Généricité

Marc Champesme
mailto:Marc.Champesme@lipn.univ-paris13.fr

13 janvier 2021

La généricité pourquoi faire ?

- Améliorer la sûreté du logiciel (éviter des erreurs logiciels)
- Améliorer la lisibilité du code.

Dans quels langages ?

- EIFFEL
- C++ (templates)
- JAVA depuis la version 1.5 : l'innovation la plus importante depuis la création du langage JAVA

Utiliser la généricité

Sans la généricité

Définition de classe

```
public class ArrayList
extends AbstractList { ...
}
```

Création d'instance

```
new ArrayList()
```

Avec la généricité

Définition de classe générique

```
public class ArrayList<E>
extends AbstractList<E> {
... }
```

Création d'instance

```
new ArrayList<String>()
```

Utiliser la généricité

Sans la généricité

Polymorphisme

```
List uneListe = new ArrayList();
uneListe.add("aba");
uneListe.add(new Rectangle(...));
```

Avec la généricité

Polymorphisme

```
List<String> uneListe = new ArrayList<String>();
uneListe.add("aba");
```

Erreur de compilation !

```
uneListe.add(new Rectangle(...));
```

Utiliser la généricité

Sans la généricité

Cast obligatoire

```
List uneListe = new ArrayList();
uneListe.add("aba");
uneListe.add(new Rectangle(...));
String s = (String) uneListe.get(0);
```

Erreur à l'exécution !

```
 String s = (String) uneListe.get(1);
```

(`uneListe.get(1);` est un `Rectangle`)

Utiliser la généricité

Avec la généricité

Fini les cast obligatoires

```
List<String> listStr = new ArrayList<String>();
listStr.add("aba");
String s = listStr.get(0); // cast inutile
```

Le compilateur garantie que `listStr.get(0)` contient une instance de `String` (ou `null`)

Généricité et héritage

Le code suivant est il légal ?

```
List<String> listStr = new ArrayList<String>();
List<Object> listObj = listStr;
```

Supposons qu'il le soit, on pourrait alors écrire :

```
listObj.add(new Object());
String s = listStr.get(0);    // Tentative d'affectation d'un
                              // Object a une variable
                              // de type String
```

Pour éviter cette erreur à l'exécution :

Cette instruction est illégale (erreur de compilation)

```
List<Object> listObj = listStr;
```

Relation entre généricité et héritage

Attention ! ! !

Si `TypeInf` est un sous-type de `TypeSup`, et que `GenType` est un type générique, alors IL EST FAUX de dire que :

`GenType<TypeInf>` est un sous-type de `GenType<TypeSup>`

ou encore :

Attention ! ! !

Quelques soient deux types distincts `Type1` et `Type2` (et quelques soient les relations d'héritage que ces deux types pourraient avoir), alors :

il n'existe AUCUNE relation d'héritage entre
`GenType<Type1>` et `GenType<Type2>`

Relation entre généricité et héritage

Exemple

`List<String>` n'est pas un sous-type de `List<Object>`

Par contre

Si `GenTypeInf<E>` est un sous-type de `GenTypeSup<E>` et que `TypeQuelc` est un type quelconque, alors :

`GenTypeInf<TypeQuelc>`
est un sous-type de
`GenTypeSup<TypeQuelc>`

Exemple

`ArrayList<String>` est un sous-type de `List<String>`

Conséquence

Soit la méthode :

```
void printCollection(Collection<Object> c) {
    for (Object e : c) {
        System.out.println(e);
    }
}
```

Alors :

```
List<Object> listObj = ...;
Collection<String> collStr = ...;
a.printCollection(listObj);    // Légal
a.printCollection(collStr);    // Illegal
```

Conséquence

Pourtant :

```
Collection<String> collStr = ...;
for (Object e : collStr) {
    System.out.println(e);
}
```

... est correct,

alors comment faire pour que `a.printCollection(collStr)` soit légal ?

Table des matières

Types joker

La solution consiste à utiliser un type joker :

```
void printCollection(Collection<?> c) {
    for (Object e : c) {
        System.out.println(e);
    }
}
```

Alors :

```
List<Object> listObj = ...;
Collection<String> collStr = ...;
a.printCollection(listObj);    // Légal
a.printCollection(collStr);    // Légal
```

Limites du type Joker

Mais, heureusement, l'utilisation d'un type Joker est soumise à certaines restrictions :

```
Collection<?> c = new ArrayList<String>();
c.add(new Object());     // Illégal (heureusement)!
c.add("porte");          // Illégal aussi!
```

Attention ! ! !

`Collection<?>` n'est pas une collection de n'importe quoi !
C'est une collection d'éléments d'UN type que l'on ne connait pas.

Table des matières

Bienvenue au Zoo !

Supposons maintenant la hiérarchie de classes suivante :

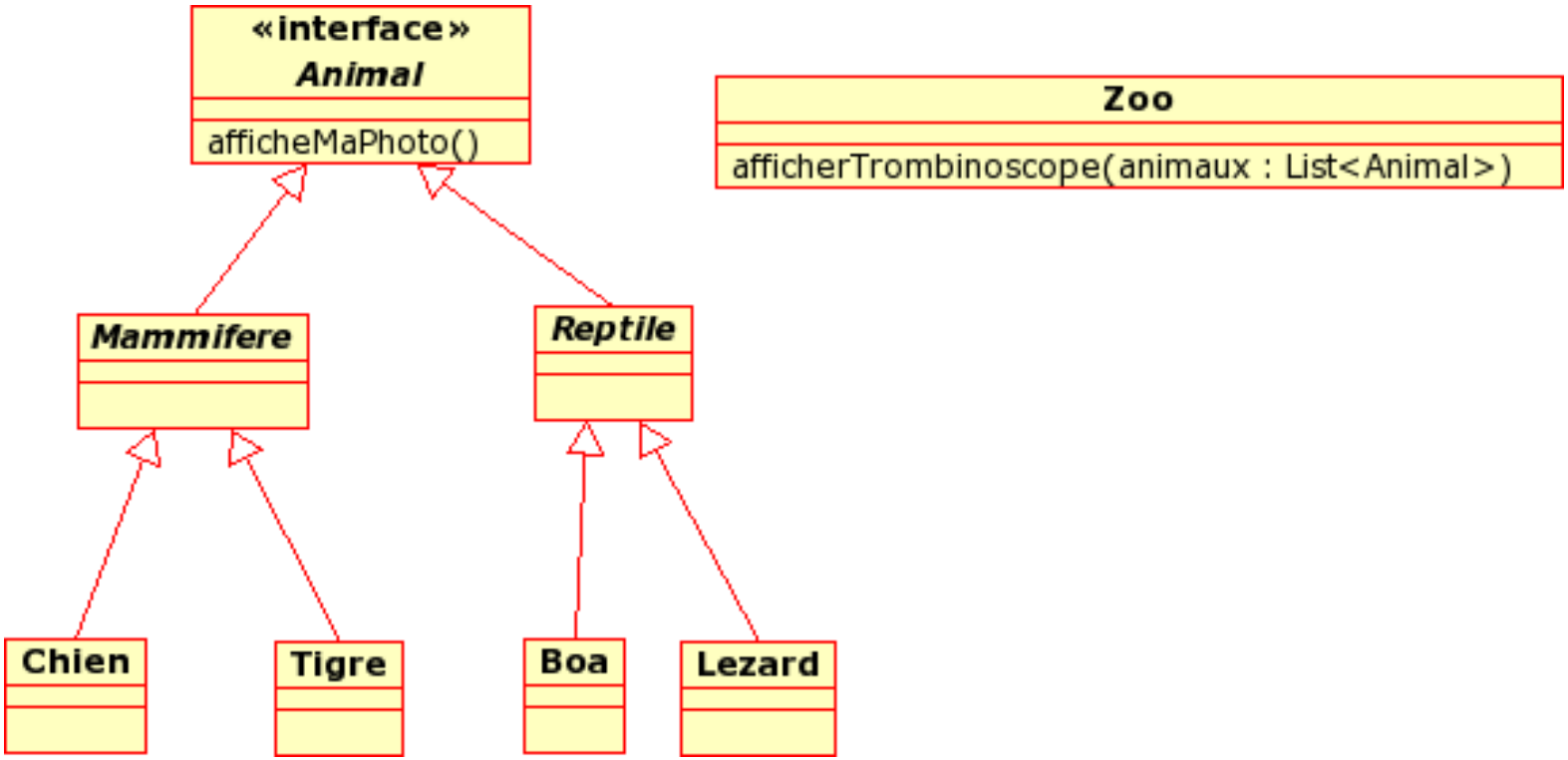

FIGURE – Hiérarchie de classes pour le Zoo.

Trombinoscope

Le code de la méthode afficherTrombinoscope pourrait être par exemple :

```
void afficherTrombinoscope(List<Animal> animaux) {
    for (Animal a : animaux) {
        a.afficheMaPhoto();
    }
}
```

Mais comme nous l'avons vu précédemment :

```
Zoo unZoo = ...
List<Chien> listChien = ...
unZoo.afficherTrombinoscope(listChien); // Illégal!
```

Première tentative de solution

L'utilisation du type Joker "?" n'arrange rien :

```
void afficherTrombinoscope(List<?> animaux) {
    for (Animal a : animaux) {  // Illégal !
        a.afficheMaPhoto();
    }
}
```

De même :

```
void afficherTrombinoscope(List<?> animaux) {
    for (Object a : animaux) {  // Légal
        a.afficheMaPhoto();     // Illégal
    }
}
```

Première tentative de solution

Ce qu'il nous faudrait :

Nous ne souhaitons pas passer en argument une liste de n'importe quel type inconnu, mais une liste d'éléments de type inconnu mais dont le super-type est `Animal` car seuls les animaux possèdent la méthode `afficheMaPhoto()`.

La solution : un type Joker borné

Spécifions que le type inconnu doit être sous-type de `Animal` :

```
void afficherTrombinoscope(List<? extends Animal> animaux) {
    for (Animal a : animaux) {  // Légal !
        a.afficheMaPhoto();
    }
}
```

Et maintenant :

```
Zoo unZoo = ...
List<Chien> listChien = ...
unZoo.afficherTrombinoscope(listChien); // Légal!
                                        // car Chien hérite de Animal
```

Un autre problème

Supposons que nous voulions définir la méthode suivante :

```
void ajouterUnLezard(List<? extends Animal> animaux) {
    animaux.add(new Lezard(...)); // Illégal
}
```

- Heureusement, sinon nous pourrions ajouter un lézard à une liste de chiens !
- Ce qu'il nous faut ici c'est pouvoir spécifier que la liste doit pouvoir accepter les lézards, c'est à dire que le type des éléments de la liste doit être un super-type de `Lezard` (comme `Reptile`, `Animal` ou même `Object`).

Et la solution...

Spécifions que le type inconnu doit être super-type de `Lezard` :

```
void ajouterUnLezard(List<? super Lezard> animaux) {
    animaux.add(new Lezard(...)); // Légal
}
```

Et maintenant :

```
Zoo unZoo = ...
List<Chien> listChien = ...
List<Reptile> listReptile = ...
List<Object> listObj = ...
unZoo.ajouterUnLezard(listChien); // Illégal
unZoo.ajouterUnLezard(listReptile); // Légal
unZoo.ajouterUnLezard(listObj); // Légal
```