Généricité avancée, gestion mémoire

Pratique de la programmation orientée-objet
Michel Schinz – 2014-05-19

Sous-typage

Sous-typage

Rappel : en Java, les classes, les interfaces et les énumérations définissent des types. Ces types sont liés entre-eux par une relation de sous-typage.
Par exemple, le type `String` est un sous-type du type `Object` car la classe `String` hérite de la classe `Object`.
De manière similaire, le type `Number` est un sous-type du type `Serializable` car la classe `Number` implémente l'interface `Serializable`.
Lorsqu'un type T_2 est sous-type d'un type T_1, on dit que T_1 est un super-type de T_2.

Propriétés du sous-typage

Formellement, la relation de sous-typage est :

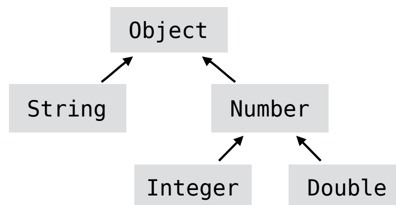
- **réflexive**, c-à-d que tout type est sous-type de lui-même,
- **transitive**, c-à-d que si un type T_1 est sous-type d'un type T_2 et T_2 est sous-type de T_3, alors T_1 est aussi sous-type de T_3,
- **anti-symétrique**, c-à-d que si T_1 est sous-type de T_2 et T_2 est sous-type de T_1, alors $T_1 = T_2$.

En mathématiques, une relation possédant ces trois propriétés est appelée **ordre partiel**.

Graphe des types

La relation de sous-typage peut être visualisée sous la forme d'un graphe dirigé dans lequel chaque type est un nœud et un arc lie le nœud d'un type à celui de ses super-types directs.

Un (minuscule) extrait du graphe des types standard Java :



Q : comment les types `Integer` et `Object` sont-ils liés par cette relation de sous-typage ? Et `String` et `Number` ?

Polymorphisme d'inclusion

Le **polymorphisme d'inclusion** permet de substituer à une valeur d'un type T_1 donné une valeur d'un autre type T_2 pour peu que T_2 soit un sous-type de T_1. On appelle aussi cela le **principe de substitution** (*substitution principle*).

Par exemple, si une fonction prend en argument une valeur de type `Number`, en plus d'une valeur de ce type on peut lui passer une valeur de type `Integer`, `Double`, etc.

```
Number add(Number n1, Number n2) {
  return new Double(n1.doubleValue()
                    + n2.doubleValue());
}
add(new Integer(1), new Double(3.14));
```


Sous-typage et généricité

Listes génériques

Pour illustrer les concepts que nous allons examiner, nous réutiliserons notre interface des listes génériques :

```
public interface List<E> {
  boolean isEmpty();
  int size();
  void add(E newElem);
  void remove(int index);
  E get(int index);
  void set(int index, E elem);
  Iterator<E> iterator();
}
```

Liste de nombres

Le principe de substitution nous permet d'ajouter n'importe quel type de nombre – c-à-d n'importe quel sous-type de `Number` – à une liste de `Number` :

```
List<Number> l = new LinkedList<>();
Integer i = 1;
l.add(i);
Double d = 3.14;
l.add(d);
```

Chacun des deux appels à `add` est valide car `Integer` et `Double` sont des sous-types de `Number`.

Ajout groupé

Pour pouvoir plus facilement ajouter tous les éléments d'une liste à une liste existante, essayons de définir une méthode `addAll` dans `List`. Premier essai :

```
public interface List<E> {
  …
  void addAll(List<E> other);
}
public class LinkedList<E>
    implements List<E> {
  …
  void addAll(List<E> other) {
    for (E elem: other)
      add(elem);
  }
}
```


Sous-typage et généricité

Malheureusement, la méthode `addAll` que nous venons de définir n'est pas utilisable comme nous le désirerions :

```
List<Number> l = new LinkedList<>();
List<Integer> li = new LinkedList<>();
Integer i = 1;
li.add(i);
l.addAll(li); // refusé !
```


Généricité et sous-typage

Le code précédent est refusé car, en Java, une instanciation d'un type générique n'est *jamais* sous-type d'une autre instanciation de ce même type générique…

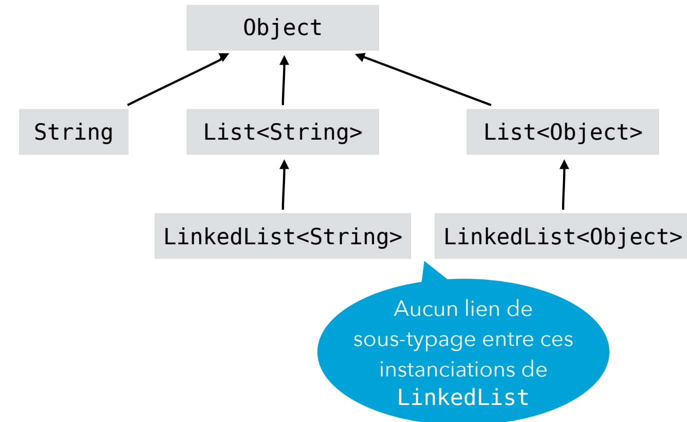
Par exemple, le type `List<`*U*`>` n'est jamais sous-type de `List<`*V*`>` sauf dans le cas trivial où *U*=*V*.

Le seul moyen de rendre l'appel à `addAll` valide est donc de changer le type de la seconde liste pour en faire une liste de `Number`. Cela n'est pas très satisfaisant, car il est clairement valide d'ajouter une liste d'entiers à une liste de nombres. Il nous faudra trouver une solution plus tard !

Généricité et sous-typage

Attention, la restriction mentionnée ne signifie pas que deux types génériques *différents* ne peuvent pas être liés par une relation de sous-typage !
Par exemple, `LinkedList<String>` est un sous-type de `List<String>` car la classe générique `LinkedList<E>` implémente l'interface `List<E>`.
Par contre, deux instanciations différentes du même type générique ne sont jamais liées entre-elles par la relation de sous-typage.

Généricité et sous-typage



Justification de la limitation

Pourquoi les concepteurs de Java ont-ils choisi d'imposer cette restriction sur la généricité ?
Pour le comprendre, admettons que `List<Integer>` soit un sous-type de `List<Number>`. Cela nous autoriserait à écrire le code suivant :

```
void addPi(List<Number> l) {
  l.add(Math.PI);
}
List<Integer> l = new LinkedList<>();
addPi(l);
```

Quel problème cela pose-t-il ?

`addAll` v2

Question : n'est-il pas possible de définir une méthode `addAll` qui soit plus générale que celle définie précédemment, et qui permette l'ajout – valide – d'une liste d'entiers à une liste de nombres ?

addAll v2

Réponse : oui, en la rendant générique et bornée !

```
interface List<E> {
  …
  <F extends E> void addAll(List<F> other);
}
```


addAll v2

Avec cette nouvelle définition :

```
interface List<E> {
  …
  <F extends E> void addAll(List<F> other);
}
```

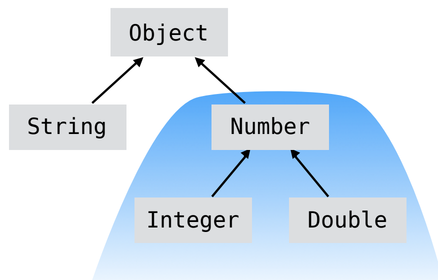
le code précédent est maintenant valide :

```
List<Number> l = new LinkedList<>();
List<Integer> li = new LinkedList<>();
Integer i = 1;
li.add(i);
l.addAll(li);
```

Question : quel type est inféré pour le paramètre `F` dans l'appel à `addAll` ?

Borne supérieure

La borne (supérieure) permet l'utilisation de n'importe quel *sous-type* de la borne, ici `Number`.



addAll v2

Bien entendu, changer le type dans l'interface `List` n'a de sens que si les mises en œuvre concrètes de la méthode restent valides. Pour `addAll`, c'est le cas :

```
public class LinkedList<E>
    implements List<E> {
  …
  <F extends E> void addAll(List<F> other){
    for (F elem: other)
      add(elem);
  }
}
```

Question : pourquoi ce code est-il valide ?

Jokers
(ou *wildcards*)

Jokers

Le paramètre de type F de la méthode `addAll` n'est pas utilisé ailleurs que dans son type. Il n'est donc pas nécessaire de le nommer, et Java permet d'utiliser dans ce cas un **joker** (*wildcard*) borné, noté **?** :

```
public interface List<E> {
  …
  void addAll(List<? extends E> other);
}
```

A noter qu'il est aussi possible d'utiliser un joker sans le borner explicitement, ce qui équivaut à le borner avec `Object`. Par exemple, `List<?>` est totalement équivalent à `List<? extends Object>`.

Ajout groupé inversé

Nous avons réussi à définir une méthode `addAll` satisfaisante. Essayons maintenant de définir une méthode `addAllInto` qui ajoute tous les éléments du récepteur dans la liste passée en argument. Premier essai :

```
public interface List<E> {
  …
  void addAllInto(List<E> other);
}
class LinkedList<E> implements List<E> {
  …
  void addAllInto(List<E> other) {
    other.addAll(this);
  }
}
```


Ajout groupé inversé

Bien entendu, cette première version possède les mêmes limitations que notre première version de la méthode `addAll`, à savoir que le code suivant est invalide :

```
List<Number> l = new LinkedList<>();
List<Integer> li = new LinkedList<>();
Integer i = 1;
li.add(i);
li.addAllInto(l); // refusé !
```

addAllInto v2

Nous pourrions bien entendu essayer de résoudre le problème de la même manière que pour `addAll`, c-à-d en utilisant un joker équipé d'une borne supérieure :

```
public interface List<E> {
  …
  void addAllInto(List<? extends E> other);
}
List<Number> l = new LinkedList<>();
List<Integer> li = new LinkedList<>();
Integer i = 1;
li.add(i);
li.addAllInto(l);
```

Question : pourquoi cela ne fonctionne-t-il pas ?

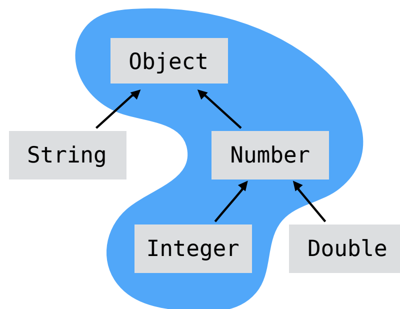
addAllInto v3

La borne de `addAllInto` doit être une borne *inférieure* et pas *supérieure* ! Heureusement, Java offre de telles bornes sur les jokers – mais pas sur les paramètres de type :

```
public interface List<E> {
  …
  void addAllInto(List<? super E> other);
}
List<Number> l = new LinkedList<>();
List<Integer> li = new LinkedList<>();
Integer i = 1;
li.add(i);
li.addAllInto(l);
```


Borne inférieure

La borne inférieure permet l'utilisation de n'importe quel *super-type* de la borne, ici `Integer`.



Ajout groupé bidirectionnel

Admettons pour terminer que l'on désire définir une méthode `addAllFromAndInto` qui ajoute tous les éléments de l'argument au récepteur et inversement :

```
public interface List<E> {
  …
  void addAllFromAndInto(List<E> other);
}
class LinkedList<E> implements List<E> {
  …
  void addAllFromAndInto(List<E> other) {
    this.addAll(other); other.addAll(this);
  }
}
```

Question : quel type de borne utiliser et pourquoi ?

Loi des bornes

La « loi des bornes » ci-dessous permet de se souvenir aisément de quelle genre de borne utiliser dans quelle situation :

- Lorsqu'on désire *uniquement lire* dans une structure, on utilise une borne *supérieure* (avec `extends`) ;
- lorsqu'on désire *uniquement écrire* dans une structure, on utilise une borne *inférieure* (avec `super`) ;
- lorsqu'on désire *à la fois lire et écrire* dans une structure, on n'utilise *aucune borne*.

(Par structure, on entend p.ex. une liste, etc.)

Principe PECS

En anglais, cette loi des bornes est parfois nommée ***PECS***, acronyme de *Producer Extends, Consumer Super*.
Cet acronyme permet de se souvenir facilement que :

- lorsque la structure que l'on utilise est un *producteur*, c-à-d qu'on y lit des valeurs, il faut utiliser `extends` pour borner son type, et
- lorsque la structure que l'on utilise est un consommateur, c-à-d qu'on y écrit des valeurs, il faut utiliser `super` pour borner son type.

Lorsqu'on désire à la fois lire et écrire, on ne peut borner son type.

Types bruts

Compatibilité

La généricité n'a été introduite que tardivement dans le langage Java, à un moment où beaucoup de code non-générique avait déjà été écrit.
Idéalement, tout ce code non-générique aurait dû être adapté du jour au lendemain, et la question de la compatibilité entre les deux formes de code ne se serait pas posée.
En pratique, cela n'est bien entendu pas possible, et les concepteurs de Java ont donc introduits des concepts facilitant la compatibilité entre le code générique et le code non-générique.
Nous ne considérerons ici que le cas du code non-générique utilisant du code générique.

Types bruts

Lorsque la généricité a été ajoutée à Java, sa bibliothèque standard a été modifiée pour en tirer parti. Par exemple, l'interface `List` a été transformée en `List<E>`, où `E` représente le type des éléments de la liste.

Rigoureusement, une fois cette modification faite, le type `List` (sans argument de type) est invalide et son utilisation devrait être refusée par le compilateur. Mais cela rendrait impossible la compilation de beaucoup d'ancien code...

Pour éviter ce problème, les concepteurs de Java ont introduit la notion de **type brut** (*raw type*), qui est simplement un type générique utilisé sans paramètres. Dans notre exemple, `List` est un tel type.

Types bruts

La version brute d'un type interagit avec la version générique de ce même type de la manière suivante :

- une version générique peut être utilisée partout où la version brute est attendue, sans provoquer l'affichage d'un avertissement,
- la version brute peut être utilisée partout où une version générique est attendue, mais cela provoque l'affichage d'un avertissement.

Par exemple, si on passe une valeur qui a le type brut `List` à une méthode qui attend une valeur de type `List<String>`, le code est accepté avec un avertissement.

Utilisation des types bruts

Règle : n'utilisez jamais les types bruts dans votre code, ils n'existent que pour faciliter la migration du code écrit avant l'introduction de la généricité.

Gestion mémoire

Gestion mémoire

Une caractéristique importante de la machine virtuelle Java est qu'elle gère automatiquement la mémoire.
Cela signifie que la mémoire allouée aux objets créés au moyen de l'opérateur `new` de Java ne doit – et ne peut – pas être libérée explicitement par le programmeur.
Au lieu de cela, la mémoire allouée aux objets qui sont devenus inatteignables – c-à-d plus référencés – est automatiquement libérée.
La partie de la machine virtuelle Java qui se charge de cette tâche est nommée le **ramasse-miettes** (*garbage collector* ou *GC*).

Ramasse-miettes

La boucle ci-dessous consomme vite la totalité de la mémoire disponible, et l'exécution s'arrête avec une exception `OutOfMemoryError` :

```
int[][] arrays = new int[1000][];
for (int i = 0; i < 1000; ++i) {
  arrays[i] = new int[1000000];
}
```

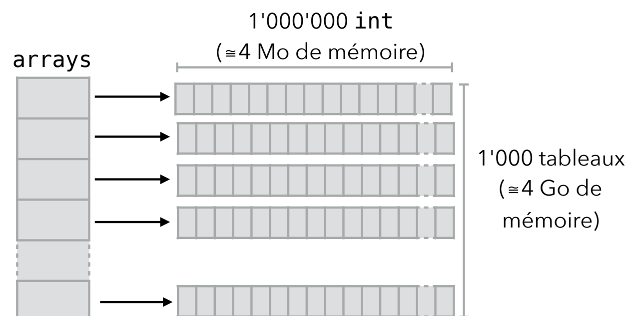
En ajoutant simplement la ligne suivante :

```
arrays[i] = null;
```

comme seconde ligne de la boucle, le programme s'exécute sans problème puisqu'aucune référence n'est gardée sur les tableaux alloués, qui peuvent donc être libérés par le ramasse-miettes.

Boucle d'allocation v1

```
int[][] arrays = new int[1000][];
for (int i = 0; i < 1000; ++i)
  arrays[i] = new int[1000000];
```



Ramasse-miettes

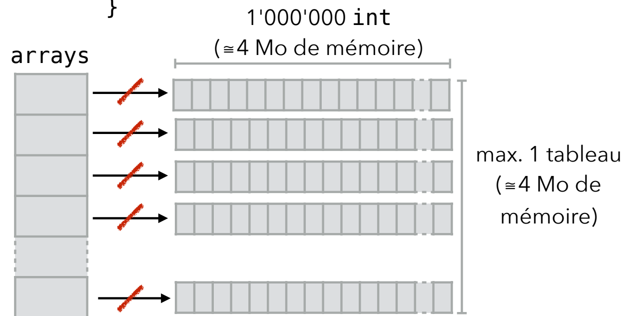
Le travail du ramasse-miettes peut être observé en passant l'option `-verbose:gc` à `java`. La première version de la boucle produit ceci :

```
[GC 16349K->16041K(69056K), 0,00653 secs]
[GC 32579K->31666K(87168K), 0,00819 secs]
[GC 67441K->66774K(107008K), 0,01315 secs]
…
[Full GC 984715K->984662K(991168K), …]
[Full GC 984662K->984648K(991168K), …]
Exception in thread "main"
java.lang.OutOfMemoryError: Java heap space
```

≃ 1 Go

Boucle d'allocation v2

```
int[][] arrays = new int[1000][];
for (int i = 0; i < 1000; ++i) {
  arrays[i] = new int[1000000];
  arrays[i] = null;
}
```



Ramasse-miettes

La seconde version de la boucle produit ceci (toujours avec l'option `-verbose:gc`) :

```
[GC 16347K->368K(68992K), 0,0017120 secs]
[GC 16903K->400K(87040K), 0,0009230 secs]
[GC 36173K->368K(87040K), 0,0009550 secs]
[GC 35687K->352K(123136K), 0,0010370 secs]
[GC 70878K->384K(123136K), 0,0010970 secs]
[GC 70837K->336K(192832K), 0,0009030 secs]
[GC 141146K->308K(192832K), 0,0013020 secs]
[GC 141055K->308K(337216K), 0,0004210 secs]
…
[GC 379227K->308K(431040K), 0,0002650 secs]
[GC 379223K->308K(430528K), 0,0005050 secs]
```

et le programme se termine normalement.

Effacement des références

La présence du ramasse-miettes facilite beaucoup la gestion mémoire en Java mais il faut néanmoins prendre garde à éviter de conserver des références inutiles.

Par exemple, dans le code ci-dessous, il est important que la référence au tableau d'entiers disparaisse lors de l'appel à la méthode `remove`, afin que la mémoire qui lui est associée puisse être libérée :

```
List<Object> l = new ArrayList<>();
l.add(new int[1000000]);
l.remove(0);
```

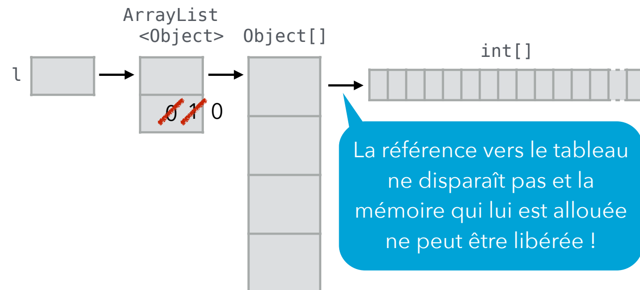

`ArrayList.remove`

Question : la méthode `remove` ci-dessous garantit-elle la disparition de la référence ?

```
public final class ArrayList<E>
    implements List<E> {
  private int size = 0;
  private Object[] array = new Object[…];
  …
  public void remove(int i) {
    checkIndex(i);
    System.arraycopy(array, i + 1,
                     array, i,
                     size - 1 - i);
    size -= 1;
  }
}
```

ArrayList.remove

```
List<Object> l = new ArrayList<>();
l.add(new int[1000000]);
l.remove(0);
```



ArrayList.remove

La méthode `remove` de `ArrayList` doit stocker `null` dans tous les éléments inutilisés du tableau sous-jacent :

```
public final class ArrayList<E>
    implements List<E> {
  …
  public void remove(int i) {
    checkIndex(i);
    System.arraycopy(array, i + 1,
                     array, i,
                     size - 1 - i);
    array[size - 1] = null;
    size -= 1;
  }
}
```


ArrayList.remove

```
List<Object> l = new ArrayList<>();
l.add(new int[1000000]);
l.remove(0);
```