Chapitre 6

Généricité

Thibaud Martinez
thibaud.martinez@dauphine.psl.eu

Dauphine | PSL
UNIVERSITÉ PARIS

Définition

La **généricité** ou **programmation générique** consiste à écrire un code qui peut être réutilisé pour des objets de nombreux types différents.

Les types (classes et interfaces) peuvent être des paramètres lors de la définition de classes, interfaces et méthodes.

Ainsi, la programmation générique fait appel à des **paramètres de type (*type parameters*)**.

Exemple : ArrayList

La classe `ArrayList` définit une liste redimensionnable (contrairement aux tableaux (*arrays*) pour lesquels la taille est fixée).

`ArrayList` est une **classe générique** : la classe prend en paramètre un type qui définit le type des objets de la liste.

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

Le compilateur va vérifier qu'on n'insère pas d'objets de mauvais type.

```
files.add(new File("..")); // can only add String objects to an ArrayList<String>
```

💥 Le code ci-dessus ne compilera pas.

Définir une classe générique

Une classe générique est une classe comportant un ou plusieurs paramètres de types.

```
public class Pair<T> {
    private T premier;
    private T second;

    public Pair(T premier, T second) {
        this.premier = premier;
        this.second = second;
    }

    public T getPremier() { return premier; }
    public T getSecond() { return second; }

    public void setPremier(T newValue) { premier = newValue; }
    public void setSecond(T newValue) { second = newValue; }
}
```

Une classe générique peut avoir plus d'un paramètres de type.

```
public class Pair<T, U> { ... }
```

✨ Il est courant d'utiliser des lettres majuscules pour les variables de type, et de les garder courtes.

Instancier des classes génériques

```
Pair<String> p1 = new Pair<String>("Foo", "Bar");
```

On peut également utiliser la syntaxe *"diamond"*.

```
Pair<String> p2 = new Pair<>("Foo", "Bar");
```

Enfin, il est possible d'utiliser l'**inférence de type** lorsqu'on instancie une classe générique.

```
var p3 = new Pair<String>("Foo", "Bar");
```

Définir une méthode générique

On peut également aussi définir une méthode unique avec des paramètres de type.

```
class Tableau {
    public static <T> T trouverMilieu(T[] a) {
        return a[a.length / 2];
    }
}
```

Il est possible de définir des méthodes génériques, à la fois dans les **classes ordinaires** et dans les **classes génériques**.

Appel d'une méthode générique

```
String[] l = {"Steven", "Georges", "Francis"};
String milieu = Tableau.<String>trouverMilieu(l);
```

Dans ce cas (et dans la plupart des cas), on peut **omettre** le paramètre de type `<String>` de l'appel de méthode. Le compilateur dispose de suffisamment d'informations pour **déduire** la méthode que l'on souhaite appeler.

```
milieu = Tableau.trouverMilieu(l);
```

Types élémentaires et généricité

```
int[] nombres = {1, 2, 3, 4, 5};
int milieu = Tableau.<int>trouverMilieu(nombres);
```

💥 Le code ci-dessus ne compile pas. Il n'est possible d'utiliser la généricité **qu'avec des objets** et donc **pas avec les types élémentaires**.

→ La solution : **utiliser des *wrapper classes***.

```
Integer[] nombres = {1, 2, 3, 4, 5};
Integer milieu = Tableau.<Integer>trouverMilieu(nombres);
```

ℹ️ Cette limitation est due au fait, qu'en Java, la généricité est implémentée avec la méthode d'effacement de type (***type erasure***).

Wrapper classes

Les **classes enveloppes (*wrapper classes*)** permettent d'utiliser des types de données élémentaires en tant qu'objets.

Ce sont les classes suivantes : `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Boolean` et `Character`.

Autoboxing

C'est la **conversion automatique** que le compilateur effectue entre les types primitifs et les objets des *wrapper classes* correspondantes.

L'*autoboxing* est appliqué quand une variable de type élémentaire est :

- affectée à une variable de la *wrapper class* correspondante;

```
Character ch = 'a';
```

- passée en tant qu'argument à une méthode qui attend un objet de la *wrapper class* correspondante.

```
var l = new ArrayList<Integer>();
l.add(10);
```

Bornes pour les paramètres de type

Les **bornes (*bounds*)** permettent de **restreindre les types** qui peuvent être utilisés comme paramètres de type dans un type générique.

Par exemple, une méthode peut n'accepter que des instances de `Number` ou de ses sous-classes.

```
public <T extends Number> void afficheEntier(T n){
    System.out.println(n.intValue());
}
```

Bornes multiples

Un paramètre de type peut avoir plusieurs bornes.

```
<T extends B1 & B2 & B3>
```

Un paramètre de type avec plusieurs bornes est un **sous-type** de tous les types énumérés dans la limite.

⚠️ Si l'une des bornes est une classe, elle doit être spécifiée en premier.

```
class A { }
interface B { }
interface C { }

class D <T extends A & B & C> { }
```

Généricité et héritage

On considère la hiérarchie de classes suivantes :

```
public class A { }
public class B extends A { }
public class C extends A { }
```

💥 On ne peut pas écrire :

```
ArrayList<A> listA = new ArrayList<B>();
```

Autrement dit, `ArrayList<A>` n'est pas un sous-type de `ArrayList<B>`.

Wildcard types

Le mot-clé *wildcard* `?` permet de résoudre ce problème.

```
ArrayList< ? extends A> listA = new ArrayList<B>();
```

`ArrayList< ? extends A>` désigne une liste d'objets qui sont des instances de la classe `A`, ou des sous-classes de `A` (par exemple `B` et `C`).

→ on pourra appeler des méthodes de `A` sur les éléments de `listA`.