

R3.02 : Développement efficace

Types génériques

J-F. Kamp

Septembre 2025

Introduction : sans le type générique

```
ArrayList myIntList = new ArrayList();  
  
myIntList.add ( new Integer(42) );  
  
Integer x = (Integer) myIntList.get ( 0 );
```

Transtypage obligatoire (**Integer**)
même si on sait que ce sont des
Integer.

Mais on aurait pu mettre un autre
transtypage (**AutreChose**) qui
provoque une erreur à l'exécution !!

Et si on exprimait clairement le type
du contenu ?

Apport de la classe générique

```
ArrayList<Integer> myIntList = new ArrayList<Integer>();  
  
myIntList.add ( new Integer(42) );  
  
Integer x = myIntList.get ( 0 );
```

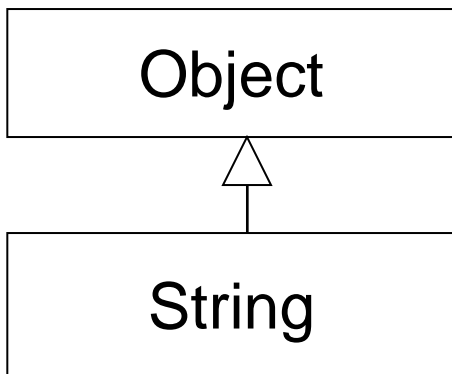
- La déclaration spécifie clairement le type du contenu
- Il n'y a plus besoin de transtypage
(Integer)
- Le compilateur peut vérifier la cohérence à tout moment

Classe ListeChaine<E> générique

```
public interface Liste<T> {  
  
    public void inserer ( T data ) ;  
    public T obtenirValeur () ;  
    ...  
}  
  
public class ListeChaine<E> implements Liste<E> {  
  
    private Element sentinel;  
  
    public ListeChaine () {...}  
  
    public void inserer ( E data ) {...}  
  
    public E obtenirValeur() {...}  
  
    private class Element {  
  
        Element suiv, prec;  
        E data;  
  
        Element ( Element prec, Element suiv, E data )  
    }  
}
```

Le type générique et le sous-typage

Règle de base : un sous-type peut toujours être affecté à un super-type.



```
Object b = new Object();  
String s = new String();
```

```
b = s; // tjrs possible !  
s = b; // NON
```

Questions

```
ArrayList <Object> lo = new ArrayList<>();
```

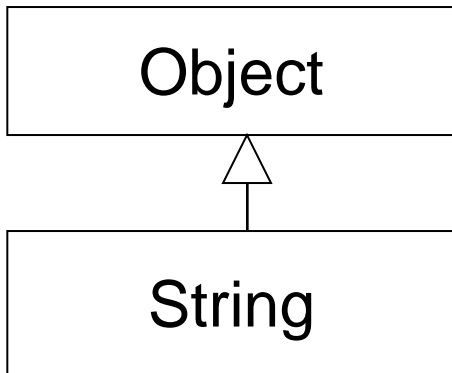
```
lo.add ( new String() );           // OK sans pb
```

```
Object data = lo.get(0);           // OK sans pb
```

```
String st = (String) lo.get(0);    // OK MAIS transtypage
```

Le type générique et le sous-typage

Règle de base : un sous-type peut toujours être affecté à un super-type.



```
Object b = new Object();
String s = new String();
```

```
b = s; // tjrs possible !
s = b; // NON
```

Question

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

```
ArrayList <Object> lo = ls; // possible ?
```

Instinctivement oui car une liste de *String* est une liste d'*Object*.

Le type générique et le sous-typage

Question1 : affectation

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

```
ArrayList <Object> lo = ls; // (1) possible ?
```

Instinctivement oui car une liste de *String* est une liste d'*Object*.

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

```
ArrayList <Object> lo = ls; // possible ?
```

```
// si oui alors
```

```
lo.add ( new Object() );
```

```
String s = ls.get ( 0 ); // (2) impossible !!
```

En (2) : impossible d'affecter un *Object* dans un *String*. Une liste de *String* **n'est donc pas** une liste d'*Object*. Donc en (1) ce n'est pas possible et ça **ne compile pas**.

Le type générique et le sous-typage

Question2 : le passage de paramètre entre sous-type et type

```
void printCollection ( ArrayList<Object> c ) {  
    Iterator<Object> i = c.iterator();  
  
    while ( i.hasNext() ) {  
        Object tmp = i.next();  
        System.out.println ( tmp );  
    }  
}  
  
// Peut-on faire...  
ArrayList <String> ls = new ArrayList<>();  
printCollection ( ls ); // NON pour les mêmes raisons
```

Conclusion : à ce stade une classe paramétrée *MaClasse<UnType>* n'accepte comme type *<UnType>* et rien d'autre : ni un sous-type et encore moins un super-type.

Le type Wildcard

Java 5 introduit un nouveau symbole
? qui veut dire : type quelconque
(*wildcard type*).

```
void printCollection ( ArrayList<?> c ) {
```

```
    Iterator<?> i = c.iterator();
```

```
    while ( i.hasNext() ) {
```

```
        Object tmp = i.next();
```

```
        System.out.println ( tmp );
```

```
    }
```

```
}
```

```
// Et maintenant ça compile
```

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

```
printCollection ( ls ); // OUI
```

Le type Wildcard

Attention, *wildcard* ? n'est pas équivalent à **Object**. **<?>** veut juste dire « un type » mais pas forcément *Object*.

// Exemple

```
ArrayList <?> c = new ArrayList<String>();  
c.add ( new Object() );    // NON, ne compile pas  
c.add ( new String() );    // NON plus... et pourtant...
```

En effet, on ne sait pas quel est le type de la collection. Ici ajouter un *Object* dans une collection de *String* n'est pas possible.

Le type Wildcard

```
// Par contre...  
ArrayList <String> ls = new ArrayList<>();  
ls.add ( new String("OK") );  
ArrayList <?> c = ls;  
Object o = c.get(0); // OUI !
```

En effet, le contenu d'une *ArrayList<?>* est forcément un objet donc il est toujours possible de le récupérer dans une variable de type *Object*.

Conclusion : une classe paramétrée avec ?

- peut accepter n'importe quelle type
- peut être utilisée uniquement en consultation (pas en modification)

Le type Wildcard borné

// Soit le code suivant...

```
public abstract class Shape {  
    public abstract void draw ( Canvas c );  
}  
  
public class Circle extends Shape {  
    private int x, y, radius;  
    public void draw ( Canvas c ) { ... }  
}  
  
public class Rectangle extends Shape {  
    private int x, y, width, height;  
    public void draw ( Canvas c ) { ... }  
}
```

Le type Wildcard borné

```
// Ces formes géométriques peuvent être dessinées  
// sur un Canvas
```

```
public class Canvas {  
  
    ...  
  
    // Un dessin contient un ensemble de figures  
    // stockées par exemple dans une liste.  
    public void drawAll ( List<Shape> s ) {  
  
        Iterator<Shape> i = s.iterator();  
  
        while ( i.hasNext() ) {  
            Shape tmp = i.next();  
            tmp.draw(this);  
        }  
    }  
}
```

Ce n'est pas aussi générique qu'on le voudrait puisque passer un paramètre **List<Rectangle>** à **drawAll(...)** n'est pas possible.

Le type Wildcard borné

Solution : le type *wildcard* borné.

Notation : **<? extends Truc>**

```
// On peut maintenant dessiner Rectangle ou Circle
// du moment que c'est sous-type de Shape
// List<? extends Shape> s
```

```
public class Canvas {
```

```
    ...
```

```
    // Un dessin contient un ensemble de figures,
    // stockées par exemple dans une liste.
```

```
    public void drawAll ( List<? extends Shape> s ) {
```

```
        Iterator<Shape> i = s.iterator();
```

```
        while ( i.hasNext() ) {
            Shape tmp = i.next();
            tmp.draw(this);
```

```
        }
```

```
    }
```

```
}
```

Le type Wildcard borné

Et donc on peut écrire (compilation et exécution OK)

```
Canvas myC = new Canvas();
```

```
ArrayList <Rectangle> listR = new ArrayList<>();
```

```
Rectangle r1 = new Rectangle (...);
```

```
Rectangle r2 = new Rectangle (...);
```

```
listR.add ( r1 );
```

```
listR.add ( r2 );
```

```
myC.drawAll ( listR ); // ici ça compile
```

Le type Wildcard borné

Tout n'est pas encore faisable...

```
void someMethod ( List<? extends Shape> s ) {  
    ....  
    s.add ( new Rectangle() ); // NON !  
}
```

Pourquoi ce n'est pas possible...

Le type *wildcard* ? doit être sous-type de *Shape* MAIS rien ne dit que le type *Rectangle* est sous-type de *<?>*.