# La généricité en Java

Pour assimiler ce concept, ajouté au JDK depuis la version 1.5, on va 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.

Par contre, un problème de taille se pose : lorsqu'on veut travailler avec ces données, on va devoir faire un cast.

C'est là que se situe le problème. Mais, depuis la version 1.5 du JDK, la généricité est là pour régler ce problème.

## I. Classe générique

### 1. Généricité simple d'une classe

Il existe un exemple très simple permettant d'illustrer les bases de la généricité. Voici le diagramme de classe de la classe **Exemple** en figure suivante.

| **Exemple** |
|---|
| attribut : int |
| setAttribut(val : int)<br>getAttribut() : int<br>Exemple(val : int) |

Classe **Exemple**

Et si on veut créer une classe qui permet de travailler avec n'importe quel type de données.

| **Exemple** |
|---|
| attribut : Object |
| setAttribut(val : Object)<br>getAttribut() : Object<br>Exemple(val : Object) |

Classe **Exemple** qui utilise le type Object

Si on veut utiliser les données de la classe **Exemple**, il faut faire un Cast.

```
class Exemple
{ Object attribut;
   Exemple(int val)
   {  this.attribut=val;
   }
   public Object getAttribut()
   {return this.attribut;}

   public void setAttribut(Object attribut) {
```

```
        this.attribut = attribut;
    }
}
public class Test {
public static void main(String[] args) {
Exemple e = new Exemple(12);
 int nbre=e.getAttribut();
// Incompatible types : Object cannot be converted to int
   }
}
```

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 :

```
Exemple e = new Exemple(12);
int nbre = (Integer)e.getAttribut();
```

Pour le moment, on peut dire que la classe Exemple est une solution pour tous les types de données, mais on sera donc sans doute tenté d'écrire une classe par type de donnée (ExempleInt, ExempleString, etc.). Et c'est là que la généricité s'avère utile, car avec cette dernière, n'aurez qu'une seule classe à développer.

| **Exemple<T>** |
|---|
| attribut : T |
| setAttribut(val : T)<br>getAttribut() : T<br>Exemple(val : T) |

- Classe **générique**

Dans cette classe, le T n'est pas encore défini. On en occupera 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.

```
class Exemple<T>
{ T attribut;
   Exemple(T val)
   { this.attribut=val;
   }
   public T getAttribut()
   {return this.attribut;}

    public void setAttribut(T attribut) {
     this.attribut = attribut;
   }
```

```
}
public class Test {
     public static void main(String[] args) {
      Exemple<Integer> e = new Exemple<Integer>(12);
       int nbre = e.getAttribut();
   }
}
```

**Paramètres de type**

Les conventions de dénomination des paramètres de type sont importantes pour apprendre la généricité de manière approfondie.

Les paramètres de type courants sont les suivants :

- T : Type
- E : Element
- K : Key
- N : Number
- V : Valeur

**Remarques** :

- On note la présence d'un "paramètre de type" nommé ici **T**, dans class Exemple<T>

Il sert à préciser que, dans la définition de classe qui suit, T représente un type quelconque. Ce paramètre T peut alors être utilisé là où un type précis peut l'être normalement. Ici, on le rencontre :

- dans les déclarations du champ attribut,
- dans l'en-tête du constructeur et de la méthode getAttribut et setAttribut.

- Si l'objet ne reçoit pas le bon type d'argument, il y a donc un conflit entre le type de données qu'on a fait passer à l'instance lors de sa création et le type de données qu'on a utilisé.

**Exemple 1**

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

**Exemple 2**

```
public static void main(String[] args) {
  Exemple<Integer> e = new Exemple<Integer>(12);
  e.setAttribut(12.2f);
  //Ici, on essaie de mettre un nombre à virgule flottante à la place d'un entier
}
```

**Solution**

```
public static void main(String[] args) {
  Exemple<Integer> val = new Exemple<Integer>(5);
  Exemple<String> valS = new Exemple<String>("TOTOTOTO");
  Exemple<Float> valF = new Exemple<Float>(12.2f);
  Exemple<Double> valD = new Exemple<Double>(12.202568); }
```

**Remarque**

Les types de données qu'on a employés pour déclarer des variables de type primitif !

sont les classes de ces types primitifs.

En effet, lorsqu'on déclare une variable de type primitif, on peut utiliser ses classes enveloppes (on parle aussi de classe wrapper) ; elles ajoutent les méthodes de la classe

Object aux types primitifs que vous utilisez ainsi que des méthodes permettant de caster leurs attributs, etc. À ceci, on peut dire 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.

**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';
}
```

### 1.2 Généricité multiple d'une classe

La généricité d'une classe peut être multiple. Nous avons créé une classe Généricité_Multiple qui prend deux paramètres génériques. Ci-dessous le code source de cette classe :

```
class Genericité_Multiple<T, S> {
  //Variable d'instance de type T
  private T attribut1;
  //Variable d'instance de type S
  private S attribut2;

  //Constructeur avec paramètres
  public Genericité_Multiple(T val1, S val2){
    this.attribut1 = val1;
    this.attribut2 = val2;
  }
  //Méthodes d'initialisation des deux attributs
```

```
  public void setAttribut(T val1, S val2){
    this.attribut1 = val1;
    this.attribut2 = val2;
  }
  //Retourne l'attribut T
  public T getAttribut1() {
    return attribut1;
  }
  //Définit l'attribut T
  public void setAttribut1(T attribut1) {
    this.attribut1 = attribut1;
  }
  //Retourne la attribut S
  public S getAttribut2() {
    return attribut2;  }
  //Définit la attribut S
  public void setAttribut2(S attribut2) {
    this.attribut2 = attribut2;
  } }
public class Test {
public static void main(String[] args) {
  Genericité_Multiple<String, Boolean> ex1 = new Genericité_Multiple<String,
Boolean>("toto", true);
  System.out.println("Attribut de l'objet ex1 : val1 = " + ex1.getAttribut1() + ", val2 =
" + ex1.getAttribut2());

  Genericité_Multiple<Double, Character> ex2 = new Genericité_Multiple<Double,
Character>(12.2585, 'C');
  System.out.println("Attribut de l'objet ex2 : val1 = " + ex2.getAttribut1() + ", val2 =
" + ex2.getAttribut2()); }}
```

Le résultat de l'exécution de ce code est le suivant :

```
Attribut de l'objet ex1 : val1 = toto, val2 = true
Attribut de l'objet ex2 : val1 = 12.2585, val2 = C
```

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.

### 1.3 Les limites des classes génériques

- **On ne peut pas instancier un objet d'un type paramétré**

```
class Exemple <T>
{ T attribut;
   Exemple(T val)
```

```
{ this.attribut=val;
}
void f( )
{  attribut=new T(); //interdit d'instancier un objet de type paramétré
}}
```

L'appel new T() est rejeté en compilation. En effet, au moment de l'exécution, le type T aura disparu (il aura été remplacé par Object).
La machine virtuelle n'a donc plus aucun moyen de connaître le type exact de l'objet à instancier.

- Il n'est pas possible de créer une classe générique dérivée de Throwable, donc aussi de Exception ou de Error

```
class Exemple <T> extends Exception // erreur de compilation
{ ....
}
```

- Ni de lever une exception (throw) à l'aide d'un objet d'une classe générique : throw

```
class Test{
   void f() throws Exemple <Integer> // erreur de compilation
{ ……..
 }
}
```

- Ni d'intercepter une exception en se basant sur un objet d'une classe Générique

```
catch (Exemple<Integer> e) {……..}
```

- Un champ statique ne peut pas être d'un type paramétré

```
class Exemple <T>
{ static T attribut; //erreur de compilation
}
```

## II. Méthode générique

Comme une classe ou une interface, une méthode peut être paramétrée par un ou plusieurs types. Une méthode générique peut être incluse dans une classe non générique, ou dans une classe générique (si elle utilise un paramètre autre que les paramètres de type formels de la classe).

**Exemple**

```
class Test_Méthode_Genrique{

public static < E > void afficher_element(E element)
{ System.out.println(element );
}
public static void main( String args[] ) {
   int element_Entier = 5;
   Character element_caractère = 'j';
System.out.print( "l'élement entier est :" );
afficher_element( element_Entier);
System.out.print( "L'élément caractère est : " );
afficher_element( element_caractère );
```

A l'exécution on aura :

```
l'élement entier est :5
L'élément caractère est : j
```

## III. Héritage et programmation générique

Pour faire l'héritage avec les classes génériques il y a plusieurs façons :

- La classe dérivée conserve les paramètres de type de la classe de base, sans en ajouter d'autres, comme dans :

```
class C <T> { …}
class D <T> extends C <T> {…..  }
Exemple
Ici, C et D utilisent le même paramètre de type.
Ainsi D<String> dérive de C<String>,
D<Double> dérive de C<Double>.
```

Il en irait de même avec :

```
class D<T, U> extends C<T, U>
{……..
}
Exemple :
 D<String, Double> dérive de C<String, Double>.
```

- La classe dérivée utilise les mêmes paramètres de type que la classe de base, en en ajoutant de nouveaux, comme dans :

class D <T, U> extends C <T> {

}

**Exemple** :
D<String, Double> dérive de C<String>,
D<Integer, Double> dérive de C<Integer>.

• La classe dérivée introduit des limitations sur un ou plusieurs des paramètres de type de la classe de base, comme dans :

class D <T extends Number> extends C<T>

Ici, on peut utiliser D<Double>, qui dérive alors de C<Double> ;

en revanche, on ne peut pas utiliser D<String> (alors qu'on peut utiliser C<String>).

• La classe de base n'est pas générique, la classe dérivée l'est.

```
class X {…}
class D<T> extends X {…}
```

D<String>, D<Double>, D<Point> dérivent toutes de X.

• En revanche, ces situations seront incorrectes :

```
class D extends C<T> // erreur : D doit disposer au moins du paramètre T
class G<T> extends C<T extends Number> // erreur
```

# IV. Le type joker

Les concepteurs du JDK 5.0 ont proposé une démarche qui permet l'emploi d'un *joker* qui peut être simple ou avec des limitations.

## 4.1 Le concept de joker simple

Avec notre classe *Exemple*<*T*> précédente, nous pouvons bien entendu définir les objets suivants :

```
Exemple<Integer> ei ;
Exemple<Double> ed ;
Exemple <?> eq  // eq désigne un objet de la classe exemple d'un type quelconque
Exemple <Object> eo ;
```

Mais, observons ces deux affectations :

```
eq=ed;   // OK : affectation d'un Exemple<Double> à un Exemple<?>
eo=ed;   // erreur de compilation incompatible type
         // Exemple<Double> ne peut pas être converti en Exemple<Object>
eq.setAttribut(? att1); // erreur de compilation
                        //Il n'est pas possible de modifier l'objet référence par eq
```

En conclusion on peut dire qu'il n'est pas possible **l'appel de toute méthode recevant un argument du type correspondant à ?**.

### 4.2 Le concept de Joker avec limitations

On peut imposer des limitations à un joker, comme on le fait pour des paramètres de type. Ainsi, avec notre classe *Exemple* précédente nous pouvons définir :

```
Exemple <Object> eo ;
Exemple <Integer> ei ;
Exemple <? extends Number> eqn ;
// ? représente un type quelconque dérivé de Number
```

Observons ces affectations :

```
L'affectation suivante sera illégale :
eqn=eo;          // erreur de compilation : Object ne dérive pas de Number
tandis que celle-ci sera légale :
eqn=ei;          // OK : Integer dérive bien de Number
```

Bien entendu, là encore, il ne sera pas possible d'appeler, à partir de *eqn*, une méthode modifiant l'objet référencé.

D'autres exemples :

```
Exemple <? super Pointcol>eqn ;
//? représente une classe quelconque ascendante de Pointcol (ou Pointcol elle même)
Point p ;
Exemple <Point> ep ;
Exemple <Integer> ei ;
l'affectation suivante sera rejetée :
eqn =ei ;  // erreur de compilation : Number n'est pas classe ascendante de Pointcol
tandis que celle-ci sera acceptée :
eqn=ep;  // OK : Point est classe ascendante de Pointcol
```

### 4.3Le type joker appliqué à une méthode :

Nous avons étudié le concept de joker dans des déclarations d'objets génériques. Il peut également s'appliquer à des arguments d'une méthode.

Supposez qu'on souhaite réaliser une méthode nommée **afficherForme** effectuant l'affichage des éléments ArrayList fourni en argument. Il est tout à fait possible d'exploiter les jokers en écrivant son en-tête de cette manière :
public static void **afficherForme**(ArrayList<? extends Forme> m)

Ci-dessous l'exemple complet :

```
import java.util.ArrayList;
abstract class Forme{
abstract void afficher();
}
class Rectangle extends Forme{
@Override
void afficher(){System.out.println("Rectangle");}
}
class Cercle extends Forme{
@Override
void afficher(){System.out.println("Cercle");}
}
class Test{
 public static void afficherForme(ArrayList<? extends Forme> m){
 for( Forme element: m)
    element.afficher();
}
   public static void main(String args[]){
      Rectangle r1=new Rectangle();
      Rectangle r2=new Rectangle();
      ArrayList<Rectangle> Ar1= new ArrayList<Rectangle>();
      ArrayList<Cercle> Ar2=new ArrayList<Cercle>();
      Ar1.add(r1);
      Ar1.add(r2);
      Ar2.add(new Cercle());
      Ar2.add(new Cercle());
      afficherForme(Ar1);
      afficherForme(Ar2);
}}
```

**On aura à l'exécution :**

```
Rectangle
Rectangle
Cercle
Cercle
```