Programmation orientée objets

Leçon 10 : Généricité : approfondissement

Prof. Rolf Ingold

Département d'informatique

UNI
FR

UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Contenu de la leçon

- Rappel sur la généricité
- Lien avec l'héritage
- Types indéfini
- Méthodes génériques
- Méthodes génériques avec un type contraint
- Contraintes de type par le haut
- Contraintes de type par le bas
- Hiérarchie des types contraints
- Restrictions liées à la généricité

Rappel : concept de classe générique

- Une **classe générique** est définie au moyen de la syntaxe

```
class MyClass<T,...>  ...
```

où `T,...` sont des **types formels** qui représentent des types non spécifiés, mais utilisés à l'intérieur de `MyClass`

- Lors de l'instanciation d'une classe générique, on substitue aux **types formels** des **types effectifs** (classes ou interfaces)

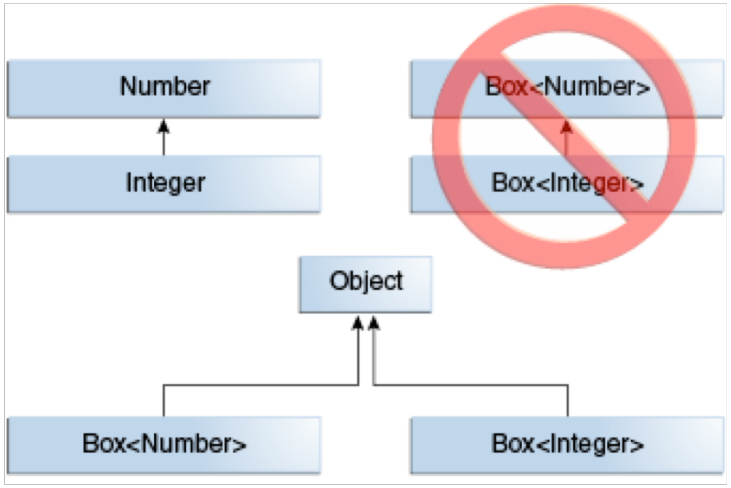
```
Box<Cigar> myCigarBox = new Box<Cigar>();
```

  - si le type effectif peut être déduit du contexte, il n'a pas besoin d'être précisé (utilisation du "diamant" vide <>)

```
Box<Jewel> myJewelBox = new Box<>();
```

Rappel : pas d'héritage induit

- Par contre, **attention** ! : l'héritage des types effectifs ne se répercute pas sur les types génériques instanciés par eux



- Exemple : soit la déclaration déclaration de méthode

```
public static void doSomething(Box<Number> b) { ... }
```

il n'est pas possible de l'invoquer de la manière suivante

```
Box<Integer> intBox = new Box<>();
doSomething(intBox); // COMPILATION ERROR !
```


UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Instanciation de type avec la classe `Object`

- Etant donné les définitions suivantes

```
public static void printList(List<Object> list) {
    for (Object elem : list) {
        System.out.print(elem + " ");
    }
    System.out.println();
}
List<String> strList = Arrays.asList("one", "two", "three");
List<Integer> intList = Arrays.asList(1, 2, 3);
```

peut-on utiliser les invocations suivantes ?

```
printList(strList);
printList(intList);
```

  - NON ! car `List<String>` et `List<Integer>` ne sont pas des sous-classes de `List<Object>`

- Pour y remédier, à la place de **`Object`** un peu utiliser un type indéfini

Type indéfini

- Un **type indéfini** est un type non connu au moment de la compilation

- Il est représenté par le signe **?** (aussi appelé **joker**)

```
public static void printList(List<?> list) {
    for (Object elem : list) {
        System.out.print(elem + " ");
    }
    System.out.println();
}
```

et ainsi il devient possible d'invoquer

```
List<String> strList = Arrays.asList("one", "two", "three");
List<Integer> intList = Arrays.asList(1, 2, 3);

printList(strList);
printList(intList);
```

Méthodes opérant sur des types indéfinis

- Considérons une méthode statique qui a pour tâche de copier les éléments d'un tableau dans une liste générique
  - attention : la méthode suivante est erronée

```
static void copyArrayToList(Object[] array, List<?> list) {
    for (Object obj : array) {
        list.add(obj); // error
    }
}
```

  - car le paramètre effectif utilisé pourrait être d'un type plus restrictif que `List<Object>`

Méthodes génériques

- En lieu et place, il faut utiliser une **méthode générique**, i.e. une méthode paramétrée par un type
  - le type formel **<T>** est placé avant le type du résultat
  - il peut ensuite être utilisé comme type de paramètres formels

```
static <T> void copyArrayToList(T[] array, List<T> list) {
    for (T obj : array) {
        list.add(obj); // correct
    }
}
```

Méthode génériques avec plusieurs types formels

- Une méthode générique peut avoir plusieurs types formels
  - syntaxiquement, les types formels sont indiqués entre <> et précèdent le type du résultat de la méthode

```
public static <K,V> boolean identical(Pair<K,V> p1,
                                      Pair<K,V> p2) {
    return p1.getKey().equals(p2.getKey())
        && p1.getValue().equals(p2.getValue());
}
```

Invocation de méthodes génériques

- Une méthode générique peut être invoquée en précisant les paramètres effectifs devant le nom de la méthode

```
Pair<Integer,String> first = new Pair<>(1, "apple");
Pair<Integer,String> second = new Pair<>(2, "pear");
boolean same = Util.<Integer,String>identical(first, second);
```

- Ce n'est toutefois pas obligatoire, car le contexte permet au compilateur de connaître ces types (et c'est toujours le cas)

```
boolean same = Util.identical(first, second);
```

Méthode générique avec un type contraint

- Considérons une méthode statique qui compte les éléments d'un tableau qui sont supérieurs à un seuil donné
  - le programme ci-dessous n'est pas compilable

```
public static <T> int nbGreater(T[] array, T threshold) {
    int count = 0;
    for (T e : array) {
        if (e > threshold) { // compiler error !!!
            ++count;
        }
    }
    return count;
}
```

  - en effet, l'opérateur `>` n'est pas défini pour tous les types

- Il faut donc contraindre l'utilisation avec un type effectif qui permet la comparaison

Méthode générique avec un type contraint

- La méthode **`nbGreater`** n'est applicable qu'à des types comparables
  - en se basant sur l'interface

```
public interface Comparable<T> {
    public int compareTo(T other);  // returns -1 | 0 | 1
}
```

  - on peut définir la méthode de comptage comme suit

```
public static <T extends Comparable<T>>
              int nbGreater (T[] array, T threshold) {
    int count = 0;
    for (T elt : array) {
        if (elt.compareTo(threshold) > 0) {
            count++;
        }
    }
    return count;
}
```

Type indéfinis avec contraintes

- Ce type de contraintes peut aussi s'appliquer à un type indéfini

- On distingue les contraintes
  - **par le haut** : être une sous-classe d'une classe donnée (y compris elle-même)
  - **par le bas** : être une superclasse d'une classe donnée, (y compris elle-même)

Contrainte d'un type indéfini par le haut

- Pour définir un type contraint par le haut, on utilise la syntaxe
  `<? extends type>`

- Exemple : étant donné la définition d'une fonction qui établit la somme des éléments d'une liste

```
public static double sumOfList(List<? extends Number> list) {
    double sum = 0.0;
    for (Number nb : list) {
        sum += nb.doubleValue();
    }
    return sum;
}
```

on peut l'utiliser sur différents types de listes (utilisant différents types de nombres)

```
List<Integer> intList = Arrays.asList(1, 2, 3);
System.out.println(sumOfList(intList));
List<Double> dblList = Arrays.asList(1.2, 2.3, 3.5);
System.out.println(sumOfList(dblList));
```

Contrainte d'un type indéfini par le bas

- Pour définir un type contraint par le bas, on utilise la syntaxe
  `<? super type>`

- Exemple : une procédure qui ajoute les nombres de 1 à 10 à une liste quelconque, i.e qui accepte des entiers parmi d'autres objets

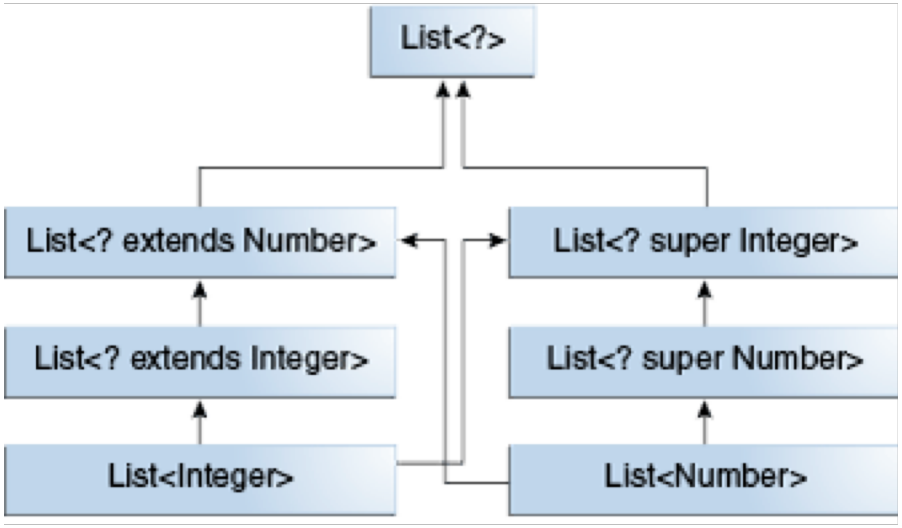
```
public static void addNumbers(List<? super Integer> list) {
    for (int i = 1; i <= 10; i++) {
        list.add(i);
    }
}
```

- peut être utilisée dans une liste d'objets quelconques

```
List<Object> objList = new ArrayList<>();
addNumbers(objList);
```

Hiérarchie des types contraints

- En conclusion, il existe bien des liens d'héritage entre types génériques
  - exemple d'héritage pour des listes génériques de nombres



Restrictions liées à la généricité

- Les types formels d'une classe générique ne peuvent pas
  - être instanciés par des types primitifs
  - être utilisés comme type de variables statiques
  - faire l'objet d'une instanciation d'une autre classe générique

- Les types génériques ne peuvent pas
  - être utilisés dans une conversion de type (type cast)
  - être utilisés avec `instanceof`
  - être utilisés comme type de base d'un tableau
  - être utilisés pour les exceptions (sous-type de `Throwable`)
  - se distinguer par l'instanciation, pour la surcharge