Complément de cours de Compléments en Programmation Orientée Objet : immuabilité

Objet immuable :[1] objet non modifiable. Cette notion est importante car :

- elle aide à la vérification du programme : si l'objet immuable satisfait une certaine propriété à un moment donné, alors elle sera toujours satisfaite[2] ;
- en programmation concurrente, un objet immuable peut être partagé en sécurité car les accès en compétition (modifications concurrentes non synchronisées) sont impossibles.

Cette non-modifiabilité concerne tout le graphe de l'objet[3]. Tout comme la notion de graphe d'objet, la définition d'immuabilité d'un objet est à adapter au cas par cas, en fonction des propriétés que l'on cherche à assurer.

Ainsi, dans certains cas, on peut considérer que le graphe d'objet, c'est juste l'objet principal, ou bien que c'est tout l'ensemble des objets atteignables en suivant les références depuis l'objet principal, ou bien quelque chose entre les deux[4].

Type immuable : type de données dont toutes les instances (présentes et futures) sont des objets immuables. Concrètement en Java, il s'agit des types primitifs et des types définis par les classes[5] immuables (respectant les règles que nous allons détailler ci-dessous).

Type scellé : type dont les sous-types sont fixés dès la compilation du fichier qui le définit. En Java, les types scellés sont exactement les types définis par une classe `C` à constructeurs privés[6], une classe `final`[7], un type membre privé[8] ou une classe locale[9] et dont les sous-types directs sont eux-mêmes scellés (cela implique qu'ils sont tous dans le même fichier que le supertype, et récursivement tous leurs sous-types aussi).

Remarque : une énumération est un type scellé dont le nombre d'instances est fini.

Écrire un type immuable en Java : il s'agit d'écrire un type dont toutes les instances sont garanties immuables. Pour cela, il est nécessaire et suffisant que :

- ce type soit scellé (sinon on pourra créer des sous-types mutables)
- que pour chacun des sous-types, les instances créées soient immuables.

1. En Anglais : « *immutable* ». Notez que le contraire est, aussi bien en Anglais qu'en Français : « mutable ».
2. Par exemple, un carré immuable sera toujours un carré.
3. Tous le graphe d'enregistrements en mémoire se référençant les uns et les autres pour former un objet au sens abstrait du terme. Cf. cours.
4. Exemple raisonnable : l'objet principal + la partie encapsulée du graphe atteignable, c'est-à-dire les objets dont les références ne sont pas partagées (en UML, cela correspondrait à l'objet principal + les sous-objets qui le « composent »).
5. Il s'agit normalement de classes. Cela dit, en toute rigueur, on peut définir un type (privé) immuable à l'aide d'une interface membre privée. Il me semble que c'est le seul cas où ce ne sera pas une classe : en effet, rien ne peut empêcher d'implémenter une interface non privée depuis un autre fichier, sans recompiler le fichier définissant l'interface.
6. Dans ce cas, les sous-types directs ne peuvent être que dans le même type englobant de premier niveau, donc dans le même fichier.
7. Dans ce cas, pas de sous-type du tout.
8. Dans ce cas, les sous-types directs ne peuvent être que dans le même type englobant, donc le même fichier.
9. Dans ce cas, les sous-types sont tous définis dans la même méthode, donc le même fichier.

Pour le premier point, nous avons déjà expliqué comment le faire en Java.

Pour le second point, il suffit de faire en sorte que pour chaque nœud du graphe d'objet, soit tous ses attributs soient `final`, soit le nœud soit lui-même de type immuable pour une autre raison (par exemple : la documentation de sa classe dit que c'est le cas). Précisons que ceci est juste une condition suffisante qu'il est possible d'assouplir [10].

Remarquez qu'il est indispensable que les instances indirectes soient aussi immuables : en effet, le polymorphisme par sous-typage autorise une référence d'un type `C` donnée à pointer sur une instance d'un sous-type. Ainsi, si celui-ci n'était pas immuable, il n'est pas possible de compter sur le fait que l'objet pointé par cette référence ne sera jamais modifié.

Contre-exemple :

```
1  class PretendumentImmuable {
2     private final int x = 42; // garantie insuffisante
3     public int getX() { return x; }
4  }
5
6  class SousClasse extends PretendumentImmuable {
7     private int x = 42;
8     @Override public int getX() { return x; }
9     public void setX(int x) { this.x = x; }
10 }
11
12 ... // plus loin, dans une autre classe
13
14    public static void f(PretendumentImmuable v) {
15        System.out.println(v.getX());
16        if (v instanceof SousClasse) ((SousClasse) v).setX(43);
17        System.out.println(v.getX());
18    }
19
20    public static void main(String[] args) { f(new SousClasse()); /* -> affiche 42, puis 43 ! */ }
```

Pour aller un peu plus loin. L'immuabilité n'est en fait qu'un exemple de contrat qu'on définit par rapport à une instance, avant de généraliser la définition à un type tout entier.

Pour cette généralisation, le polymorphisme par sous-typage impose que le contrat soit aussi respecté par tous les sous-types. Or il est, en toute généralité, impossible d'imposer que tous les sous-types respectent le contrat hérité.

Une façon de s'en assurer est de sceller le type principal. Évidemment, cette restriction empêche de tierces personnes d'étendre ce type, ce qui n'est pas toujours acceptable. Ainsi, on se contente souvent d'écrire le contrat dans la Javadoc du type principal (dans ce cas, typiquement une interface), en précisant bien que les implémenteurs s'engageront à faire respecter ce contrat à leurs implémentations.

10. Par exemple : un attribut correctement encapsulé et qui n'aurait pas de setteur ou mutateur pourrait ne pas être `final`. Mais quel serait intérêt de ne pas écrire `final` ?
En fait, le cas peut se présenter si l'on souhaite utiliser, à l'intérieur du graphe d'un objet immuable, une instance d'une classe mutable (contenant des champs non `final`) écrite par un tiers, ou alors un tableau (les cases d'un tableau ne peuvent pas être marquées `final`).