Programmation Java avancée

Animé par :

Dr. Hafedh Nefzi

Chapitre 2

LA GÉNÉRICITÉ

1. Définitions & exemples

Paramétrisation

- 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 ⇒ code figé pour t, non réutilisable.
- méthode paramétrée par n'importe quel tableau x ⇒ code réutilisable pour tout tableau concret, de n'importe quelle taille.

Paramétrisation

- **Par données** : une méthode est paramétrable par variables désignant des valeurs quelconques (entiers, booléens, une instance d'un objet)

⇒ *Code réutilisable sur n'importe quelle donnée (du bon type).*

- **Par types** : une classe ou méthode est paramétrable par variables désignant des types quelconques

⇒ *Réutilisable sur n'importe quel type concret.*

Définition

- Paramétrisation d'un morceau de code (classes, méthodes), par un type.
- 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
  - Réutilisable pour n'importe quel T :

    liste = *new ArrayList<**Compte**>();* *Ici, T=Compte.*

Exemple

```
public class ArrayList<E>
             implements List<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> elt) : ajoute un élément de type ***E*** dans la liste ;

Exemple (2)

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

2. La généricité : pourquoi ?

Intérêts de la généricité

- Le typage est plus précis ⇒ 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.

Avant JDK5

- 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 ;
- Que pensez-vous de ce code ?

```
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);
```

Avant JDK5

```
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();
    res = res +((Compte)o).getSolde();
}
```

- Quelles lignes ne compilent pas ici ?
- D'autres problèmes ?

Avant JDK5

```
....
avant.add(intrus); // compile car tout est Object
....
for (Object o: avant){
    // Ne compile pas, car Objet 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).

Collections génériques (1)

```
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 élément de la liste ? pourquoi ?

Collections génériques (1)

```
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
}
```

- 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.

Intérêts de la généricité

- Les bibliothèques Java spécifient et implantent 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 réutilisables !

3. 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 de type Cell<T> et non pas de type CellImpl<T>
- Quel intérêt ? ⇒ changer l'implantation sans changer programme utilisateur/tests.

Exercice

Créer une classe générique Stock<T> qui contiendra une liste d'objets de type T et offrira les fonctionnalités suivantes :

- ajouter(T item): ajoute un élément au stock.
- supprimer(T item): supprime un élément du stock.
- afficher(): affiche tous les éléments du stock.

Créer deux classes Livre et Electronique, qui représenteront des types de produits différents.

Classe Livre : titre, auteur, prix.

Classe Electronique : nom, marque, prix.

Créer une classe Main pour tester :

- Créer un stock de Livre et un stock de Electronique.
- Ajouter des éléments à chaque stock.
- Afficher le contenu des stocks.
- Supprimer un élément et afficher le stock mis à jour.

Questions ?

>> Les interfaces fonctionnelles