Méthodes et classes génériques

Parfois, on souhaite créer une classe mais on ne souhaite pas préciser le type exact de tel ou tel attribut. C'est souvent le cas quand la classe sert de conteneur à un autre type de classe. En Java, il est possible de créer des méthodes et des classes dont certains types sont des paramètres qui seront résolus au moment de l'invocation et de l'instanciation. On parle alors de méthodes et de classes génériques.

L'exemple de la classe ArrayList

En Java, l'API standard fournit un ensemble de classes que l'on appelle couramment les collections. Ces classes permettent de gérer un ensemble d'objets. Elles apportent des fonctionnalités plus avancées que les tableaux. Par exemple la classe java.util.ArrayList permet de gérer une liste d'objets. Cette classe autorise l'ajout en fin de liste, l'insertion, la suppression et bien évidemment l'accès à un élément selon son index et le parcours complet des éléments.

```
package com.cgi.udev;

import java.util.ArrayList;
import java.util.List;

public class TestArrayList {

  public static void main(String[] args) {
    List list = new ArrayList();

    list.add("bonjour le monde");
    list.add(1); // boxing ! list.add(Integer.valueOf(1));
    list.add(new Object());

    String s1 = (String) list.get(0);
    String s2 = (String) list.get(1); // ERREUR à l'exécution : ClassCastException
    String s3 = (String) list.get(2); // ERREUR à l'exécution : ClassCastException
```

```
    }

}
```

Pour des instances de la classe ArrayList, on peut ajouter des éléments avec la méthode ArrayList.add et accéder à un élément selon son index avec la méthode ArrayList.get. Dans l'exemple précédent, on voit que cela n'est pas sans risque. En effet, un objet de type ArrayList peut contenir tout type d'objet. Donc quand le programme accède à un élément d'une instance de ArrayList, il doit réaliser explicitement un transtypage (*cast*) avec le risque que cela suppose de se tromper de type. Ce type de classe exige donc beaucoup de rigueur d'utilisation pour les développeurs.

Une situation plus simple serait de pouvoir déclarer en tant que développeur qu'une instance de ArrayList se limite à un type d'éléments : par exemple au type String. Ainsi le compilateur pourrait signaler une erreur si le programme tente d'ajouter un élément qui n'est pas compatible avec le type String ou s'il veut récupérer un élément dans une variable qui n'est pas d'un type compatible. Les classes et les méthodes génériques permettent de gérer ce type de situation. Elles sont une aide pour les développeurs afin d'écrire des programmes plus robustes.

Création et assignation d'une classe générique

La classe ArrayList et l'interface List sont justement une classe générique et une interface générique supportant un type paramétré.

> List est une interface implémentée notamment par la classe ArrayList.

Il est possible, par exemple, de déclarer qu'une instance est une liste de chaînes de caractères :

```
List<String> list = new ArrayList<String>();
```

On ajoute entre les signes **<** et **>** le paramètre de type géré par la liste. À partir de cette information, le compilateur va pouvoir nous aider à résoudre les ambiguïtés. Il peut maintenant déterminer si un élément peut être ajouté ou assigné à une variable sans nécessiter un transtypage explicite du développeur.

```
list.add("bonjour");
String s = list.get(0); // l'opération de transtypage n'est plus nécessaire
```

Par contre :

```
list.add(1); // Erreur de compilation : type String attendu

Object o = "je suis une chaîne affectée à une variable de type Object";
list.add(o); // Erreur de compilation : type String attendu

Voiture v = (Voiture) list.get(0); // Erreur de compilation Voiture n'hérite pas de String
```

Pour les types paramétrés, le principe de substitution s'applique. Comme la classe String hérite de la classe Object, il est possible de récupérer un élément de la liste dans une variable de type Object :

```
Object o = list.get(0); // OK
```

Une classe générique peut permettre de déclarer plusieurs types paramétrés. Par exemple, la classe java.util.HashMap permet de créer des tableaux associatifs (parfois appelés dictionnaires ou plus simplement *maps*) pour associer une clé à une valeur. La classe HashMap permet de spécifier le type de la clé et le type de la valeur. Pour créer un tableau associatif entre le nom d'une personne (type String) et une instance de la classe *Personne*, on peut écrire :

```
Map<String, Personne> tableauAssociatif = new HashMap<String, Personne>();
```

> Map est une interface implémentée notamment par la classe HashMap.

Notation en diamant

Lors de l'initialisation, il n'est pas nécessaire de préciser le type des paramètres à droite de l'expression. Le compilateur peut réaliser une inférence de types à partir de la variable à gauche de l'expression :

```
Map<String, Personne> tableauAssociatif = new HashMap<>();
List<Integer> listeDeNombres = new ArrayList<>();
```

Il s'agit d'un raccourci d'écriture qui évite de se répéter. On appelle la notation **<>**, la notation en diamant.

Substitution et type générique

Avec l'héritage, nous avons vu que nous pouvons affecter à une variable (ou à un paramètre ou un à attribut) une référence d'un objet du même type ou d'un type qui en hérite. On appelle cela le principe de substitution.

```
Object obj = new String();
```

Dans l'exemple ci-dessus, il est possible d'affecter un objet du type String à une variable de type Object car String hérite de Object. Avec les types génériques, le principe de substitution est possible mais devient un peu plus complexe. Par exemple :

```
List<Object> listeString = new ArrayList<String>(); // ERREUR DE COMPILATION
```

Il n'est pas possible d'affecter une ArrayList de String à une variable de type ArrayList de Object. En effet, si cela était autorisé, il serait alors possible d'ajouter avec la méthode List.add n'importe quel objet de type Object ou d'un type héritant de Object. Donc un développeur pourrait ajouter à cette liste une instance d'une classe *Voiture* par exemple sans que le compilateur puisse détecter le problème :

```
listeString.add(new Voiture()); // Il vaut mieux ne pas pouvoir faire cela !
```

Pour les types génériques, il est nécessaire d'introduire la notion de type borné (*bounded type*) pour pouvoir gérer la substitution correctement. Mais avant d'aller plus loin, il est important de comprendre qu'il existe deux cas fondamentaux. Prenons une exemple de classes qui héritent les unes des autres : *Vehicule*, *Voiture*, *VoitureDeCourse*.

```
package com.cgi.udev;


public class Vehicule {
  // ...
}
```

```
package com.cgi.udev;
```

```
public class Voiture extends Vehicule {
  // ...
}
```

```
package com.cgi.udev;

public class VoitureDeCourse extends Voiture {
  // ...
}
```

Si nous créons une instance de ArrayList pour le type *Voiture* :

```
ArrayList<Voiture> listeVoitures = new ArrayList<>();
```

Si on souhaite ajouter des objets dans cette liste, le principe de substitution nous assure que nous pouvons ajouter sans risque une instance de la classe *Voiture* ou une instance de la classe *VoitureDeCourse* (puisqu'une *VoitureDeCourse* est une *Voiture*).

```
listeVoitures.add(new Voiture());
listeVoitures.add(new VoitureDeCourse());
```

Si on souhaite accéder à une élément de cette liste, le principe de substitution nous dit que nous pouvons affecter sans risque un élément de cette liste à une variable de type *Voiture* ou de type *Vehicule* (puisqu'une *Voiture* est un *Vehicule*).

```
Voiture voiture = listeVoitures.get(0);
Vehicule vehicule = listeVoitures.get(0);
```

Il y a donc une différence selon que nous souhaitons ajouter un élément à cette liste ou que nous souhaitons consulter un élément de cette liste. L'ajout s'apparente à utiliser le type paramétré comme paramètre d'entrée et la consultation s'apparente à utiliser le type paramétré comme paramètre de sortie.

Une liste de *Voiture* peut donc aussi être considérée comme :

- une liste de quelque chose qui est au mieux de type *Voiture* dans le cas où l'on souhaite uniquement consulter les éléments de la liste.
- une liste de quelque chose qui est au moins de type *Voiture* dans le cas où on ne souhaite qu'ajouter de nouveaux éléments à la liste.

Il est possible d'exprimer cela en Java. Pour le premier cas, *Voiture* correspond à la borne supérieure (*upper bounded type*) et nous pouvons écrire l'expression suivante :

```
List<? extends Voiture> listePourConsultation = listeVoitures;
Voiture voiture = listePourConsultation.get(0);
```

L'expression **<? extends Voiture>** désigne une **capture** et permet au compilateur de déterminer l'ensemble des classes acceptables.

Pour le second cas, *Voiture* correspond à la borne inférieure (*lower bounded type*) et nous pouvons écrire l'expression suivante :

```
List<? super Voiture> listePourAjout = listeVoitures;
listePourAjout.add(new Voiture());
listePourAjout.add(new VoitureDeCourse());
```

Il est également possible d'utiliser uniquement le caractère de subsitution **?** dans la déclaration de la capture :

```
List<?> listePourAjout = listeVoitures;
```

Dans ce cas, on ne fournit aucune information au compilateur sur le type paramétré de l'instance de la classe.

> Pour une classe supportant plusieurs types génériques, on peut au besoin déclarer une capture pour chaque type :

```
Map<?, ? extends Personne> tableauAssociatif = new HashMap<String, Personne>();
```

La déclaration de capture est surtout utile pour la création de méthodes et classes supportant les types génériques.

Écrire une méthode générique

L'utilisation des captures devient utile lorsque l'on veut écrire une méthode générique qui supporte les types paramétrés. Reprenons notre exemple ci-dessus des classes *Vehicule*, *Voiture* et *VoitureDeCourse*. La classe *Vehicule* définit la propriété *vitesse* accessible en lecture :

```
package com.cgi.udev;


public class Vehicule {
```

```
  private int vitesse;

  public int getVitesse() {
    return vitesse;
  }

}
```

Nous voulons ajouter la méthode de classe *getPlusRapide* qui retourne le véhicule le plus rapide parmi une liste de véhicules :

```
package com.cgi.udev;

import java.util.List;

public class Vehicule {

  private int vitesse;

  public int getVitesse() {
    return vitesse;
  }

  public static Vehicule getPlusRapide(List<Vehicule> vehicules) {
    Vehicule plusRapide = null;
    int vitesse = 0;
    for (Vehicule vehicule : vehicules) {
      if(vehicule.getVitesse() >= vitesse) {
        vitesse = vehicule.getVitesse();
        plusRapide = vehicule;
      }
    }
    return plusRapide;
  }
}
```

Si nous nous contentons de cette implémentation, nous allons certainement rencontrer quelques problèmes lors de l'utilisation de la méthode *Vehicule.getPlusRapide* :

```
List<Voiture> listeVoitures = new ArrayList<>();
listeVoitures.add(new Voiture());
```

```
listeVoitures.add(new VoitureDeCourse());

Vehicule plusRapide = Vehicule.getPlusRapide(listeVoitures); // ERREUR DE COMPILATION
```

Le code ci-dessus ne compile pas. En effet, on tente de passer en paramètre à la méthode *Vehicule.getPlusRapide* une liste de type *Voiture* alors que la méthode est écrite pour une liste de type *Vehicule*. Nous pourrions utiliser la surcharge en fournissant une implémentation pour chaque type de liste, mais la bonne solution est de déclarer *Vehicule.getPlusRapide* comme une méthode générique :

```
package com.cgi.udev;

import java.util.ArrayList;
import java.util.List;

public class Vehicule {

  private int vitesse;

  public int getVitesse() {
    return vitesse;
  }

  public static <T extends Vehicule> T getPlusRapide(List<T> vehicules) {
    T plusRapide = null;
    int vitesse = 0;
    for (T vehicule : vehicules) {
      if(vehicule.getVitesse() >= vitesse) {
        vitesse = vehicule.getVitesse();
        plusRapide = vehicule;
      }
    }
    return plusRapide;
  }
}
```

Pour déclarer une méthode générique, il faut décrire le type ou les types paramétrés supportés entre **< >**. Pour l'exemple ci-dessus, on utilise la capture **<T extends Vehicule>**. T est le nom du type générique que l'on peut utiliser dans la signature et le code de la méthode. Dans notre exemple T représente donc le type *Vehicule* ou un type qui hérite de *Vehicule*. On peut donc parcourir les éléments de type **T** de la liste, lire leur propriété *vitesse* et retourner l'instance pour laquelle la vitesse est la plus élevée.

Maintenant nous pouvons utiliser cette méthode en passant une liste de type *Vehicule*, de *Voiture* ou de *VoitureDeCourse*

```
List<Voiture> listeVoitures = new ArrayList<>();
listeVoitures.add(new Voiture());
listeVoitures.add(new VoitureDeCourse());

Voiture plusRapide = Vehicule.getPlusRapide(listeVoitures);
```

Notez que la méthode *Voiture.getPlusRapide* retourne le type générique **T**. Donc le compilateur infère que si on appelle cette méthode avec une liste de type *Voiture* en paramètre alors cette méthode retourne une instance assignable à une variable de type *Voiture*.

> Par convention un type paramétré s'écrit avec une seule lettre en majuscule :

- T pour identifier un type générique en général
- E pour identifier un type générique qui représente un élément
- K pour identifier un type générique qui est utilisé comme clé (*key*)
- V pour identifier un type générique qui est utilisé comme une valeur
- U, V, W pour identifier une suite de types génériques si la méthode supporte plusieurs types génériques.

Écrire une classe générique

Une classe peut également être générique et supporter un ou plusieurs types paramétrés. Par exemple, si nous voulons implémenter une classe *Paire* qui permet d'associer une instance d'une classe avec une instance d'une autre classe, il suffit d'utiliser des types paramétrés en les déclarant entre **< >** après le nom de la classe :

```
package com.cgi.udev;

public class Paire<U, V> {

  private U valeurGauche;
  private V valeurDroite;

  public Paire(U valeurGauche, V valeurDroite) {
    this.valeurGauche = valeurGauche;
    this.valeurDroite = valeurDroite;
```

```
  }

  public U getValeurGauche() {
    return valeurGauche;
  }

  public V getValeurDroite() {
    return valeurDroite;
  }

  @Override
  public String toString() {
    return valeurGauche + " " + valeurDroite;
  }

}
```

La classe *Paire* peut maintenant être utilisée pour associer n'importe quel type d'instances :

```
Paire<String, Integer> paireStringInteger = new Paire<>("test", 1);

Paire<Voiture, Voiture> paireVoitureVoiture = new Paire<>(new Voiture(), new Voiture());
```

Comme pour les méthodes, il est possible de préciser une capture pour les types paramétrés :

```
public class Paire<U extends Number, V> {

  private U valeurGauche;
  private V valeurDroite;

  public Paire(U valeurGauche, V valeurDroite) {
    this.valeurGauche = valeurGauche;
    this.valeurDroite = valeurDroite;
  }

  public U getValeurGauche() {
    return valeurGauche;
  }

  public V getValeurDroite() {
```

```
    return valeurDroite;
  }

  @Override
  public String toString() {
    return valeurGauche + " " + valeurDroite;
  }
}
```

En précisant **<U extends Number, V>** dans la déclaration de la classe, nous limitons le premier type paramétré au type Number ou un type qui en hérite.

> La classe Number est la classe parente des classes enveloppes Integer, Long, Short, Byte, Float et Double.

```
Paire<Integer, String> paireIntegerString = new Paire<>(1, "Test");
Paire<Float, String> paireFloatString = new Paire<>(1.3f, "Test");
```

Limitations

Les méthodes et les classes génériques ont des limitations.

Les types paramétrés ne s'appliquent que pour des classes. On ne peut pas spécifier un type primitif. Si on désire créer une instance de ArrayList pour des nombres, alors on peut passer par la classe enveloppe Integer :

```
ArrayList<Integer> listeDeNombres = new ArrayList<Integer>();
```

La déclaration d'un type paramétré ne fait pas partie du nom d'une classe. Donc il n'est pas possible de spécifier un type paramétré avec le mot-clé **instanceof** :

```
if (listeVoiture instanceof List<Voiture>) { // ERREUR DE COMPILATION
  // ...
}
```

Il n'est pas possible d'instancier un type paramétré dans le corps d'une méthode générique :

```
 public static <T> doSomething(List<T> l) {
   l.add(new T()); // ERREUR DE COMPILATION
 }
```

Il n'est pas possible de déclarer un attribut de classe (**static**) en utilisant un type paramétré :

```
public class Test<T> {

  private static T attribut; // ERREUR DE COMPILATION

}
```

Il n'est pas possible de créer des tableaux en spécifiant des types paramétrés :

```
List<String>[] tableau = new List<String>[10]; // ERREUR DE COMPILATION
```

Il n'est pas possible d'utiliser un type paramétré dans une expression **catch** :

```
public static <T extends Exception> void doSomething() {
  try {
    // ...
  } catch (T t) { // ERREUR DE COMPILATION
    // ...
  }
}
```

Il n'est pas possible de surcharger (*overload*) une méthode en ne changeant que le type paramétré d'un paramètre :

```
public class Test {

  public void doSomething(List<String> l) {
    // ...
  }

  public void doSomething(List<Integer> l) { // ERREUR DE COMPILATION
    // ...
  }
}
```

> Beaucoup des limitations des classes et des méthodes génériques viennent de ce que l'on appelle *l'effacement du type* (*type erasure*). Les types paramétrés ne sont pas conservés dans le bytecode produit par le compilateur.

Pour l'exemple ci-dessus, la suppression du type par le compilateur conduit à la classe suivante :

```
public class Test {

  public void doSomething(List l) {
    // ...
  }

  public void doSomething(List l) {
    // ...
  }
}
```

Donc, le résultat de la compilation amènerait à déclarer une classe avec deux méthodes strictement identiques. Voilà pourquoi il n'est pas possible de surcharger une méthode juste en changeant le type paramétré d'un paramètre.

Created 2025-02-10 21:01:21 UTC by Nicolas
Updated 2025-02-10 21:03:59 UTC by Nicolas