La généricité

13 avril 2016

1 Un exemple connu

La classe ArrayList permet de représenter un tableau sous forme d'un objet et de bénéficier ainsi de nombreuses méthodes telles que l'ajout ou le retrait d'éléments. Lorsqu'on veut utiliser cette classe, il faut préciser le type des éléments que l'on place dans le tableau.

Par exemple, si l'on veut un ArrayList contenant des String, il faudra déclarer la variable et créer l'objet ArrayList comme suit.

```
ArrayList<String> var;
var = new ArrayList<String>();
```

Si l'on va voir la documentation de la classe ArrayList, on voit entre autres les choses suivantes :

- `Class ArrayList<E>`
- `boolean add(E e)`

Dans le nom de la classe apparaît un E qui ne correspond pas à un type java, et ce E apparaît également comme type de paramètres ou de résultat de méthodes de la classe. Ce E est un nom qui peut être remplacé par n'importe quel type lorsqu'on crée effectivement un objet. Lorsque cette classe est définie (et c'est ce qui est décrit dans la documentation), le E n'a pas de valeur précise. C'est à la création de l'objet, lors de l'instanciation de la classe, que ce E est remplacé par le nom d'une classe bien précise (String dans notre cas).

On appelle *généricité* cette possibilité de retarder la définition d'un type utilisé dans une classe.

2 Ecriture d'une classe générique

Si nous voulons écrire nous-même une classe générique, il faut reprendre à peu près la même chose que ce que l'on voit dans la documentation de la classe ArrayList : il faut donner des noms provisoires aux types dans le nom de la classe, avec les signes inférieur et supérieur. Après cela, les nom provisoires peuvent être utilisés pour déclarer des variables, des paramètres, des types de résultat. Et là, il ne faut pas mettre d'inférieur et supérieur.

Nous allons prendre l'exemple d'une mini-classe ArrayList faite maison, c'est à dire une classe encapsulant un tableau et offrant quelques méthodes : add, set, get, remove et size.

```
public class MyArrayList<UnType>{
  private UnType[] tab;
  private int nb=0;
  MyArrayList(){
    // On voudrait écrire ici tab=new UnType[100]
```

```
    // mais ça n'est pas permis par le langage
    tab=(UnType[]) new Object[100];
  }
  public void add(UnType e){
    tab[nb]=e;
    nb++;
  }
  public UnType set(int i, UnType e){
    if (i<0 || i>=nb)
      throw new IndexOutOfBoundsException();
    UnType res = tab[i];
    tab[i]=e;
    return res;
  }
  public UnType get(int i){
    return tab[i];
  }
  public UnType remove(int i){
    if (i<0 || i>=nb)
      throw new IndexOutOfBoundsException();
    UnType res = tab[i];
    for(int j=i;j<nb-1; j++)
      tab[j]=tab[j+1];
    nb--;
    return res;
  }
  public int size(){
    return nb;
  }
}
```

Ensuite, on peut utiliser la classe générique en remplaçant, à chaque objet créé, UnType par le nom d'une classe Java.

```
public class Utilise{
  public static void main(String[] args){
    MyArrayList<String> mal;
    mal = new MyArrayList<String>();
    MyArrayList<Chose> mal2=new MyArrayList<Chose>();
    mal.add("un");
    mal.add("deux");
    System.out.println(mal.size());
    System.out.println(mal.remove(1));
    Chose cpt = new Chose("riri");
    mal2.add(cpt);
    mal2.add(new Chose("fifi"));
    System.out.println(mal2.get(0).nom);
  }
```

```
}
class Chose{
  String nom;
  Chose(String n){
    nom=n;
  }
  public String getNom(){
    return nom;
  }
}
```

3 Note à propos des types génériques

Lors de l'instanciation d'une classe générique, on peut remplacer les noms de types par n'importe quel type référence, c'est à dire un nom de classe, un type tableau, un nom d'interface, mais pas un type primitif (int, double, boolean, char).

Si l'on essaye par exemple cette instanciation :

```
MyArrayList<int> mal4 = new MyArrayList<int>();
```

le compilateur refuse le programme avec l'erreur :

```
Utilise.java:16: error: unexpected type
    MyArrayList<int> mal4 = new MyArrayList<int>();
                ^
  required: reference
  found:    int
```

Il existe quatre classes prédéfinies permettant de représenter avec des objets des entiers, nombres à virgule, booléens et caractères : ce sont les classes Integer, Double, Boolean et Character.

On peut par exemple écrire quelque chose comme :

```
MyArrayList<Integer> mal5 = new MyArrayList<Integer>();
mal5.add(7);
mal5.add(12);
int x = mal5.get(0);
System.out.println(x);
```

4 Généricité et interface

Nous avons vu l'utilisation de généricité pour les classes. La généricité est aussi utilisée avec les interfaces d'une façon un peu différente. En effet, les interfaces ne sont pas instanciées, ce n'est donc pas à la création d'objet que les types inconnus sont remplacés par des types prévis. C'est lors de *l'implémentation de l'interface*.

Dans la plupart des cas, on utilise le type inconnu pour représenter le type de la classe qui implémente l'interface, car on a besoin d'utiliser ce type pour les paramètres et/ou résultat des méthodes de l'interface.

Voyons l'exemple de l'interface prédéfinie `Comparable` de Java. Cette interface peut s'écrire comme suit :

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

Cette interface décrit la propriété d'avoir un ordre sur un type. La méthode `compareTo` doit comparer l'objet sur laquelle elle est appelée avec l'objet passée en paramètre et l'on veut généralement comparer des objets de même type. Un appel à la méthode s'écrira comme suit : `objet1.compareTo(objet 2)`.

Cette interface est implémentée par la classe prédéfinie String (la classe des chaînes de caractère) avec comme valeur de T le type String lui-même. Dans la documentation de `String` on voit :

```
public class String implements Comparable<String>...
```

On voit quelque chose d'analogue avec d'autres classes prédéfinies de Java :

- `public class Integer implements Comparable<Integer>`
- `public class Date implements Comparable<Date>`
- ...

Le plus souvent, la généricité est ainsi employée dans les interfaces pour donner un nom au type de la classe qui l'implémente.

5 Exemple d'interface générique

Nous proposons l'interface `Affiliable` qui propose de représenter des notions de parenté entre objets du même type. Les méthodes fournies permettent de déclarer un lien de filiation et de tester les liens entre deux objets. On pourra utiliser cette interface avec des êtres humains, des animaux, des programmes objets (héritage), mais on ne pourra pas créer de lien de filiation entre un homme et un chien ou entre une chouette et un programme : la filiation se fera uniquement entre objets de la même classe et pour cela, on a besoin de la généricité.

```
public interface Filiation<T>{
  T[] getFils();
  void declareFils(T fils);
  boolean estFils(T t);
  boolean estPere(T t);
  boolean estAncetre(T t);
}
import java.util.ArrayList;
public class Humain implements Filiation<Humain>{
  String nom;
  Humain pere;
  ArrayList<Humain> fils;
  public Humain(String n){
    nom = n;
    fils=new ArrayList<Humain>();
  }
  public Humain[] getFils(){
    return fils.toArray(new Humain[0]);
  }
  public void declareFils(Humain t){
```

```
    if (t.pere != null)
      t.pere.fils.remove(fils);
    t.pere = this;
    this.fils.add(t);
  }
  public boolean estFils(Humain t){
    return t.pere.equals(this);
  }
  public boolean estPere(Humain t){
    return this.pere.equals(t);
  }
  public boolean estAncetre(Humain t){
    Humain ancetre = t.pere;
    while (ancetre!= null && ancetre != this)
      ancetre = ancetre.pere;
    return (ancetre != null);
  }
}
```