Initiation au classes et aux méthodes « génériques » :

De Java 1.4 à Java 1.5

UNIVERSITÉ CAEN NORMANDIE

1. Les classes génériques, ou classes paramétrées

1.1 Problème

Jusqu'à la version 1.4 de Java, le moyen de rendre des classes très génériques était de leur faire manipuler le type le plus général qui soit, à savoir Object, puisque la classe correspondante est au sommet de la hiérarchie d'héritage.
Ainsi, par exemple, la classe ArrayList, qui permet de collectionner des références sur des Objects, est utilisable avec n'importe quelle instance.
Néanmoins, cette façon de procéder a certains inconvénients. En particulier, lorsque les objets que l'on manipule via une telle classe sont d'un type plus spécifique que le type Object, il est impossible de s'en assurer au moment de la compilation. Par exemple, si l'on crée la classe CollectionFormes qui possède une ArrayList (non paramétrée) référencée par formes, dans laquelle on stocke des instances de type Forme (ou, bien sûr, par polymorphisme, d'un de ses sous-types), même si notre méthode d'ajout de formes, grâce à sa signature, veille à ce que l'on ne puisse y ajouter que des instances de Forme :

```
public void ajoutForme(Forme f)
{
     formes.add(f) ;
}
```

rien ne permet de s'assurer, lors de la compilation, que le programme soit exempt d'une erreur telle que :

```
formes.add(new Integer(3)) ;
```

Réciproquement, lorsque l'on va utiliser cette ArrayList pour, par exemple, appliquer une certaine méthode à tous les objets qu'il référence, comme la translation, on va devoir effectuer un transtypage (ou cast) à chacune des références que l'on obtient par la méthode get() :

```
Forme f = (Forme) formes.get(i) ;  // danger à l'exécution
f.translation(x,y) ;
```

Pour résumer, cette façon de concevoir les choses permet certes d'être générique, mais au prix d'une prise de risques à l'exécution (la compilation ne pouvant prévoir les erreurs possibles : rien n'indique qu'un ArrayList ne contient par exemple que des formes, puisque sa méthode add prend en paramètre un Object), et d'une écriture assez lourde (casts rendus souvent nécessaires).

1.2 Solution

Java 1.5 (et versions suivantes) répond élégamment à ce problème en proposant la notion de « classes génériques ». En fait de classe dites « génériques », nous préférerons parler de classes paramétrables. Il s'agit de donner aux classes, lors de leur conception, la possibilité de les paramétrer avec un ou plusieurs types, et d'utiliser ce ou ces paramètres de types à divers endroits du code de la classe (notamment en tant que types de retour ou de paramètres de méthodes).
Voyons l'exemple de ArrayList tel qu'apparaissant dans l'API 1.5 :
Elle est paramétrée par le type formel E, qui est comme une variable représentant un certain type :

```
public class ArrayList<E>
```

Sa méthode add a pour paramètre une référence de ce type E

```
public boolean add(E o)
```

Et sa méthode get renvoie une référence de type E :

```
public E get(int i)
```

1.3. Utilisation

De la sorte, on peut maintenant créer un ArrayList de Formes, tout simplement en utilisant le type Forme comme paramètre lors de la création de la variable et de l'instance :

```
ArraytList<Forme> formes = new ArrayList<Forme>() ;
```

Ainsi, on peut directement écrire :

```
Forme f = formes.get(i) ;
```

car le type de retour de get n'est plus forcément Object, mais ce qui a été passé en paramètre (ce que l'on reçoit dans le paramètre formel E dans la définition de la classe). En l'occurrence, dans notre exemple, il s'agit de Forme.

Par ailleurs, le compilateur est désormais capable de repérer des erreurs qui n'étaient pas repérables dans Java 1.4 :

```
formes.add(new Rectangle(p1,p2)) ;
formes.add(new Integer(5)) ; // Erreur à la compilation
```

1.4 Exemple : les listes chaînées

Nous pouvons à présent reprendre nos classes de listes chaînées (vues en TP cette semaine) afin de les rendre paramétrables :

```
public class Cellule<E>
{
     private Cellule<E> precedent, suivant;
     private E valeur;

(...)

public E getValeur()
     {
     return valeur;
     }

public void setValeur(E valeur)
     {
     this.valeur=valeur;
     }

}

public class ListChainee<E>
{
     private Cellule<E> premierElement;
     private Cellule<E> dernierElement;

(...)

}
```

Classes génériques ou paramétrables ?

Maintenant que le mécanisme de base est vu, on peut se poser la question de l'appellation de telles

classes. On parle de classes génériques dans la mesure où à partir d'une même écriture générique comme par exemple ArrayList<E>, on va pouvoir créer des ArrayList de n'importe quoi, comme par exemple des ArrayList<Forme>, des ArrayList<String>, etc. On peut alors se dire que ArrayList<E> est une classe « générique » permettant de générer autant de classes que l'on souhaite.
En fait, il n'en est rien. Aussi bien à la compilation qu'à l'exécution, Java ne va considérer qu'une seule classe ArrayList<E>, et ne générera donc bien qu'un fichier .class correspondant. Cette classe est cependant paramétrable, et lors de son instanciation, son instance se voit bien passer un paramètre particulier. Il est donc plus juste de considérer que `new ArrayList<Forme>()` est une instance de ArrayList<E> paramétrée par Forme plutôt que de considérer qu'il s'agit d'une instance d'une nouvelle classe spécifique ArrayList<Forme>.
A l'opposé, la notion de templates en C++ amène à la création d'autant de classes que d'utilisations avec des paramètres différents.

1.5. Classes à paramètres multiples

Voir par exemple l'API 1.5 de :

```
public Class HashMap<K,V>
```

Cette classe permet de créer des Map ayant des clés de type K, et des valeurs de type V :

```
HashMap map=new HashMap<String, Forme> // crée un maping avec des clés sous forme de Strings, vers des valeurs étant des Formes.
```

2. Polymorphisme, wildcards

2.1. Polymorphisme : gare à l'intuition

Posons-nous la question suivante :
une ArrayList<Forme> est-elle une ArrayList<Object> ? Ou, de façon moins formelle, « une ArrayList de formes est-elle une ArrayList d'objets » ?
L'intuition peut nous amener à répondre oui : une Forme étant un Objet, on doit pouvoir mettre dans une ArrayList d'objets une ArrayList de formes. Mais l'intuition est trompeuse, comme peut nous en convaincre le code suivant :

```
ArrayList<Forme> listeFormes = new ArrayList<Forme>() ;
listeFormes.add(new Rectangle(p1,p2)) ;
ArrayList<Object> listeObjects = listeFormes ; // Erreur à la compilation
listeObjects.add("bonjour"); // Attention, on pourrait alors ajouter un String
listeObjects.get(1).translation(1,2) ; // On essaierait de
```

`translater un String…`

Conclusion, si **F extends E** :
- par polymorphisme, on peut écrire E e = new F() ;
- mais **on ne peut pas écrire ~~A<E> ae = new A<F>()~~** ;

2.2. Wildcards ‘ ?’ (nota : les exemples de cette partie sont empruntés à la documentation java)

```
public void printCollection(Collection<Object> c)
{
     for (Object e : c)
     {
     System.out.println(e);
     }
}
```

La méthode précédente permet-elle d’afficher une Collection<String> ?

La solution : la « wildcard » ‘ ?’

```
public void printCollection(Collection<?> c)
{
for (Object e : c)
     {
     System.out.println(e);
     }
}
```

On accepte ici une Collection de n’importe quoi. Les éléments que l’on récupère dans e peuvent donc être des éléments d’un type quelconque. Le compilateur peut donc nous les délivrer en tant qu’Object sans qu’il n’y ait jamais de problème. Rappelons que next(), utilisée de façon cachée dans l’écriture for each ci-dessus, a pour déclaration paramétrée `public E next()`, qui peut donc, selon le paramètre, être `public String next()`, `public Forme next()`, etc. Donc, dans tous les cas, quelque chose qui peut être vu comme un Object. Le compilateur s’en acquitte donc bien.

Attention cependant : si on peut lire de façon sécurisée les éléments de la collection en tant qu’Objects, on ne peut par contre pas y ajouter quoi que ce soit :

```
Collection<?> c = new ArrayList<String>();
c.add(new Object()); // Erreur à la compilation
```

Dans le code précédent, c peut être une référence sur une collection de n’importe quoi. À l’exécution, il va d’ailleurs s’agir dans notre exemple d’une collection de Strings. Lorsque l’on utilise la méthode add, cette dernière prend en paramètre une référence du type du paramètre de la collection paramétrée, à savoir, dans notre exemple, String : `public void add(E o)` s’entend ici `public void add(String o)`, ce qui rend bien notre seconde ligne dangereuse (elle mènerait à

une erreur de type à l'exécution). Naturellement, Java 1.5 va donc refuser cette ligne à la compilation.

2.3. Wildcard bornées : ' ? extends UneClasse'

Une wildcard peut avoir une borne supérieure, c'est-à-dire une classe au-delà de laquelle on ne peut pas remonter. Ainsi, ' ? extends Forme' signifie « n'importe quelle classe étant une Forme, donc Forme, Rectangle, Carré, Polygone, etc ». Attention, extends est à considérer comme « étant un », et pas nécessairement « étant un sous … », car Forme est bien compatible avec ' ? extends Forme'.

Alors que la méthode suivante n'est utilisable qu'avec des Collection<Forme>, et avec rien d'autre :

```
public void affiche(Collection<Forme> collectionFormes)
for (Forme f : collectionFormes)
     f.affiche() ; // méthode de la classe Forme
```

la méthode suivante est utilisable aussi bien avec des Collection<Forme> qu'avec des Collection<Rectangle>, des Collection<Polygone>, etc. :

```
public void affiche(Collection<? extends Forme> collectionFormes)
for (Forme f : collectionFormes)
     f.affiche() ; // méthode de la classe Forme
```

Attention cependant, un accès en écriture est ici encore interdit :

```
public void ajoutRectangle(Collection<? extends Forme> formes)
     {
     formes.add(new Rectangle(new Point(1,2), 2, 3) // Erreur à la
     compilation
     }
```

En effet, la méthode add est ici à considérer comme public void add(<? extends Forme> o). Donc, cela peut être, selon le paramétrage de la collection reçue, public void add(Forme o), ou public void add(Rectangle o), mais aussi public void add(Cercle o). Or un Rectangle n'est pas un cercle. C'est pourquoi le compilateur ne tolère pas cette instruction : elle serait dangereuse, et ne fonctionnerait à l'exécution que dans les cas où le paramètre de la collection serait Rectangle, ou l'un de ses super-types.

3. Méthodes « génériques » (ou paramétrées)

Considérons l'exemple suivant, issu de la documentation Java. Nous souhaitons écrire une méthode statique permettant d'insérer le contenu d'un tableau dans une collection. Nous pouvons faire la tentative suivante :

```
public static void fromArrayToCollection(Object[] a, Collection<?>
c)
{
     for (Object o : a)
     {
     c.add(o); // erreur à la compilation
     }
}
```

Attention, comme nous l'avons déjà vu, ce code est refusé à la compilation. Souvenez-vous que l'usage de la méthode c.add(o) peut par exemple essayer d'ajouter un String à une collection de Formes.
Nous allons utiliser un paramètre T permettant de contraindre le type du tableau et celui de la collection :

```
public static <T> void fromArrayListToCollection(ArrayList<T> a,
Collection<T> c)
{
     for (T o : a)
     {
     c.add(o);
     }
}
```

Ce code est licite, et peut être utilisé en lui passant un tableau d'un type T et une collection du même type T.

On peut être plus général en proposant d'ajouter à une collection de U une ArrayList d'un sous-type de U, et non nécessairement directement un U. Pour cela, nous allons procéder à un double paramétrage via T et U, et en contraignant l'un par rapport à l'autre :

```
public static <T extends U, U> void
fromArrayToCollection(ArrayList<T> a, Collection<U> c)
     {
          for (T o : a)
          {
               c.add(o);
          }
     }
```

Remarquons enfin que l'appel à la méthode n'a pas besoin d'être paramétré, le compilateur inférant

automatiquement le type T adéquat (ou générant une erreur si cela s'avère impossible, comme dans le cas d'un ArrayList de Strings et une collection de Formes).

4. Complément : les boucles « for each »

Voyons à présent un apport (depuis Java 1.5) concernant les boucles *for* appliquées à des collections ou à des tableaux, qui nous sera utile par la suite.

Lorsque l'on itère sur une collection, on peut bien sûr utiliser un Iterator :

```
Collection<String> maCollection=new ArrayList<String>();
for (Iterator<String> it=maCollection.iterator(); it.hasNext(); )
     {
     System.out.println("affichage de l'élément : "+it.next());
     }
```

Inconvénient : long à écrire (*it* apparait trois fois dans le code), source d'erreurs (notamment par l'emploi de plusieurs *next()* dans le corps de l'itération si on a besoin d'accéder plusieurs fois à la référence.

Java 1.5 (ou supérieur) est capable d'interpréter la syntaxe suivante :

```
for (String s : maCollection)
{
System.out.println("affichage de l'élément : "+s);
}
```

Il faut lire ce for comme "pour tout s dans maCollection", ou "for each s in maCollection".

C'est plus concis, plus lisible, est moins source d'erreurs (on peut utiliser s autant de fois que l'on veut dans la boucle, à la difference de next() qui imposerait de passer par une variable intermédiaire). En fait, le compilateur traduit cette écriture concise en l'écriture utilisant l'itérateur, mais cet itérateur n'apparaît donc plus explicitement dans le code source. Seul inconvenient : l'itérateur étant masqué, on n'a plus accès à sa méthode remove(). Lorsque l'on a besoin de cette dernière, on utilisera l'écriture classique basée explicitement sur l'itérateur.

De même, avec un tableau, "for each" peut s'utiliser ainsi :

```
// somme des entiers d'un tableau
public int somme(int[] a)
     {
```

```
int resultat = 0;
for (int i : a)
   resultat += i;
return resultat;
}
```