Université de Montpellier / Faculté Des Sciences

Notes de cours
Introduction à la généricité en Java

Structures de données — HMIN 215

24 janvier 2020

1 Paramétrage d'une classe ou d'une interface

Examinons une forme ultra-simplifiée de l'interface de la liste. On y voit apparaître le type `Object` partout où l'on souhaite mentionner le type des objets stockés dans la liste : comme paramètre de l'opération `add` par exemple.

```
public interface List
{
    void add(Object element);
    Object get(int index);
    int size();
}
```

Ce n'est pas idéal car :
- si on veut faire une liste ne contenant que des rectangles, par exemple, il faudra contrôler ce que l'on met à chaque fois que l'on fait un `add`. Ce sera réalisé par exemple avec une expression comme `if (element instanceof Irectangle)`.
- si on récupère un `element` par un `get`, on obtient un objet de type statique `Object`, si on veut appliquer une méthode comme le calcul du périmètre, il faudra faire un "typecast". Ce sera réalisé par exemple avec une expression comme : `((Irectangle) element).perimetre()`.

Pour rendre générique l'interface et créer des listes d'objets d'un même type, on va :
- Indiquer un type formel (`E`) derrière le nom de l'interface.
- Remplacer `Object` par `E` partout dans l'interface.

C'est ce qui est fait dans l'API des collections de Java.

```
public interface List<E>
{
    void add(E element);
    E  get(int index);
    int size();
}
```

`<E>` introduit ce que nous appellerons le **paramètre de généricité**. On déclare que l'on décrit une liste d'éléments de type `E`, ce type étant précisé plus tard. `E` est utilisé ensuite dans l'interface (ce sera la même chose dans une classe) comme si c'était un type ordinaire, connu, même si c'est en réalité un paramètre formel.

Pour créer des listes, il faut ensuite procéder à ce que l'on appelle, suivant les langages de programmation, une *instanciation* (par analogie à la création d'objets) ou une *invocation* (par analogie à l'invocation d'une méthode dans laquelle on lie les paramètres formels avec des paramètres réels). On peut ainsi déclarer une variable de type liste de rectangles comme une invocation de l'interface générique `List<E>` (première instruction ci-dessous), puis créer une liste par invocation de la classe `ArrayList<E>` (deuxième instruction ci-dessous).

```
List<Irectangle> listeRectangles; // invocation de List<E>
listeRectangles = new ArrayList<Irectangle>(); // invocation de ArrayList<E>
// ou à partir de Java 1.7
listeRectangles = new ArrayList<>(); // invocation de ArrayList<E>
```

2 Classes génériques paramétrées par plusieurs paramètres

Nous continuons l'exploration de la généricité paramétrique qui autorise la définition d'algorithmes et de types complexes (classes, interfaces) paramétrés par des types.

On peut avoir plusieurs paramètres de généricité. Nous travaillons par la suite avec la classe `Paire` qui représente des couples d'éléments. Les éléments d'une paire peuvent avoir chacun un type différent. Les paires sont des objets de base manipulés par certaines structures de données telles que les dictionnaires associatifs. Il existe donc un équivalent de cette classe dans l'API des collections Java (`Map.Entry<K,V>`).

```
public class Paire<A,B>
{
  private A fst;
  private B snd;
  public Paire(){}
  public Paire(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();}
}
```

Nous savons déjà créer une liste d'étudiants avec une `ArrayList` paramétrée par le type `Etudiant`. Pour créer des paires, nous appliquons le même principe. Remarquez :

- qu'il n'y a pas de nécessité d'écrire une condition vérifiant ce que l'on met dans la paire, si les types des paramètres réels passés au constructeur ne sont pas corrects, la compilation échouera.
- qu'il n'y a pas de nécessité d'utiliser de `cast` (ou coercition) lorsque l'on récupère le premier membre (`fst`) ou le second membre (`snd`) d'une paire.

```
Paire<Integer,String> p1 = new Paire<Integer,String>(9,"plus grand chiffre");
Integer i=p1.getFst();
```

```
String s=p1.getSnd();
System.out.println(p1);

Paire<String,Etudiant> pe1
       = new Paire<String, Etudiant>("Mohamed", new Etudiant());
Paire<String,Etudiant> pe2
       = new Paire<String, Etudiant>("Guillaume", new Etudiant());

// Exercice : écrivez des instructions similaires avec les types
// qui sont leurs équivalents en Java
// Map.Entry et AbstractMap.SimpleEntry
// en consultant la documentation Java 8 ou JAva 9 en ligne
```

Vous aurez peut-être remarqué que nous n'avons pas utilisé le type primitif `int`, mais la classe `Integer`. C'est dû à une limitation de la généricité paramétrique propre à Java : *On ne peut pas utiliser de type primitif comme paramètre !* Ce problème n'est pas très grave, il est modéré par un autre mécanisme de Java, *l'autoboxing*, qui convertit de manière transparente les valeurs des types primitifs en instances.

3 Le paramétrage de méthodes et la portée des paramètres de généricité

Une règle importante à noter : les paramètres de généricité portent sur les attributs et les méthodes d'instance. Ils ne portent pas sur les attributs et les méthodes de classe (`static`).

3.1 Paramétrage d'une méthode de classe (`static`)

Une méthode `static` va donc avoir elle-même ses propres paramètres de généricité. Ci-dessous on voit ainsi le paramétrage de la méthode de classe `copieFstTab` dont le rôle est de copier la première composante d'une paire dans un tableau à un certain indice.

```
class Paire<A,B>
{
    .....
    public static<X,Y> void copieFstTab
                                (Paire<X,Y> p, X[] tableau, int i)
    {tableau[i]=p.getFst();}
    // exercice : code à compléter pour vérifier que l'indice est correct
}
```

3.2 Compléments de paramétrage d'une méthode d'instance

Par ailleurs, on peut paramétrer les méthodes d'instance avec des types différents de ceux de la classe générique concernée. Par exemple, la méthode d'instance `memeFst` compare les deux premières composantes de deux paires dont la deuxième composante n'est pas forcément de même type. Elle demande d'introduire un paramètre de généricité pour le type de la deuxième composante.

```
class Paire<A,B>
{
    .....
    public <C> boolean memeFst(Paire<A,C> p)
    {return p.getFst().equals(this.getFst());}
}
```

Voici une utilisation de ces méthodes. Dans un premier temps, on copie le premier membre de la paire `p5` dans un tableau d'entiers à l'indice 0. Puis on compare les premiers membres des paires `p5` et `p2`.

```
Paire<Integer,String> p5 = new Paire<Integer,String>(9,
                                        "plus grand chiffre");
Integer[] tab=new Integer[2];
Paire.copieFstTab(p5,tab,0);
// l'appel est sur la classe Paire et non sur une instance
// car copieFstTab est static

Paire<Integer,Integer> p2 = new Paire<Integer,Integer>(9,10);
System.out.println(p5.memeFst(p2));
// memeFst n'est pas static : elle est appelée sur une instance
```

4 Généricité et héritage

4.1 Sous-typage des paramètres de généricité

Quoique `String` soit un sous-type de `Object`, et que du point de vue de la spécialisation "naturelle" une liste de `String` soit une sorte de liste d'`Object`, le point de vue du sous-typage est différent. En pratique, on ne peut pas écrire qu'une liste de `String` est une liste d'`Object`, comme nous le voyons en testant :

```
ArrayList<Object> a = new ArrayList<Object>();
a = new ArrayList<String>();
```

La raison en est que certaines opérations admises sur une liste d'`Object`, comme `add(new Integer())`, ne peuvent effectivement pas s'appliquer à une liste de `String` (sinon, on ajouterait un entier à une liste de chaînes de caractères). Les opérations qui ne modifient pas la liste ne sont pas en cause, cela provient de celles qui la modifient.

4.2 Héritage et invocation

Différentes combinaisons sont possibles entre héritage, généricité et invocation d'une classe générique. Nous allons les découvrir au travers d'exercices.