Université de Montpellier / Faculté Des Sciences

Notes de cours 3
Compléments sur la généricité en Java

Structures de données — HMIN 215

2 février 2020

1 Généricité bornée

Lorsque l'on écrit une classe générique, il arrive souvent que l'on attende du type passé en paramètre qu'il respecte un certain nombre de contraintes :

— les objets du type doivent fournir certains services (certaines opérations, plus rarement avoir certains attributs),

— les objets du type correspondent à une certaine abstraction.

Par exemple, si on désire munir la classe `Paire<A,B>` d'une méthode de saisie, une *contrainte* que l'on va poser est que les types A et B doivent disposer d'une méthode de saisie également.

La contrainte peut être représentée sous forme d'une classe, d'une classe abstraite ou mieux d'une interface :

```
public interface Saisissable
{
        void saisie(Scanner c);
}
```

Nous introduisons une nouvelle classe `PaireSaisissable` pour éviter des ambiguïtés avec la classe `Paire` du cours précédent. En fin de cours, nous évoquons les relations qu'elles peuvent avoir. Les paires saisissables sont des paires, munies d'une méthode permettant leur saisie. On déclare dans l'entête de `PaireSaisissable` que les types formels doivent spécialiser l'interface `Saisissable`. C'est ce que l'on appellera la "borne" ou la "contrainte". Les paires saisissables sont de plus elles-mêmes saisissables, ce qui vous explique que la nouvelle classe étend également l'interface `Saisissable` (ce n'est pas toujours le cas dans des définitions de bornes).

```
class PaireSaisissable<A extends Saisissable, B extends Saisissable>
                       implements Saisissable
{
  private A fst; private B snd;
  public PaireSaisissable(A f, B s){fst=f; snd=s;}
  public A getFst(){return fst;}
```

```
  public B getSnd(){return snd;}
  public void setFst(A a){fst=a;}
  public void setSnd(B b){snd=b;}
  public String toString(){return getFst()+''-''+getSnd();}
  public void saisie(Scanner c){
    System.out.print("Valeur first:"); fst.saisie(c);
    System.out.print("Valeur second:"); snd.saisie(c);}
}
```

Pour utiliser cette classe générique, nous aurons besoin d'un type concret qui réponde à la contrainte (ce n'est pas le cas de `String` ni de `Integer` par exemple).

```
public class StringSaisissable implements Saisissable
{
    private String s;
    public StringSaisissable(String s){this.s=s;}
    public void saisie(Scanner c)
        {s=c.next();}
    public String toString(){return s;}
}
```

Nous pouvons à présent écrire le programme correspondant.

```
Scanner c = new Scanner(System.in);
StringSaisissable s1 = new StringSaisissable("");
StringSaisissable s2 = new StringSaisissable("");
PaireSaisissable<StringSaisissable,StringSaisissable> mp =
      new PaireSaisissable<>(s1,s2);
mp.saisie(c);
```

2 Compléments sur les bornes

On peut avoir des contraintes multiples, il y a de nombreux exemples dans l'API de Java.

```
class Paire<A extends Saisissable & Serializable,
                    B extends Saisissable & Serializable>
   {.....}
```

On peut écrire des contraintes récursives, ce qui sera illustré et utilisé lors du TP.

```
public interface Comparable<T>
   {int compareTo(T o);}

public class orderedSet<A extends Comparable<A>>
   {....}
```

Enfin quand la borne n'a pas besoin d'être donnée explicitement, on peut utiliser le caractère joker (wildcard en anglais).

Il procure tout d'abord un super-type à toutes les instanciations. `Paire<?,?>` est un super-type de `Paire<Integer, String>`. Ce peut être utilisé pour le typage d'une variable, mais comme le type n'est pas complètement précisé, on ne peut ensuite pas faire tout ce que l'on veut, en particulier, on ne peut pas écrire d'expressions dont la vérification nécessite de connaître le paramètre de type.

```
Paire<?,?> p3 = new Paire<Integer, String>();
p3.setFst(12); // INCORRECT : setFst dépend du paramètre de type
System.out.println(p3); // CORRECT : car l'appel de toString
                        // ne demande pas de connaître le type
```

On peut utiliser les jokers pour simplifier l'écriture de certaines opérations. Par exemple, on peut remarquer que X et Y ne sont pas utilisés dans la vérification de l'écriture suivante :

```
public static<X,Y> void affiche(Paire<X,Y> p)
  {
   System.out.println(p.getFst()+" "+p.getSnd());
  }
```

On peut donc les faire disparaître en introduisant le caractère joker :

```
public static void affiche(Paire<?,?> p)
{
 System.out.println(p.getFst()+" "+p.getSnd());
}
```

3 Jeu de contraintes sur les paramètres des méthodes

Nous étudions à présent l'effet des paramètres de types choisis pour les méthodes, et en particulier, nous cherchons à vous montrer comment avoir des méthodes dont le champ d'application ne serait pas artificiellement réduit.

Examinons une méthode qui met dans le premier champ de la paire receveur du message (`this`) le premier élément d'une liste passée en paramètre (la liste sert de source de données) :

```
public class Paire<A,B>{
     public void prendListFst(List<A> c)
      {this.setFst(c.get(0));}
....
}
```

Regardons son utilisation dans le contexte d'un programme :

```
Paire<Object,String> p6 = new Paire<>();
List<Object> lo = new LinkedList<>();
List<Integer> li = new LinkedList<>();
lo.add(new Integer(6));
li.add(new Integer(6));
p6.prendListFst(lo);  // correct
p6.prendListFst(li);   // ne fonctionne pas
```

Analysons la raison pour laquelle la dernière instruction ne fonctionne pas. `li` est une `List<Integer>`, ce n'est pas une `List<Object>` (cf cours précédent) et elle ne peut pas être passée en paramètre à `prendListFst(List<Object> c)`. Pourtant, quand on examine ce que réalise la méthode, il n'y a pas d'inconvénient à mettre un `Integer` dans une liste d'`Object`. Pour que cela fonctionne, il faut être capable de dire à la fonction `prendListFst` qu'elle peut admettre comme paramètre une liste d'objets de type A ou d'un sous-type de A.

On peut la réécrire ainsi :

```
public <X extends A> void prendListFst(List<X> c)
    {this.setFst(c.get(0));}
```

Mais comme le paramètre de type X ne sert à rien pour le compilateur (deuxième possibilité), on peut le faire disparaître au profit du joker :

```
// on rappelle que A est le type formel du premier champ
public void prendListFst(List<? extends A> c)
    {this.setFst(c.get(0));}
```

Le problème peut exister en sens inverse. Par exemple, si on s'intéresse à une méthode qui écrit le premier membre d'une paire dans une collection (la collection sert de puits de données) :

```
public class Paire<A,B>{
public void copieFstColl(Collection<A> c)
    {c.add(this.getFst());}
 ....
 }
```

Elle est trop stricte comme le montre le programme suivant :

```
 Paire<Integer,Integer> p2 = new Paire<>(9,10);
 Collection<Object> co = new LinkedList<>();
 p2.copieFstColl(co); // INCORRECT
```

La dernière instruction ne pourra pas compiler et pourtant mettre un `Integer` dans une collection d'`Object` devrait être possible. On écrit une nouvelle version qui va le permettre :

```
public void copieFstColl(Collection<? super A> c)
    {c.add(this.getFst());}

// dans un main ...
 Paire<Integer,Integer> p2 = new Paire<Integer,Integer>(9,10);
 Collection<Object> co = new LinkedList<Object>();
 p2.copieFstColl(co);  // A present CORRECT
```

4 Le principe de l'effacement de type et ses conséquences

En Java, la généricité est mise en œuvre avec la technique de l'effacement de type. Cela concerne l'étape de compilation (génération du fichier de bytecode) lors de laquelle :

— toutes les informations de type placées entre chevrons sont effacées
`class Paire {...}`
— Les variables de types restantes sont remplacées par la borne supérieure (`Object` en l'absence de contraintes)
`class Paire{private Object fst; private Object snd;..}`
— Insertion de typecast si nécessaire (quand le code résultant n'est pas correctement typé)
```
Paire p = new Paire(9,''plus grand chiffre'');
Integer i=(Integer)p.getFst();
```

Cette implémentation a plusieurs conséquences :
— A l'exécution, il n'existe en fait qu'une classe qui est partagée par toutes les invocations
`p2.getClass()==p5.getClass()` même si p2 est une `Paire<String,Integer>` et p5 une `Paire<Double,Etudiant>`.
— Les variables de type paramétrant une classe ne portent pas sur les méthodes et variables statiques
— Un attribut statique (ex. le nombre d'entrées dans toutes les files d'attente de tous les types possibles du cours précédent) n'existe qu'en un seul exemplaire
— Généralement, on ne peut pas faire de typecast comme `(Paire<Integer,Integer>)p`

Le type brut (*raw type*) est le type paramétré sans ses paramètres, par exemple `ArrayList l;` ou `Paire p`. Il assure l'interopérabilité avec le code ancien (Java 1.4 et versions antérieures). mais il faut éviter d'utiliser des types bruts car le compilateur ne fait pratiquement pas de vérification (il vous l'indique d'ailleurs par un warning).

5 Héritage, généricité, invocation

Comme expérimenté dans les exercices du précédent cours avec des variations sur les paires, on peut combiner de diverses manières les différents outils que sont l'héritage, la généricité et l'invocation :
— Classe générique dérivée d'une classe non générique
```
class Graphe{}
class GrapheEtiquete<TypeEtiq> extends Graphe{}
```
— Classe générique dérivée d'une classe générique
`class TableHash<TK,TV> extends Dictionnaire<TK,TV>{}`
— Classe dérivée d'une instanciation d'une classe générique (on peut même dériver d'une instanciation partielle)
```
class Agenda extends Dictionnaire<Date,String>{}
class Agenda<Evt> extends Dictionnaire<Date,Evt>{}
```

Nous pourrions redéfinir la classe `PaireSaisissable` comme une sous-classe de `Paire`. Les détails vous sont laissés à titre d'exercice.