La généricité en Java

Pour assimiler ce concept, ajouté au JDK depuis la version 1.5, nous allons essentiellement travailler avec des exemples tout au long de ce chapitre. Le principe de la généricité est de faire des classes qui n'acceptent qu'un certain type d'objets ou de données de façon dynamique !

Avec ce que nous avons appris au chapitre précédent, vous avez sûrement poussé un soupir de soulagement lorsque vous avez vu que ces objets acceptent tous les types de données. Par contre, un problème de taille se pose : lorsque vous voudrez travailler avec ces données, vous allez devoir faire un `cast` ! Et peut-être même un `cast` de `cast`, voire un `cast` de `cast` de `cast`...

C'est là que se situe le problème… Mais comme je vous le disais, depuis la version 1.5 du JDK, la généricité est là pour vous aider !

Principe de base

Bon, pour vous montrer la puissance de la généricité, nous allons tout de suite voir un cas de classe qui ne l'utilise pas.

Il existe un exemple très simple que vous pourrez retrouver aisément sur Internet, car il s'agit d'un des cas les plus faciles permettant d'illustrer les bases de la généricité. Nous allons coder une classe `Solo`. Celle-ci va travailler avec des références de type `String`. Voici le diagramme de classe de cette dernière en figure suivante.

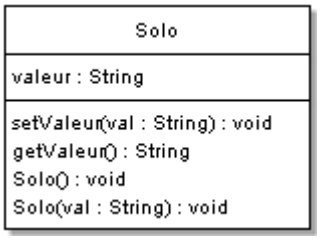
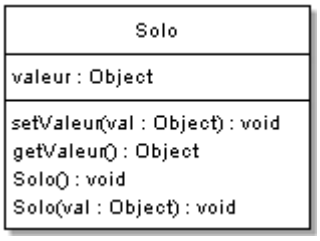


Diagramme de la classe Solo

Vous pouvez voir que le code de cette classe est très rudimentaire. On affecte une valeur, on peut la mettre à jour et la récupérer… Maintenant, si je vous demande de me faire une classe qui permet de travailler avec n'importe quel type de données, j'ai une vague idée de ce que vous allez faire. Ne serait-ce pas quelque chose s'approchant de la figure suivante ?



Classe Solo travaillant avec des Object

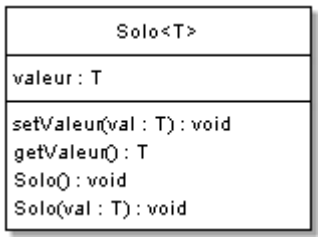
J'en étais sûr. Créez la classe `Solo`, ainsi qu'une classe avec une méthode `main`. Si vous voulez utiliser les données de l'objet `Solo`, vous allez devoir faire un `cast`. Testez ce code dans votre `main` :

```
public class Test {
  public static void main(String[] args) {
    Solo val = new Solo(12);
    int nbre = val.getValeur();
  }
}
```

Vous constatez que vous essayez vainement de mettre un objet de type `Object` dans un objet de type `Integer` : c'est interdit ! La classe `Object` est plus globale que la classe `Integer`, vous ne pouvez donc pas effectuer cette opération, sauf si vous castez votre objet en `Integer` comme ceci :

```
Solo val = new Solo(12);
int nbre = (Integer)val.getValeur();
```

Pour le moment, on peut dire que votre classe peut travailler avec tous les types de données, mais les choses se corsent un peu à l'utilisation. Vous serez donc sans doute tentés d'écrire une classe par type de donnée (`SoloInt`, `SoloString`, etc.). Et c'est là que la généricité s'avère utile, car avec cette dernière, vous pourrez savoir ce que contient votre objet `Solo` et n'aurez qu'une seule classe à développer ! Voilà le diagramme de classe de cet objet en figure suivante.



Objet générique

Et voici son code :

```
public class Solo<T> {

  //Variable d'instance
  private T valeur;

  //Constructeur par défaut
  public Solo(){
    this.valeur = null;
  }

  //Constructeur avec paramètre inconnu pour l'instant
  public Solo(T val){
    this.valeur = val;
  }

  //Définit la valeur avec le paramètre
  public void setValeur(T val){
    this.valeur = val;
  }
```

```
  //Retourne la valeur déjà « castée » par la signature de la méthode !
  public T getValeur(){
        return this.valeur ;
  }
}
```

Impressionnant, n'est-ce pas ? Dans cette classe, le `T` n'est pas encore défini. Vous vous en occuperez à l'instanciation de la classe. Par contre, une fois instancié avec un type, l'objet ne pourra travailler qu'avec le type de données que vous lui avez spécifié ! Exemple de code :

```
public static void main(String[] args) {
  Solo<Integer> val = new Solo<Integer>(12);
  int nbre = val.getValeur();
}
```

Ce code fonctionne très bien, mais si vous essayez de faire ceci :

```
public static void main(String[] args) {
  Solo<Integer> val = new Solo<Integer>("toto");
  //Ici, on essaie de mettre une chaîne de caractères à la place d'un entier
  int nbre = val.getValeur();
}
```

… ou encore ceci :

```
public static void main(String[] args) {
  Solo<Integer> val = new Solo<Integer>(12);
  val.setValeur(12.2f);
}
```

… vous obtiendrez une erreur dans la zone de saisie. Ceci vous indique que votre objet ne reçoit pas le bon type d'argument, il y a donc un conflit entre le type de données que vous avez passé à votre instance lors de sa création et le type de données que vous essayez d'utiliser dans celle-ci ! Par contre, vous devez savoir que cette classe ne fonctionne pas seulement avec des `Integer`. Vous pouvez utiliser tous les types que vous souhaitez ! Voici une démonstration de ce que j'avance :

```
public static void main(String[] args) {
  Solo<Integer> val = new Solo<Integer>();
  Solo<String> valS = new Solo<String>("TOTOTOTO");
  Solo<Float> valF = new Solo<Float>(12.2f);
}
```

Vous avez certainement remarqué que je n'ai pas utilisé ici les types de données que vous employez pour déclarer des variables de type primitif ! Ce sont les classes de ces types primitifs.

En effet, lorsque vous déclarez une variable de type primitif, vous pouvez utiliser ses classes enveloppes (on parle aussi de classe wrapper) ; elles ajoutent les méthodes de la classe `Object` à vos types primitifs ainsi que des méthodes permettant de caster leurs valeurs, etc. À ceci, je dois ajouter que depuis Java 5, est géré ce qu'on appelle l'`autoboxing`, une fonctionnalité du langage permettant de transformer automatiquement un type primitif en classe wrapper (on appelle ça le `boxing`) et inversement, c'est-à-dire une classe wrapper en

type primitif (ceci s'appelle l'`unboxing`). Ces deux fonctionnalités forment l'`autoboxing`. Par exemple :

```
public static void main(String[] args){
  int i = new Integer(12);         //Est équivalent à int i = 12
  double d = new Double(12.2586);  //Est équivalent à double d = 12.2586
  Double d = 12.0;
  Character c = 'C';
  al = new ArrayList();
  //Avant Java 5 il fallait faire al.add(new Integer(12))
  //Depuis Java 5 il suffit de faire
  al.add(12);
}
```

Plus loin dans la généricité !

Vous devez savoir que la généricité peut être multiple ! Nous avons créé une classe `Solo`, mais rien ne vous empêche de créer une classe `Duo`, qui elle prend deux paramètres génériques ! Voilà le code source de cette classe :

```
public class Duo<T, S> {
  //Variable d'instance de type T
  private T valeur1;

  //Variable d'instance de type S
  private S valeur2;

  //Constructeur par défaut
  public Duo(){
    this.valeur1 = null;
    this.valeur2 = null;
  }

  //Constructeur avec paramètres
  public Duo(T val1, S val2){
    this.valeur1 = val1;
    this.valeur2 = val2;
  }

  //Méthodes d'initialisation des deux valeurs
  public void setValeur(T val1, S val2){
    this.valeur1 = val1;
    this.valeur2 = val2;
  }

  //Retourne la valeur T
  public T getValeur1() {
    return valeur1;
  }

  //Définit la valeur T
  public void setValeur1(T valeur1) {
    this.valeur1 = valeur1;
  }

  //Retourne la valeur S
  public S getValeur2() {
    return valeur2;
  }

  //Définit la valeur S
  public void setValeur2(S valeur2) {
    this.valeur2 = valeur2;
```

```
  }
}
```

Vous voyez que cette classe prend deux types de références qui ne sont pas encore définis. Afin de mieux comprendre son fonctionnement, voici un code que vous pouvez tester :

```
public static void main(String[] args) {
  Duo<String, Boolean> dual = new Duo<String, Boolean>("toto", true);
  System.out.println("Valeur de l'objet dual : val1 = " + dual.getValeur1() + ", val2 = " +
dual.getValeur2());

  Duo<Double, Character> dual2 = new Duo<Double, Character>(12.2585, 'C');
  System.out.println("Valeur de l'objet dual2 : val1 = " + dual2.getValeur1() + ", val2 = "
+ dual2.getValeur2());
}
```

Vous voyez qu'il n'y a rien de bien méchant ici. Ce principe fonctionne exactement comme dans l'exemple précédent. La seule différence réside dans le fait qu'il n'y a pas un, mais deux paramètres génériques !

Attends une minute… Lorsque je déclare une référence de type `Duo<String, Boolean>`, je ne peux plus la changer en un autre type !

En fait, non. Si vous faites :

```
public static void main(String[] args) {
  Duo<String, Boolean> dual = new Duo<String, Boolean>("toto", true);
  System.out.println("Valeur de l'objet dual: val1 = " + dual.getValeur1() + ", val2 = " +
dual.getValeur2());

}
```

… vous violez la contrainte que vous avez émise lors de la déclaration du type de référence ! Vous ne pourrez donc pas modifier la déclaration générique d'un objet. Donc si vous suivez bien, on va pouvoir encore corser la chose !

Généricité et collections

Vous pouvez aussi utiliser la généricité sur les objets servant à gérer des collections. C'est même l'un des points les plus utiles de la généricité !

En effet, lorsque vous listiez le contenu d'un `ArrayList` par exemple, vous n'étiez *jamais* sûrs à 100 % du type de référence sur lequel vous alliez tomber (normal, puisqu'un `ArrayList` accepte tous les types d'objets)… Eh bien ce calvaire est terminé et le polymorphisme va pouvoir réapparaître, plus puissant que jamais !

Voyez comment utiliser la généricité avec les collections :

```
public static void main(String[] args) {

  System.out.println("Liste de String");
  System.out.println("------------------------------");
  List<String> listeString= new ArrayList<String>();
  listeString.add("Une chaîne");
  listeString.add("Une autre");
```

```
  listeString.add("Encore une autre");
  listeString.add("Allez, une dernière");

  for(String str : listeString)
    System.out.println(str);

  System.out.println("\nListe de float");
  System.out.println("------------------------------");

  List<Float> listeFloat = new ArrayList<Float>();
  listeFloat.add(12.25f);
  listeFloat.add(15.25f);
  listeFloat.add(2.25f);
  listeFloat.add(128764.25f);

  for(float f : listeFloat)
    System.out.println(f);
}
```

La généricité sur les listes est régie par les lois vues précédemment : pas de type `float` dans un `ArrayList<String>`.

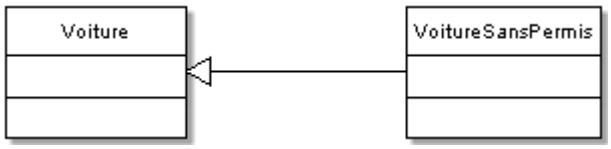
Vu qu'on y va *crescendo*, on pimente à nouveau le tout !

Héritage et généricité

Là où les choses sont pernicieuses, c'est quand vous employez des classes usant de la généricité avec des objets comprenant la notion d'héritage ! L'héritage dans la généricité est l'un des concepts les plus complexes en Java. Pourquoi ? Tout simplement parce qu'il va à l'encontre de ce que vous avez appris jusqu'à présent…

Acceptons le postulat suivant

Nous avons une classe `Voiture` dont hérite une autre classe `VoitureSansPermis`, ce qui nous donnerait le diagramme représenté à la figure suivante.



Hiérarchie de classes

Jusque-là, c'est simplissime. Maintenant, ça se complique :

```
public static void main(String[] args) {
  List<Voiture> listVoiture = new ArrayList<Voiture>();
  List<VoitureSansPermis> listVoitureSP = new ArrayList<VoitureSansPermis>();

  listVoiture = listVoitureSP;   //Interdit !
}
```

Si vous avez l'habitude de la covariance des variables, sachez que cela n'existe pas avec la généricité ! En tout cas, pas sous la même forme.

Imaginez deux secondes que l'instruction interdite soit permise ! Dans `listVoiture`, vous avez le contenu de la liste des voitures sans permis, et rien ne vous empêche d'y ajouter une voiture. Là où le problème prend toute son envergure, c'est lorsque vous voudrez sortir toutes les voitures sans permis de votre variable `listVoiture`. Eh oui ! Vous y avez ajouté une voiture ! Lors du balayage de la liste, vous aurez, à un moment, une référence de type `VoitureSansPermis` à laquelle vous tentez d'affecter une référence de type `Voiture`. Voilà pourquoi ceci est interdit.

Une des solutions consiste à utiliser le `wildcard` : « `?` ». Le fait de déclarer une collection avec le `wildcard`, comme ceci :

```
ArrayList<?> list;
```

… revient à indiquer que notre collection accepte n'importe quel type d'objet. Cependant, nous allons voir un peu plus loin qu'il y a une restriction.

Je vais maintenant vous indiquer quelque chose d'important. Avec la généricité, vous pouvez aller encore plus loin. Nous avons vu comment restreindre le contenu d'une de nos listes, mais nous pouvons aussi l'élargir ! Si je veux par exemple qu'un `ArrayList` puisse avoir toutes les instances de `Voiture` et de ses classes filles… comment faire ?

Ce qui suit s'applique aussi aux interfaces susceptibles d'être implémentées par une classe !

Attention les yeux, ça pique :

```
public static void main(String[] args) {
  //List n'acceptant que des instances de Voiture ou de ses sous-classes
  List<? extends Voiture> listVoitureSP = new ArrayList<VoitureSansPermis>();
}
```

Une application de ceci consiste à écrire des méthodes génériques, par exemple une méthode qui permet de lister toutes les valeurs de notre `ArrayList` cité précédemment :

```
public static void main(String[] args) {

  List<? extends Voiture> listVoitureSP = new ArrayList<VoitureSansPermis>();
  afficher(listVoitureSP);
}

//Méthode générique !
static void afficher(ArrayList<? extends Voiture> list){
  for(Voiture v : list)
    System.out.println(v.toString());
}
```

Eh, attends ! On a voulu ajouter des objets dans notre collection et le programme ne compile plus !

Oui… Ce que je ne vous avais pas dit, c'est que dès que vous utilisez le `wildcard`, vos listes sont verrouillées en insertion : elles se transforment en collections en lecture seule..

En fait, il faut savoir que c'est à la compilation du programme que Java ne vous laisse pas faire : le `wildcard` signifie « tout objet », et dès l'utilisation de celui-ci, la JVM verrouillera la compilation du programme afin de prévenir les risques d'erreurs. Dans notre exemple, il est

combiné avec `extends` (signifiant héritant), mais cela n'a pas d'incidence directe : c'est le `wildcard` la cause du verrou (un objet générique comme notre objet `Solo` déclaré `Solo<?> solo;` sera également bloqué en écriture).

Par contre, ce type d'utilisation fonctionne à merveille pour la lecture :

```
public static void main(String[] args){

  //Liste de voiture
  List<Voiture> listVoiture = new ArrayList<Voiture>();
  listVoiture.add(new Voiture());
  listVoiture.add(new Voiture());

  List<VoitureSansPermis> listVoitureSP = new ArrayList<VoitureSansPermis>();
  listVoitureSP.add(new VoitureSansPermis());
  listVoitureSP.add(new VoitureSansPermis());

  affiche(listVoiture);
  affiche(listVoitureSP);
}

//Avec cette méthode, on accepte aussi bien les collections de Voiture que les collection
de VoitureSansPermis
static void affiche(List<? extends Voiture> list){

  for(Voiture v : list)
    System.out.print(v.toString());
}
```

Avant que vous ne posiez la question, non, déclarer la méthode `affiche(List<Voiture> list) {…}` ne vous permet pas de parcourir des listes de `VoitureSansPermis`, même si celle-ci hérite de la classe `Voiture`.

Les méthodes déclarées avec un type générique sont verrouillées afin de n'être utilisées qu'avec ce type bien précis, toujours pour les mêmes raisons ! Attendez : ce n'est pas encore tout. Nous avons vu comment élargir le contenu de nos collections (pour la lecture), nous allons voir comment restreindre les collections acceptées par nos méthodes.

La méthode :

```
static void affiche(List<? extends Voiture> list){
  for(Voiture v : list)
    System.out.print(v.toString());
}
```

… autorise n'importe quel objet de type `List` dont `Voiture` est la superclasse.

La signification de l'instruction suivante est donc que la méthode autorise un objet de type `List` de n'importe quelle superclasse de la classe `Voiture` (y compris `Voiture` elle-même).

```
static void affiche(List<? super Voiture> list){
  for(Object v : list)
    System.out.print(v.toString());
}
```

Ce code fonctionne donc parfaitement :

```
public static void main(String[] args){
  //Liste de voiture
  List<Voiture> listVoiture = new ArrayList<Voiture>();
  listVoiture.add(new Voiture());
  listVoiture.add(new Voiture());

  List<Object> listVoitureSP = new ArrayList<Object>();
  listVoitureSP.add(new Object());
  listVoitureSP.add(new Object());

  affiche(listVoiture);
}

//Avec cette méthode, on accepte aussi bien les collections de Voiture que les collections
d'Object : superclasse de toutes les classes

static void affiche(List<? super Voiture> list){
  for(Object v : list)
    System.out.print(v.toString());
}
```

L'utilité du `wildcard` est surtout de permettre de retrouver le polymorphisme avec les collections. Afin de mieux cerner l'intérêt de tout cela, voici un petit exemple de code :

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

public class Garage {
  List<Voiture> list = new ArrayList<Voiture>();

  public void add(List<? extends Voiture> listVoiture){
    for(Voiture v : listVoiture)
      list.add(v);

    System.out.println("Contenu de notre garage :");
    for(Voiture v : list)
      System.out.print(v.toString());
   }
}
```

Un petit test rapide :

```
public static void main(String[] args){
  List<Voiture> listVoiture = new ArrayList<Voiture>();
  listVoiture.add(new Voiture());

  List<VoitureSansPermis> listVoitureSP = new ArrayList<VoitureSansPermis>();
  listVoitureSP.add(new VoitureSansPermis());

  Garage garage = new Garage();
  garage.add(listVoiture);
  System.out.println("--------------------------");
  garage.add(listVoitureSP);
}
```

Essayez donc : ce code fonctionne parfaitement et vous permettra de constater que le polymorphisme est possible avec les collections. Je conçois bien que ceci est un peu difficile à comprendre, mais vous en aurez sûrement besoin dans une de vos prochaines applications !

- La généricité est un concept très utile pour développer des objets travaillant avec plusieurs types de données.

- Vous passerez donc moins de temps à développer des classes traitant de façon identique des données différentes.
- La généricité permet de réutiliser sans risque le polymorphisme avec les collections.
- Cela confère plus de robustesse à votre code.
- Vous pouvez coupler les collections avec la généricité !
- Le `wildcard (?)` permet d'indiquer que n'importe quel type peut être traité et donc accepté !
- Dès que le `wildcard (?)` est utilisé, cela revient à rendre ladite collection en lecture seule !
- Vous pouvez élargir le champ d'acceptation d'une collection générique grâce au mot-clé `extends`.
- L'instruction `? extends MaClasse` autorise toutes les collections de classes ayant pour supertype `MaClasse`.
- L'instruction `? super MaClasse` autorise toutes les collections de classes ayant pour type `MaClasse` et tous ses supertypes !
- Pour ce genre de cas, les méthodes génériques sont particulièrement adaptées et permettent d'utiliser le polymorphisme dans toute sa splendeur !