

Avec l'interface fonctionnelle ci-dessus on peut définir des lambdas comme suit :

```
iddrd f1=(x1,x2)->  x1+x2 ;

Iddrd f2=(x1,x2)->  Math.pow(x1,x2) ;
```

On peut aussi affecter des lambdas expressions entre-elles :
f1=f2 ;
ou même :
f2=f1 ;

Utilisation des lambdas comme paramêtre

On commence par déclarer une interface fonctionnelle :

```
interface Lambda_DrD{    double call(double d);}
```

Voici alors comment passer une lambda en paramêtre:

```
double F1(Lambda_DrD f, double x1, double x2)
{        return(f.call(x1)+f.call(x2));                    }
```

Dans un second exemple, ci-dessous, on illustre à la fois le passage dans la liste des paramêtres, mais aussi comme valeur de retour de la méthode nommée setf3 :

```
Lambda_DrD setf3(Lambda_DrD f3)
{   return this.f3=f3;                    }
```

Utilisation des lambdas valeur de retour

On utiliser l'interface fonctionnelle suivante:

```
interface Lambda_DrD{    double call(double d);}
```

on illustre ci-dessous à la fois le passage dans la liste des paramêtres, mais aussi comme valeur de retour de la méthode nommée setf3 :

```
Lambda_DrD setf3(Lambda_DrD f3)
  {   return this.f3=f3;                    }
```

Les lambda-expressions pour le style fonctionnel

- Les lambdas-expressions permettent au programmeur un nouveau style de programmation, directement inspiré des langages fonctionnels, plus souple, plus intuitif
- Les lambdas expressions permettent d’obtenir des classes où les méthodes elles-mêmes peuvent céder la place aux lambda-expressions et devenir modifiables comme tout autre attribut

Les lambdas remplaceraient les méthodes et seraient plus flexibles

- Les lambdas expressions permettent d'obtenir des classes où les méthodes elles-mêmes peuvent céder la place aux lambdas
- Les lambdas sont modifiables comme tout autre attribut, pourraient être passer comme paramêtre et retournées comme résultat : sont-elles de super-méthodes ?
- Vers une autre manière de penser la méthode dans la classe : une manière plus souple et plus ouverte aux modifications, aux changements !

Initialisation des lambdas dans un constructeur

- Les lambdas expressions peuvent être initialisées dans le constructeur de la classe comme n'importe quelle autre valeur d'attribut :

```
1. interface Lambda_DrD{     double call(double d); }
2. class C1{
3.   Lambda_DrD f1,f2,f3;
4.   public C1 (  ){
5.            f1=x1->x1+1.0;
6.            f2=x1->2.0*x1+3.0;
7.            f3=x1->Math.pow(x1,2.0)+1.0;
8.   }
9. Lambda_DrD setf1(Lambda_DrD f1)
10. {          return this.f1=f1;     }
```

Comment modifier une lambda

1. *interface Lambda_DrD{ double call(double d);}*
2. class C1{
3. Lambda_DrD f1 ; // Définition d'une lambda
4. public C1( ) { f1=x1→x1+1.0;} // constructeur

Illustration de Modifications dans le main :

1. public static void main(String[ ] args){
2. C1 v1=new C1();
3. v1.setf1(x→x+100.0);***// ici on modifie f1***
4. d1=v1.f1.call(2.0) ;

Lambdas sans paramêtres

```
class C1{
    double a=2,b=3,c=5;
    Lambda_voidrD g1,g2,g3;
    public C1(){
        g1=()->Math.pow(a,2.0);
        g2=()->Math.pow(b,2.0);
        g3=()->Math.pow(c,2.0);
    }
```

Tableaux de Lambdas

Il est possible de définir des tableaux de Lambdas, exemple de définition lors de la déclaration :

```
Lambda_DrD[ ] tablam=
    {x1->x1+1,
     x1->x1+10,
     x1->x1+100,
     x1->Math.pow(x1,2.0),
     x1->Math.pow(x1,2.0)+x1+1};
```

Utiliser un tableau de Lambda sur une valeur

```
System.out.print("provide xx=");
double xx=scan.nextDouble();
// Ttableau de double où stocker le résultat
Double[ ] res=new double[tablam.length];
// Appel des lambdas et stockage du résultat
for(int i=0;i<tablam.length;i++){
    res[i]=tablam[i].call(xx);
}
```

Affichage du résultat de l’application d’un tableau de lambdas

On utilise la classe java.util.Arrays

Qui contient plusieurs fonctions prédéfinies pour

Afficher, convertir en chaînes de caractère, trier, etc.

```
System.out.print(java.util.Arrays.toString(res));
```

Fonctions d’ordre 2

Les fonctions d’ordre 2 sont celles qui prennent en paramêtre une autre Lambda : en particulier pour l’appliquer sur une valeur ou sur un ensemble de valeurs.

Définition d’une Lambda d’ordre 2

```
double F1(Lambda_DrD f, double x1, double x2)
{
    return(f.call(x1)+f.call(x2));
}
```

La lambda f est appliquée à x1 et x2 : elle est elle-même un paramêtre, donc F1 est d’ordre 2

Utilisation d’une fonction d’ordre 2

```
////////
// - Test de F1, fonction de second ordre
System.out.println("\n// - Test de fonction de second ordre\n");
////
```

```
double d0=v1.F1(x1->x1+2,3.0,2.0);
```

```
System.out.println("v1.F1(x1->x1+2,3.0,2.0) == "+d0);
```

Utilisation d’une fonction d’ordre 2 : changement du paramêtre

```
double d0=v1.F1(x1->x1+2,3.0,2.0);
System.out.println("v1.F1(x1->x1+2,3.0,2.0) == "+d0);
```

```
double d01=v1.F1(x1->Math.pow(x1,2.0),2.0, 3.0);
System.out.println("F1(x1->Math.pow(x1,2.0),2.0, 3.0) == "+d01)
```

Modification de la valeur d’une Lambda par affectation

```
Lambda_DrD setf1(Lambda_DrD f1){
        return this.f1=f1;
   }
```

```
v1.setf1(x->x+10.0);
d1=v1.f1.call(2.0);
System.out.println("(x->x+10.0) (2.0) == "+d1);
```

Interfaces Fonctionnelles versus les autres Interfaces de JAVA

1. Pour utiliser les lambdas avec la déclaration dans une interface alors il faut se restreindre à une seule déclaration par interface car seules les interfaces avec une seule déclaration fonctionnent: si on met deux déclarations dans une interface, les lambdas ne fonctionneront plus avec cette interface.

2. On peut considérer qu'une interface avec une seule fonction constitue la défintion d'un ensemble de fonctions et on ne peut pas définir deux ensembles de fonctions différents dans la même déclaration d'interface: d'un point de vue mathématique une interface fonctionnelle définit le domaine et l’image d’un ensemble de fonctions.

3. Si on souhaite définir N ensembles de fonctions, il faudra utiliser N définition d'interfaces fonctionnelles

Définition d'Interfaces Fonctionnelles de |R⁴ → |R

```
interface DomainDoubleRangeDouble{    double f(double x);}

interface DomainDoubleDoubleRangeDouble

{    double f(double x, double y);}

interface DomainDoubleDoubleDoubleRangeDouble

{    double f(double x, double y);}

interface dddrd extends DomainDoubleDoubleDoubleRangeDouble{}//
   Renommage

interface DomainDoubleDoubleDoubleDoubleRangeDouble

{        double f(double x, double y);}

interface ddddrd extends
   DomainDoubleDoubleDoubleDoubleRangeDouble{}//Rename
```

Renommage d’une Interface Fonctionnelle par héritage

```
interface DomainDoubleDoubleDoubleRangeDouble
{    double f(double x, double y);}

// Renommage de l’interface fonctionnelle par voie
// d’héritage (s’inspirer du typedef de C)
interface ddddrd extends
   DomainDoubleDoubleDoubleDoubleRangeDouble
{ /*Aucune extension proposée*/      }
```

Utilisation de tableaux de Lambdas

Les lambdas, citoyennes de première zone, apparaissent aussi dans des tableaux !

```
1. interface Lambda_printText {String say();}
2. class AI{
3.    Lambda_printText[ ]    tab6=
4.    {()->"text1",()->"text2",()->"text3",()->"text4"};
5.   public void test() {
6.          for(int i=0;i<4;i++)
7.          System.out.println(tab6[i].say()); }
```

implémentation de contraintes avec des lambda-expressions

Objectif: Définir plusieurs interfaces fonctionnelles pour définir des contraintes

Double -> Boolean,

Double²->Boolean,

Double⁴->Boolean,

DoubleX...->Boolean

Exemple d'Interface fonctionnelle Pour Implémenter des contraintes

```
interface IDrB{Boolean    call(Double x1);}
interface IDDrB
  {Boolean   call(Double x1, Double x2);}
interface IDDDrB
  {Boolean  call(Double x1, Double x2, Double x3);}
interface IDDDDrB
  {Boolean call(Double x1, Double x2, Double x3, Double x4);}
interface IDxNrB{Boolean  call(Double[ ] x);} // sur |R100
```

Contraintes sur un espace de Dimension N

Pour un espace de dimension N, on utilisera un tableau de Double

interface IDNrB{ Boolean call(Double[] X); }

Les Lambdas «apply » qui appliquent un paramêtre aux autres

```
////////
//  Problème : On souhaite Définir une lambda d'ordre 2
//      qui prend en paramêtre une contrainte
//      puis l'applique à son second argument :
//   Réponse :    ce sont des Fonctions d'ordre 2 qui
//      prennent leur valeurs dans Boolean
////


interface ApplyIDrB { Boolean call(IDrB  f, Double x1);}

interface ApplyIDDrB{ Boolean call(IDDrB f, Double x1, Double x2);}
```

Définition d’un ensemble de contraintes sur un point

```
////////
// Question : définir uneinterface fonctionnelle pour
// une lambda qui applique N contraintes sur un point
// dans |R¹ :
////


interface ISysConstDrB{
    Boolean call(IDrB[  ] sysc, double x1);
}
```

Lambda qui applique N contraintes sur |R³

```
////////
// Question : définir une interface fonctionnelle d'une lambda
// qui applique N contraintes sur un espace de 3 dimensions
////
interface ISysConstDDDrB{
    Boolean call(IDDDrB[  ] sysc,
                        double x1, double x2, double x3);
}
```

Définition de Map : qui applique une contrainte à un sous-ensemble de|R

```
//////////////
// Question:  définir La fonction Map qui applique une fonction
// sur un ensemble ordonné de valeurs et retourne comme résultat
//  un ensemble ordonné de résultats selon le même ordre!
////

////////
//     Pour un sous-ensemble de N valeurs réelles, on utilise un tableau.
interface MapIDrB{Boolean[ ] call(IDrB f, Double[ ] x);}
////

//      le Map va appliquer la même contrainte IdrB sur un ensemble x ordonnée de points,
//      puis le Map va retourner un ensemble ordonné de Booléens qui sont le résultat
//      de la contraintes, sur chacun des points en entrée.
```

Application d'un ensemble d'une contrainte à 100 points de |R³

////////

// Question : définir un Map Pour un ensemble de Points

// sur un espace de dimension N

// Réponse : utiliser un tableau à deux dimensions:

// la première dimension sera utilisée pour déterminer le nombre de point

// La seconde dimension du tableau sera utilisée pour les coordonnées

// Exemple avec |R³ :

interface MapIDDrB{Boolean[ ] call(IDDrB f, Double[ ][ ] x);}

On aura à l'utilisation Double[ ][ ] x=new Double[100][3]

pour représenter 100 points de |R³

Exemples de contraintes

```
////////
// Définitions de contraintes sur IR, IR², IR³, IR⁴
////
IDrB     c1=x1          -> x1>4.0;
IDDrB    c2=(x1,x2)     -> (3.0*x1+5.0*x2)>11;
IDDDrB   c3=(x1,x2,x3)  -> (5.0*x1+7.0*x2+11.0*x3)>100.0;
IDDDDrB  c4=(x1,x2,x3,x4)-> (5.0*x1+7.0*x2+11.0*x3-100.0*x4)<1000.0;
```

Définition de 2 lambdas d’ordre 2

```
////////
// Définition de deux interfaces fonctionnelles
//  d'application d'une contrainte d'ordre 2
////
    ApplyIDrB apply=(c,x1)->c.call(x1);
    ApplyIDDrB apply2=(c,x1,x2)->c.call(x1,x2);
```

Définition de la fonction Map |P(|R)

```
////////
// Utilisation de la fonction Map
// sur un ensemble ordonné de valeurs scalaires
////
MapIDrB map=(c,x)->{
 Boolean[] r= new Boolean[x.length];
 for(int i=0;i<x.length;i++){
     r[i]=c.call(x[i]);
 }
 return r;
};
```

Définition de Map sur |P(|R²)

```
////////
// Utilisation de la fonction Map sur une ensemble ordonné de Point dans IRxIR
////
MapIDDrB map2=(c,x)->{
Boolean[] r=new Boolean[x.length];
for(int i=0;i<x.length;i++){
  // Le second indice est utilisé pour l'oordonnée
  r[i]=c.call(x[i][0],x[i][1]);
} // end for
return r;
};  // end MapIDDrB
```

Définition d’un sous-ensemble de |R ordonné

```
////////
// Initialisation d'un tableau de valeurs rééelles
// à la ligne de la déclaration d'un tableau de |R
////
Double[ ] tabxx={2.0,3.0,4.0,5.0};
```

Définition d’un élément de |P(|R²)

```
///////
// Voici l'initialisation d'un tableau de dimension [5][2] lors de la
   déclaration
// Nous avons dans ce cas 5 points de |R²
//  p1=(11,12) ; p2=(21,22) ; p3=(31,32) ; p4=(41,42) ; p5=(51,52)
////
Double[ ][ ] tabyy={{11.0,12.0},{21.0,22.0},{31.0,32.0},{41.0,42.0},
                    {51.0,52.0}
                    };
```

Initialisation d’un élément de |R³

```
////////
// Initialisation d'un tableau de dimension [5][3]
// La seconde dimension du tableau est utilisée pour les coordonnées (x1,x2,x3)
// La première dimension du tableau est utilisée pour ordonner les points
// et déterminer leur nombre: p1, p2, p3, p4, p5 (pour un ensemble de 5 Points)
////
Double[  ][  ] tabzz={{11.0,12.0,13.0},  {21.0,22.0,13.0},  {31.0,32.0,13.0},
                      {41.0,42.0,13.0},  {51.0,52.0,13.0}
                     };
```

Exemple d’une classe avec plusieurs contraintes

```
class C5{
   ////////
   // Définitions de contraintes sur IR, IR², IR³, IR⁴
   ////
   IDrB     c1=x1          -> x1>4.0;
   IDDrB    c2=(x1,x2)     -> (3.0*x1+5.0*x2)>11;
   IDDDrB   c3=(x1,x2,x3)  -> (5.0*x1+7.0*x2+11.0*x3)>100.0;
   IDDDDrB  c4=(x1,x2,x3,x4)-> (5.0*x1+7.0*x2+11.0*x3-100.0*x4)<1000.0;
   ////////
   // Définition de deux interfaces fonctionnelles d'application d'une contrainte d'ordre 2
   ////
   ApplyIDrB apply=(c,x1)->c.call(x1);
   ApplyIDDrB apply2=(c,x1,x2)->c.call(x1,x2);
```

Exemple d’utilisation de apply

```
C5 v5=new C5();
Scanner scan=new Scanner(System.in);
System.out.print("Donner l'argument=");
Double xx=scan.nextDouble();
Boolean b1=v5.apply.call(x->x+1>2, xx);
// Affichage du résultat qui rappelle l’instruction
System.out.println("(x->x+1>2)("+xx+")="+b1);
```

Modification du paramêtre de apply

```
Boolean b2=v5.apply.call(
 x->Math.sqrt(x)*Math.pow(x,2.5)+3.0>178.0,
                                        xx);
// Affichage du résultat qui affiche l'instruction
System.out.println("(x-
   >Math.sqrt(x)*Math.pow(x,2.5)+3.0>10)("+xx+")="
      +b2);
```

Exemple d’utilisation de Map

```
/////////
// Fonction d'ordre #2
////
System.out.println("-EXEMPLE AVEC FONCTION D'ORDRE 2-");
Double[ ] tabxx={2.0,3.0,4.0,5.0};
Boolean[  ] tabb1=v5.map.call(x->x%2==0, tabxx);
// Affichage avec rappel de l’instruction
System.out.println("(x->x%2==0)("
      +java.util.Arrays.toString(tabxx)+")="
      +java.util.Arrays.toString(tabb1));
```

Prérequis : COURS THREADS Utilisation Runnable avec Lambdas

```
// Question : comment utiliser une Lambda
//    pour initialiser un Runnable ?
// Réponse : Initialiser un Runnable avec une Lambda
//     comme suit :
Runnable r1=()->{for(int i=0;i<N;i++)
   System.out.println("Salam");};
// Lancement du Runnable avec un bloc try-catch
Thread t1 = new Thread(r1) ;
t1.start( ) ;
```

Prérequis : COURS COMPARATOR
Comparator avec Lambda selon X

```
import java.util.Comparator;
import java.util.Arrays;
// Note : Comparable est déjà dans java.lang
class Point {double X, Y, Z;}
Comparator<Point> selonX =    // On donne une lambda comme valeur
(p1,p2) ->                    // p1 et p2 sont deux Points (X,Y,Z)
   {double x1=p1.X,   x2=p2.X;
   if(x1>x2)              return 1;
   else {if (x1==x2)    return 0;
   else                   return -1;
   }
};
```

Prérequis : COURS COMPARATOR Comparator avec Lambda selon Y

```
import java.util.Comparator;
import java.util.Arrays;
// Note : Comparable est déjà dans java.lang
class Point {double X, Y, Z;}
Comparator<Point> selonY =    // On donne une lambda comme valeur
(p1,p2) ->                          // p1 et p2 sont deux Point
   {double y1=p1.Y,   y2=p2.Y;
   if(y1>y2)              return 1;
   else {if (y1==y2)    return 0;
   else                     return -1;
   }
};
```

Prérequis : COURS COMPARATOR
Comparator avec Lambda selon Z

```
import java.util.Comparator;
import java.util.Arrays;
// Note : Comparable est déjà dans java.lang
class Point {double X, Y, Z;}
Comparator<Point> selonZ =    // On donne une lambda comme valeur
(p1,p2) ->                    // p1 et p2 sont deux Points (X,Y,Z)
   {double z1=p1.Z,   z2=p2.Z;
   if(z1>z2)              return 1;
   else {if (z1==z2)    return 0;
   else                  return -1;
   }
};
```

Prérequis : COURS COMPARATOR Utilisation du Comparator selonY

```
// Tri d’un tableau tabPoint avec le
// Comparator selonY défini comme une Lamda
java.util.Arrays.sort(tabPoint, selonY);
for(int i=0;i<tabPoint.length;i++)
   System.out.println(tabPoint[i]);
```

Prérequis : COURS COMPARATOR Utilisation du Comparator selonX

```
// Tri d'un tableau tabPoint avec le
// Comparator selonY défini comme une Lamda
java.util.Arrays.sort(tabPoint, selonX);
for(int i=0;i<tabPoint.length;i++)
   System.out.println(tabPoint[i]);
```

Prérequis : COURS COMPARATOR Utilisation d'un Comparator dans l'instruction d'appel

```
//Maintenant, il est possible de définir le Comparator, dans
   l'appel même de Arrays.sort, sous forme de lambda

java.util.Arrays.sort(tabPoint, (p1,p2)-> {double
   z1=p1.Z,z2=p2.Z;if(z1>z2) return 1; else {if(z1==z2)
   return 0; else return -1;}});

for(int i=0;i<tabPoint.length;i++)
   System.out.println(tabPoint[i]);
```

Conclusion

1. L’introduction des lambda-expressions en JAVA et le renforcement de la programmation fonctionnelle ouvre de multiples possibilités comme nous l’avons vu : mais ce n’est pas tout, c’est le style de programmation entier qui est invité à revisiter son architecture et à promouvoir la fonction au premier rang et à l’introduire dans les Collections pour une utilisation nouvelle, flexible, imaginative et dynamique à la recherche d’algorithmes encore à découvrir.

Perspectives

La programmation fonctionnelle en JAVA : sa découverte permet de se convaincre que les paradigmes de la programmation son perméables et évolutifs : les langages, plus que jamais, semblent capables de se convertir, s'étendre, se diversifier en termes de constructions, locutions, mots-clefs, mécanismes d'héritage de prototypage et d'autres encore inconnus. Finalement, le langage doit permettre au programmeur de s'exprimer comme il le pense sans que jamais la syntaxe ou les constructions de base ne deviennent des obstacles entre l'algorithme imaginé par le programmeur et la machine sur lequel il va prendre vie.