

Généricité en Java

Virginia Aponte

CNAM-Paris

10 septembre 2016

1. Définitions, exemples

Paramétrisation d'un code

Mécanisme puissant de la programmation qui consiste à généraliser un code, de manière à lui passer en paramètre les variables dont il dépend.

- le code devient **réutilisable** sur toute valeur des paramètres.

Exemple : afficher un tableau d'entiers

- boucle affichant un tableau t particulier \Rightarrow code figé pour t, non réutilisable.
- méthode paramétrée par n'importe quel tableau x \Rightarrow code réutilisable pour tout tableau concret, de n'importe quelle taille.

Deux sortes de paramétrisation

- **Par données** : une méthode est paramétrable par
 - **variables désignant des valeurs quelconques** (entiers, booleans, une instace d'un objet)
 - \Rightarrow code réutilisable sur n'importe quelle donnée (du bon type).
- **Par types** : une classe ou méthode est paramétrable par
 - **variables désignant des types quelconques**
 - \Rightarrow réutilisable sur n'importe quel type concret.

Définition de généricité

Généricité

Paramétrisation d'un morceau de code (classes, méthodes), par un **type**.

Exemple :

- classe `ArrayList<T>` : son code paramétré par la *variable de type* `T`
- `<T>` : signifie «n'importe quel type non primitif»
- méthodes de la classe : décrites pour n'importe quel `T`
- ré-utilisable pour n'importe quel `T` : `new ArrayList<Compte>()` .
Ici, `T=Compte`.

Exemple généricité : classe ArrayList<E>

```
public class ArrayList<E>
    implements List<E>, Collection<E>, Iterable<E> .
    public boolean add(<E> el) {...}
    public E get(int i){...}
    ....
}
```

- E : type (quelconque) des éléments de la liste ;
- implante plusieurs interfaces génériques ;
- les méthodes ont des types génériques :
 - `boolean add(<E> el)` ajoute un élément de type E dans la liste ;

Exemple généricité (2) : utilisation

Un `ArrayList<T>` avec `T=Compte` :

```
ArrayList<Compte> comptesBNP = new ArrayList<Compte>();  
Compte c1 = new Compte("Lucie", 150.0);  
  
comptesBNP.add(c1);    // dans liste comptes banque
```

- `comptesBNP` : liste de comptes de la banque

Employée de manière intensive dans la bibliothèque `Collections` :

- Types génériques :
 - interfaces génériques (`List<T>`, `Set<T>`, ...)
 - classes (implantations) génériques : (`LinkedList<T>`, `ArrayList<T>`, ...)
- méthodes génériques
- nombreuses implantations génériques **prédéfinies** : (`LinkedList<T>`, `ArrayList<T>`, ...)

2. Pourquoi la généricité ?

Intérêt de la généricité

- Le typage est plus précis \Rightarrow plus d'erreurs sont détectées à la compilation :
- sans généricité : certaines erreurs ne sont détectables que lors de l'exécution ;
- sans généricité : le code est souvent plus complexe.

Une collection d'objets contient le plus souvent un seul type d'objets et éventuellement ses dérivés : `Compte`, `CompteRemunere`, etc.

- Avant JDK 5, les éléments des collections étaient **déclarés** de type `Object` ;
- Impossible de déclarer un seul type d'objets pour une collection :
 - contrairement aux tableaux p.e. : `String []`

Avant JDK 5 : collections non génériques

Une collection d'objets contient le plus souvent un seul type d'objets et éventuellement ses dérivés : `Compte`, `CompteRemunere`, etc.

- Avant JDK 5, les éléments des collections étaient **déclarés** de type `Object` ;
- Impossible de déclarer un seul type d'objets pour une collection :
 - contrairement aux tableaux p.e. : `String []`

Exemple collection non générique (1)

```
Compte c = new Compte(100);  
CompteDecouvert cd= new CompteDecouvert(300, 200);  
String intrus= "cei_est_un_intrus";  
ArrayList avant = new ArrayList();  
avant.add(cd); avant.add(c);  
avant.add(intrus); // ceci compile?
```

- Ce programme compile-t-il ?
- si oui pourquoi ?

Exemple collection non générique (2)

```
Compte c = new Compte(100);
CompteDecouvert cd= new CompteDecouvert(300, 200);
String intrus= "ceci_est_un_intrus";
ArrayList avant = new ArrayList();
avant.add(cd); avant.add(c); avant.add(intrus);
double res=0;
for (Object o: avant){
    // res = res + o.getSolde(); // compile ?
    res = res + ((Compte)o).getSolde(); // compile? execute
}
```

- quelles lignes ne compilent pas ici ?
- d'autres problèmes ?

Exemple collection non générique (3)

```
....
avant.add(intrus); // compile car tout est Object
....
for (Object o: avant){
    // compile pas, car Object n'a pas de méthode getSolde()
    // res = res + o.getSolde();
    ....
    // compile?: oui, exécute?: seulement si tous
    // les élément sont convertibles en Compte.
    res = res + ((Compte)o).getSolde();
}
```

- si un élément ajouté dans la liste n'est pas de type Compte, on le saura **uniquement** à l'exécution (un peu tard).

Le même avec collections génériques

```
ArrayList<Compte> gen= new ArrayList<> ();  
gen.add(cd); gen.add(c);  
gen.add(intrus);  
res=0;  
for (Compte o: gen){  
    res = res + o.getSolde(); // pas de cast  
}
```

- Ce programme compile-t-il ?
- a-t-on besoin d'un cast sur les éléments de la liste ? pourquoi ?

Le même avec collections génériques (2)

```
ArrayList<Compte> gen= new ArrayList<> ();  
gen.add(cd); gen.add(c);  
gen.add(intrus); // ne compile pas!  
res=0;  
for (Compte o: gen){  
    res = res + o.getSolde(); // pas besoin de cast  
}
```

- L'ajout d'un élément »hors » de Compte échoue à la compilation ⇒ pas d'intrus possible.
- Pas besoin de cast, puisque le compilateur assure que tout le monde a le type Compte !
- Code plus simple et clair, exécution plus efficace.

Structures abstraites grâce à la généricité

Les bibliothèques Java spécifient et implantes des **structures de données abstraites** en utilisant des interfaces et classes **génériques**.

- Les méthodes et algorithmes ne dépendent pas (en général) de la **nature des éléments**, mais de **leur organisation**.
 - **Algorithmes** : de tri, de recherche, ou de calcul de la taille sont toujours les mêmes pour tous types d'éléments ;
- **En pratique** :
 - la définition du type des éléments (<E>) reste abstrait dans l'implantation des opérations.
 - le typage est + précis, les programmes + robustes, + simples et efficaces.
- **Conséquence** : bibliothèques hautement ré-utilisables !

3. Définir ses propres génériques

Définir ses propres génériques : cellules génériques

```
public interface Cell<T> {
    T get();
    void set(T v);
}

public class CellImpl<T> implements Cell<T> {
    private T val;    // contenu

    public CellImpl(T init){ val = init; }
    public T get(){return val;}
    public void set(T v) { val=v;}
}
```

- **Cell<T>** : interface pour cellules à contenu quelconque
- **CellImpl<T>** : implantation générique

Cellules génériques

```
public interface Cell<T> {  
    T get();  
    void set(T v);  
}
```

```
public static void main(...) {  
    Cell<String> cs = new CellImpl<String>("ABC");  
    Cell<Compte> cc = new CellImpl<Compte>(new Compte());  
    String s = cs.get();  
    Compte c = cc.get();  
}
```

- Les variables sont déclarées de type **Cell<T>**
- et non pas de type CellImpl<T>

Quel intérêt ?

Cellules génériques

```
public interface Cell<T> {  
    T get();  
    void set(T v);  
}
```

```
public static void main(...) {  
    Cell<String> cs = new CellImpl<String>("ABC");  
    Cell<Compte> cc = new CellImpl<Compte>(new Compte());  
    String s = cs.get();  
    Compte c = cc.get();  
}
```

- Les variables sont déclarées **Cell<T>**
- et non pas de type **CellImpl<T>**

Pourquoi ? ⇒ **changer l'implantation** sans changer programme utilisateur/tests.