Typage et Analyse Statique

Cours 5

Emmanuel Chailloux

Spécialité Science et Technologie du Logiciel
Master mention Informatique
Université Pierre et Marie Curie

année 2017-2018

Plan du cours 5

Programmation par objets:

- Surcharge en Java
  - par l'exemple
  - algorithmes de résolution
- Classes de type en Haskell

Résolution de la surcharge

- choix du type de la méthode à employer lors d'un appel de méthode
- résolue STATIQUEMENT selon une relation d'ordre sur la classe de définition et le type des arguments
- le type du résultat n'est pas pris en compte
- il y a des cas où la résolution échoue

```
1  class A :  m2(A) m2(A,A)
2  class B :  m2(B) m2(A,B) m2(B,A)
3  class C :  m2(A) m2(B,C) m2(C,A)
```

avec C qui hérite de B qui hérite de A.

```
1  A a1 = new A();
2  B b1 = new B();
3  C c1 = new C();
4  c1.m2(x,y)
```

Quel est le type de la méthode à utiliser en fonction des types de x et y ?

Sélection du type de la méthode (1)

Au niveau de la classe C on a 8 méthodes dont 5 à 2 arguments

	classe de définition	type de la méthode
m2	A	(A)
	A	(A,A)
	B	(B)
	B	(A,B)
	B	(B,A)
	C	(C)
	C	(B,C)
	C	(C,A)

Sélection du type de la méthode (2)

```
1  c1.m(x,y);
```

Sur l'ensemble des méthodes `m2` du receveur (ici `c1`), on ne conserve que celles dont le type (t1,t2) vérifie :

- type de x $\leq t_1$
- type de y $\leq t_2$

c1.m2(a1,b1) ;

	classe de définition	type de la méthode
m2	A	(A,A)
	B	(A,B)

Le choix de la signature de la méthode se portera sur la méthode la plus spécifique au sens du sous-typage (ici celle de signature `(A,B)`.

Cas d'ambiguïté

Soit une classe B avec 2 méthodes `m2(A,B)` et `m2(B,A)` :

```
1  A a1 = new A();
2  B b1 = new B();
3  b1.m2(a1,a1);
```

Sur les types on obtient B x (A, B) et B x (B, A)
Aucune de ces deux méthodes ne possède un type tel que (A,A) soit plus petit. il y a un clash à la compilation ! ! !

```
1  b1.m2(b1,b1)
```

Ici les deux méthodes ont un type tel que (B,B) soit plus petit, mais aucune des deux méthodes est plus petite que l'autre il y a un clash à la compilation ! ! !

Surcharge et liaison tardive

```
1  package pobj.cours4;
2  class A {
3      void m2(A a) {System.out.println("A1");}
4      void m2(A a1, A a2) {System.out.println("A2");}    }
5  class B extends A {
6      void m2(B  b) {System.out.println("B1");}
7      void m2(B b1, A a2) {System.out.println("B2");}    }
8  class C extends B {
9      void m2(C  c) {System.out.println("C1");}
10     void m2(A a1, A a2) {System.out.println("C2");}
11     void m2(B b1, B b2) {System.out.println("C3");}    }
12 class TestSurcharge {
13   public static void main(String [] args) {
14      A a1 = new A();
15      B b1 = new B();
16      C c1 = new C();
17      B b2 = c1;
18      A a2 = c1;
19      a1.m2(c1,c1);  // A2
20      b1.m2(c1,c1);  // B2
21      c1.m2(c1,c1);  // C3   b2, c1 et a2 partagent un objet
22      b2.m2(c1,c1);  // B2   mais ne sont pas de me^me type
23      a2.m2(a2,a2);  // C2
24 }}
```

Visiteur : extension des traitements

On cherche à séparer les données des traitements dans le but d'étendre les traitements. Pour cela les classes pour les expressions arithmétiques accepteront de recevoir un visiteur qui effectuera un traitement particulier (calcul de la valeur de l'expression, transformation en chaînes de caractères, ...).
On défini alors une classe abstraite visiteur qui pourra visiter chaque élément des expressions arithmétiques :

```
1 abstract class Visiteur {
2   public abstract void visite(CteV c);
3   public abstract void visite(AddV a);
4   public abstract void visite(MultV m);
5 }
```

la classe abstaite pour les expressions devient alors :

```
1 abstract class ExprArV {
2   public abstract void accepte(Visiteur v);
3 }
```

séparation effective des données et des traitements.

Expressions arithmétiques acceptant un visiteur

```
1  class CteV extends ExprArV {
2    private int val;
3    public CteV(int v) {val=v;}
4    public int getVal(){return val;}
5    public void accepte(Visiteur v){v.visite(this);}
6  }
7  abstract class OpBinV extends ExprArV {
8    protected ExprArV fg, fd;
9    ExprArV sous_expr_g(){return fg;}
10   ExprArV sous_expr_d(){return fd;}
11 }
12 class AddV extends OpBinV {
13   public AddV(ExprArV fg, ExprArV fd){
14     this.fg = fg; this.fd = fd;
15   }
16   public void accepte(Visiteur v){v.visite(this);}
17 }
18 class MultV extends OpBinV {
19   public MultV(ExprArV fg, ExprArV fd){
20     this.fg = fg; this.fd = fd;
21   }
22   public void accepte(Visiteur v){v.visite(this);}
23 }
```

Un visiteur de traitement de calcul

```
1  class VisiteurEval extends Visiteur  {
2    private int   res;
3    VisiteurEval(){res=0;}
4    public int  getRes(){return res;}
5
6    public void visite(CteV c){res=c.getVal();}
7
8    public void visite(AddV a){
9      int i;
10     a.sous_expr_g().accepte(this);
11     i=res;
12     a.sous_expr_d().accepte(this);
13     res=res + i;
14   }
15
16   public void visite(MultV m){
17     int i;
18     m.sous_expr_g().accepte(this);
19     i=res;
20     m.sous_expr_d().accepte(this);
21     res=res * i;
22   }
23 }
```

Test sur les expressions avec visiteur

```
1  package pobj.cours4;
2
3  class TestExprArV {
4    public static void main(String[] args) {
5      VisiteurEval v = new VisiteurEval();
6
7      ExprArV e1 = new CteV(10);
8      ExprArV e2 = new CteV(20);
9      ExprArV e3 = new MultV(e1,e2);
10     ExprArV e4 = new AddV(e1,e3);
11
12     e4.accepte(v);
13     System.out.println(e4 + " = " + v.getRes());
14   }
15 }
```

```
1  > java pobj/cours4/TestExprArV
2  pobj.cours4.AddV@1a0f73c1 = 210
```

Il manque un visiteur de conversion en chaîne de caractères pour afficher la formmule, mais le résultat est correct.

Redéfinition et co-variance du résultat

Depuis la version 1.5, il est possible dans le cadre d'une redéfinition de préciser (au sens du sous-typage) le type de retour. Le type du résultat de la méthode redéfinie est en relation de sous-typage avec celui défini dans la sur-classe. On parle de co-variance du type résultat.

- toujours pour éviter les *cast* inutiles

Exemple :

- dans `Point` :

```
1  protected Point clone () {
2    return new Point(getX(), getY());  }
```

- dans `PointColore` :

```
1  protected PointColore  clone () {
2    return new PointColore(getX(), getY());  }
```

- pas besoin de retourner un `Object` pour le transtyper ensuite en `Point` ou `PointColore`.

Surcharge et autoboxing

```
1  package pobj.cours4 ;
2
3  class TS2 {
4   public static int m1(int y){return 2 * y;}
5   public static Integer m1(Integer z){return 5 * z;}
6   public static int m2(int z){return 10 * z;}
7   public static void main(String[] a) {
8   int i = 1;
9   Integer k = new Integer(10);
10  System.out.println(m1(i) + "," + m1(k)); // 2,50
11  System.out.println(m1(m1(i)) +","+ m1(m1(k))); // 4,250
12  System.out.println(m2(i) +","+ m2(k)); // 10,100
13  System.out.println(m2(m2(i)) +","+ m2(m2(k))); // 100,1000
14   }
15 }
```