Immuabilité

CS-108

Michel Schinz

2022-03-08

1 Introduction

Lors de la conception d'une classe, la question se pose de savoir si celle-ci doit être **immuable** ou non. Une classe est dite immuable si l'état de ses instances ne peut pas changer après leur création.

A première vue, définir des classes immuables peut sembler inutilement restrictif. Toutefois, comme nous le verrons, ces classes permettent d'éviter plusieurs problèmes affectant les classes non immuables, et sont donc généralement préférables.

Ces problèmes sont illustrés ci-dessous au moyen d'une classe très simple permettant de représenter une date du calendrier grégorien.

2 Dates non immuables

Une classe (*non* immuable) représentant une date du calendrier grégorien pourrait s'écrire ainsi :

```
public final class Date {
  private int y, m, d;          // année, mois, jour
  public Date(int y, int m, int d) {
    // … vérification des arguments omise
    this.y = y; this.m = m; this.d = d;
  }
  public int year() { return y; }
  public void setYear(int y2) { y = y2; }
  // … idem pour month/setMonth, day/setDay
  @Override
  public String toString() {
    return y + "-" + m + "-" + d;
  }
}
```

Cette classe n'est clairement pas immuable, puisque la valeur de ses attributs `y`, `m` et `d` peut être changée au moyen des méthodes `setYear`, `setMonth` et `setDay`.

La classe `Date` écrite, on peut l'utiliser pour en écrire une autre représentant une personne, ayant pour attributs un nom et une date de naissance :

```
public final class Person {
  private final String name;    // nom complet
  private final Date birthdate; // date de naissance
  public Person(String name, Date birthdate) {
    this.name = name;
    this.birthdate = birthdate;
  }
  public String name() { return name; }
  public Date birthdate() { return birthdate; }
}
```

Les classes `Date` et `Person` terminées, on peut les utiliser pour écrire le programme suivant :

```
Date d = new Date(1903, 12, 28);
Person j = new Person(”John Von Neumann”, d);
d.setYear(1969);
Person l = new Person(”Linus Torvalds”, d);

System.out.println(j.name() + ”, né le ” + j.birthdate());
System.out.println(l.name() + ”, né le ” + l.birthdate());
```

On pourrait s'attendre à ce que ce programme affiche :

```
John Von Neumann, né le 1903-12-28
Linus Torvalds, né le 1969-12-28
```

Or, étonnamment, il affiche que John Von Neumann et Linus Torvalds sont tous deux nés le 28 décembre **1969**. Pourquoi ? Car l'appel à `setYear` modifie la date de naissance stockée dans l'objet représentant John Von Neumann.

Pour corriger ce problème, étant donné que la classe `Date` n'est pas immuable, il faut impérativement copier la date reçue dans le constructeur de `Person` avant de la stocker dans l'attribut `birthdate`. Une telle copie est appelée **copie défensive** (*defensive copy*) car son but est de se défendre contre les éventuelles modifications que pourrait subir l'objet que l'on stocke.

Avant de pouvoir corriger le problème, il faut commencer par ajouter à la classe `Date` un moyen de copier une date. Cela peut se faire soit via l'implémentation de la méthode `clone`, soit via un constructeur de copie, solution adoptée ici :

```
public final class Date {
  // … comme avant
```

```
  public Date(Date that) {
    this(that.y, that.m, that.d);
  }
}
```

Le constructeur de copie ajouté, on peut corriger le constructeur de la classe `Person` afin qu'il stocke une copie de la date de naissance reçue :

```
public final class Person {
  // … comme avant
  public Person(String name, Date birthdate) {
    this.name = name;
    this.birthdate = new Date(birthdate);
  }
}
```

La classe `Person` est-elle désormais correcte ? Malheureusement non.

Pour le constater, posons-nous la question de savoir ce qu'affiche la variante suivante du programme présenté plus haut :

```
Date d = new Date(1903, 12, 28);
Person j = new Person(”John Von Neumann”, d);
Date b = j.birthdate();
b.setYear(1969);
Person l = new Person(”Linus Torvalds”, b);

System.out.println(j.name() + ”, né le ” + j.birthdate());
System.out.println(l.name() + ”, né le ” + l.birthdate());
```

Malheureusement, une fois encore, elle affiche que John Von Neumann et Linus Torvalds sont tous deux nés en 1969. Pourquoi ? Car l'appel à `setYear` modifie une fois encore la date de naissance de John Von Neumann, mais pour une raison différente de la précédente. Cette fois-ci, le problème est que la méthode `birthdate` retourne la date de naissance stockée en interne par la classe `Person`, ce qui permet à un client de modifier celle-ci. Cela n'est clairement pas souhaitable : il y a faille d'encapsulation.

Pour corriger cette faille, il faut également faire une copie défensive dans la méthode `birthdate` afin de retourner une copie de la date de naissance :

```
public final class Person {
  // … comme avant
  public Date birthdate() {
    return new Date(birthdate);
  }
}
```

Devoir faire ces copies défensives est problématique pour deux raisons :

1. elles doivent être faites par tous les utilisateurs de la classe non immuable,
2. il est très facile de les oublier, et les problèmes posés par de tels oublis — données corrompues, principalement — sont difficiles à diagnostiquer.

Il est donc préférable d'éviter totalement ces problèmes en définissant, autant que possible, des classes immuables.

Règle de l'immuabilité : Autant que possible, définissez des classes immuables.

3 Dates immuables

Pour rendre la classe `Date` immuable, il convient de la modifier ainsi :

- les attributs `y`, `m` et `d` deviennent finaux (`final`),
- les méthodes de modification de ces attributs (`setYear`, `setMonth`, `setDay`) sont supprimées,
- le constructeur de copie, devenu inutile, est supprimé.

Ces modifications faites, on obtient :

```
public final class Date {
  private final int y, m, d;    // année, mois, jour
  public Date(int y, int m, int d) {
    // … vérification des arguments omise
    this.y = y; this.m = m; this.d = d;
  }
  public int year() { return y; }
  public int month() { return m; }
  public int day() { return d; }
}
```

Les dates étant maintenant immuables, les copies défensives faites dans le constructeur et dans la méthode `birthdate` de `Person` deviennent inutiles et peuvent être supprimées.

3.1 Méthodes de dérivation

Un avantage de la version *non* immuable de la classe `Date` était la facilité avec laquelle on pouvait obtenir une date ressemblant à une autre en ne changeant qu'une partie des valeurs, p.ex. :

```
Date d = new Date(1903, 12, 28);
d.setYear(1969); // maintenant 1969-12-28
```

Doit-on renoncer à ce style de programmation lorsqu'on rend les dates immuables ? Non, il suffit d'ajouter des méthodes retournant une nouvelle date dérivée d'une date existante.

Dans le cas des dates immuables, on peut ainsi ajouter trois méthodes, `withYear`, `withMonth` et `withDay` permettant d'obtenir une nouvelle date dont tous les attributs sauf un sont égaux à ceux de la date à laquelle on l'applique :

```
public final class Date {
  // … comme avant
  public Date withYear(int newY) {
    return new Date(newY, m, d);
  }
  // … idem pour withMonth et withDay
}
```

Ces méthodes ajoutées, on peut récrire notre exemple pour obtenir quelque chose d'aussi concis qu'avec les dates non immuables :

```
Date d = new Date(1903, 12, 28);
Date d2 = d.withYear(1969); // 1969-12-28
```

L'avantage de cette version est que l'on a maintenant toujours les deux dates à disposition : `d` représente le 28 décembre 1903 tandis que `d2` représente le 28 décembre 1969.

4 Classes immuables

Comparées aux classes qui ne le sont pas, les classes immuables ont de nombreux avantages :

- il est facile de raisonner à leur sujet, l'état de leurs instances ne changeant jamais au cours de leur vie,
- il n'est jamais nécessaire de faire des copies défensives de leurs instances — copies qu'il est facile d'oublier,
- plus généralement, il n'est jamais nécessaire de copier leurs instances, donc elle ne nécessitent ni méthode `clone`, ni constructeur de copie,
- elles peuvent être partagées entre plusieurs fils d'exécution (*threads*) dans les applications concurrentes.

Malgré leurs nombreux avantages, les classes immuables ne sont pas totalement dénuées de défauts :

- lorsqu'une classe immuable est utilisée pour représenter une valeur qui change très souvent, le fait de devoir créer une nouvelle instance à chaque changement peut nuire aux performances,
- lorsqu'il faut effectivement que le changement d'état d'un objet soit visible à tous les possesseurs d'une référence vers cet objet, le fait que celui-ci ne soit pas immuable peut simplifier la programmation.

En pratique, malgré ces inconvénients, il reste préférable d'utiliser des classes immuables autant que possible.

Notez que ce n'est que récemment que l'utilisation de données immuables est devenue réaliste en programmation, entre autres car la rapidité des processeurs modernes permet de compenser les pertes de performances que peut induire l'immuabilité. Par conséquent, les bibliothèques « anciennes » — à commencer par celle de Java — comportent souvent un grand nombre de classes non immuables. Ne les prenez pas comme modèle ! Par exemple, la classe `java.util.Date` est terriblement mal conçue, entre autres car elle n'est pas immuable.

Comme vous l'aurez peut-être constaté, les avantages d'une approche immuable (aussi appelée « non destructive ») commencent à être reconnus dans d'autres domaines que la programmation. Par exemple, les logiciels de retouche d'image récents comme Lightroom d'Adobe ou Photos d'Apple ne modifient pas l'image existante mais en produisent une nouvelle, contrairement à des logiciels plus anciens comme Photoshop. Il en va de même des bases de données récentes comme Datomic, qui ne modifie jamais les données qu'elle stocke contrairement à ses ancêtres (MySQL, PostgreSQL, etc.)

4.1 Immuable / non modifiable

Il importe de ne pas confondre deux notions proches : l'immuabilité et la « non modifiabilité ». Ainsi, on dit qu'une classe est :

immuable si ses instances ne peuvent pas changer d'état une fois créées,

non modifiable ou « en lecture seule » (*read-only*) si un morceau de code ayant accès à l'une de ses instances n'a pas la possibilité d'appeler des méthodes modifiant son état.

Dès lors, une classe immuable est toujours non modifiable, mais l'inverse n'est pas forcément vrai. Par exemple, la classe `Date` ci-dessous n'est pas modifiable, dans le sens où un morceau de code ayant accès à une de ses instances ne peut modifier son contenu (sauf dans certains cas en utilisant le transtypage) :

```
class Date {
  protected int y, m, d;
  // constructeur omis
```

```
  public int year() { return y; }
  public int month() { return m; }
  public int day() { return d; }
}
class ModifiableDate extends Date {
  public void setYear(int newY) { y = newY; }
}
```

Par contre, un objet ayant le type `Date` n'est pas forcément immuable, car il peut s'agir en réalité d'une instance de `ModifiableDate`.

Dans la vie réelle, du point de vue du propriétaire d'un compte en banque, le montant disponible sur ce compte est non modifiable, mais il n'est pas immuable pour autant. En effet, le propriétaire d'un compte ne peut pas directement changer le montant disponible. Par contre, ce montant change au cours du temps, en fonction des transferts d'argent effectués sur le compte.

4.2 Définition de classes immuables

Pour qu'une classe soit immuable, il faut qu'elle satisfasse les conditions suivantes :

- tous ses attributs sont finaux (`final`), initialisés lors de la construction et jamais modifiés par la suite,
- toute valeur non immuable fournie à son constructeur est copiée en profondeur avant d'être stockée dans un de ses attributs,
- aucune valeur non immuable stockée dans un de ses attributs n'est fournie à l'extérieur, p.ex. par un accesseur : soit chacune de ces valeurs non immuable est rendue non modifiable avant d'être fournie à l'extérieur, soit seule une copie profonde est fournie.

De plus, dans la plupart des cas, la classe elle-même doit être finale (`final`), faute de quoi il est possible de violer l'immuabilité en définissant une sous-classe non immuable.

Notez que les enregistrements (*records*) introduits en Java 17 satisfont la plupart de ces contraintes, puisque les classes qu'ils définissent sont finales et leurs attributs également. Pour en faire de véritables classes immuables, il reste à s'assurer que leur constructeur copie les éventuelles valeurs non immuables qu'il reçoit avant de les stocker dans les attributs.

Les conditions ci-dessus impliquent que, s'il est facile d'écrire une classe immuable composée uniquement de valeurs immuables, les choses se compliquent lorsque des valeur non immuables entrent en jeu.

Pour écrire une classe immuable, il faut donc autant que possible se baser sur d'autres classes immuables, mais ce n'est malheureusement pas toujours possible, par exemple avec les tableaux.

5 Tableaux et immuabilité

Un tableau Java « normal » (c-à-d pas un tableau dynamique de type `ArrayList`) est *toujours* modifiable. Dès lors, l'utilisation d'un tableau dans une classe immuable complique celle-ci, qui doit faire des copies défensives des tableaux qu'elle reçoit de l'extérieur avant de les stocker, et faire des copies défensives de ses tableaux internes avant de les fournir à l'extérieur.

Avec les tableaux dynamiques, une solution plus simple et moins coûteuse existe. La méthode `unmodifiableList` de la classe `java.util.Collections` permet d'obtenir une version non modifiable d'un tableau dynamique, dont toutes les méthodes de modification lèvent l'exception `UnsupportedOperationException`. Exemple :

```
ArrayList<String> m = new ArrayList<String>();
m.add(”un”); m.add(”deux”); m.add(”trois”);
List<String> u = Collections.unmodifiableList(m);
u.add(”quatre”); // lève UnsupportedOperationException
```

La méthode `unmodifiableList` constitue le moyen le plus simple de définir une classe immuable utilisant un tableau. Une telle classe s'écrit ainsi :

- les tableaux reçus à la construction sont copiés défensivement, rendus non modifiables au moyen de la méthode `unmodifiableList` puis stockés ainsi dans des attributs,
- ces tableaux non modifiables sont directement retournés par les méthodes d'accès.

Notez que `unmodifiableList` retourne une valeur de type `List` (une interface) et pas `ArrayList`. L'interface `List` sera décrite en détail dans le cours sur les collections mais peut pour l'instant être considérée comme équivalente à `ArrayList`.

Une classe représentant un vecteur immuable pourrait p.ex. s'écrire ainsi :

```
import java.util.List;
import java.util.ArrayList;
import static java.util.Collections.unmodifiableList;

public final class Vector {
  private final List<Double> v;
  public Vector(List<Double> a) {
    v = unmodifiableList(new ArrayList<Double>(a));
  }
  public List<Double> asList() {
    return v;
  }
}
```

Depuis la version 10 de Java, l'interface `List` offre une méthode nommée `copyOf` permettant d'obtenir une copie non modifiable d'une liste. On peut l'utiliser pour simplifier le constructeur de la classe `Vector` :

```
import java.util.List;

public final class Vector {
  private final List<Double> v;
  public Vector(List<Double> a) {
    v = List.copyOf(a);
  }
  public List<Double> asList() {
    return v;
  }
}
```

Cette nouvelle méthode rend le code beaucoup plus clair, et est toujours préférable pour peu que l'utilisation d'une version récente de Java (10 ou plus) ne pose pas de problème.

6 Bâtisseurs

Par définition, pour créer une instance d'une classe immuable, il faut spécifier la valeur de tous ses attributs. Ainsi, pour créer un vecteur immuable, il faut spécifier la valeur de tous ses éléments. Parfois, le fait de devoir spécifier la totalité des attributs est peu agréable. Par exemple, lorsqu'on lit les éléments d'un vecteur un à un depuis un fichier, il est plus simple de les ajouter un à un au vecteur plutôt que de les accumuler dans un tableau puis de créer un vecteur à partir de celui-ci.

Faut-il dès lors abandonner l'immuabilité pour faciliter la construction par étapes d'instances ?

Plutôt que d'abandonner l'idée d'avoir des classes immuables uniquement pour faciliter leur construction par étapes, il est plus judicieux d'offrir une classe séparée, non immuable, dont le seul but est de permettre la construction progressive d'instances d'une classe immuable. Une telle classe s'appelle un **bâtisseur** (*builder*).

6.1 Bâtisseur de date

Pour illustrer le concept de bâtisseur, nous allons en développer un pour la classe `Date` immuable présentée plus haut. La construction d'une instance de `Date` ne demandant que trois valeurs — le jour, le mois, l'année — la définition d'un bâtisseur n'est pas vraiment justifiée. Cela n'a toutefois que peu d'importance, les concepts présentés ci-dessous étant faciles à adapter aux cas pour lesquels un bâtisseur est véritablement utile, par exemple pour une classe de vecteurs ou de matrices immuables.

La classe d'un bâtisseur ressemble à la classe dont les instances sont à bâtir. En particulier, un bâtisseur a généralement les mêmes attributs que la classe qu'il bâtit, mais ceux-ci sont — dans la plupart des cas — modifiables.

Un bâtisseur de dates contient donc trois attributs : un pour l'année, un pour le mois, un pour le jour. A la différence des attributs de la classe `Date`, ceux-ci sont modifiables via des méthodes (`setYear`, `setMonth`, `setDay`). La valeur initiale de ces trois attributs est ici spécifiée à la construction, mais dans certains cas on peut imaginer utiliser des valeurs par défaut — p.ex. la date du jour actuel.

Finalement, la méthode `build` construit une instance de la classe à bâtir, ici une date.

```
public final class DateBuilder {
  private int y, m, d;
  public DateBuilder(int y, int m, int d) {
    this.y = y; this.m = m; this.d = d;
  }
  public int year() { return y; }
  public void setYear(int y2) { y = y2; }
  // … idem pour month/setMonth et day/setDay

  public Date build() {
    return new Date(y, m, d);
  }
}
```

Ce bâtisseur peut encore être amélioré en changeant le type de retour des méthodes de modification (`set…`). Plutôt que de ne rien retourner, elles peuvent retourner le bâtisseur lui-même, c-à-d `this`. Par exemple, la méthode `setYear` devient alors :

```
public DateBuilder setYear(int y2) {
  y = y2;
  return this;
}
```

Cette modification permet de chaîner les appels :

```
Date d = new DateBuilder(1903, 12, 28)
  .setYear(1969)
  .build(); // 1969-12-28
```

6.2 Bâtisseurs et classes non immuables

Le bâtisseur de dates ressemble fortement à la version non immuable de `Date`. Ne retrouve-t-on donc pas les mêmes problèmes ?

Non, car le bâtisseur n'est utilisé que brièvement pour *construire* des dates, pas pour les représenter par la suite. Par exemple, la classe `Person` stocke une date immuable de

type `Date` — avec tous les avantages que cela comporte — et pas un bâtisseur de date de type `DateBuilder`.

De manière générale, un bâtisseur n'est utilisé que très localement et n'est pas stocké dans un attribut d'une classe. De par son aspect local, une telle utilisation d'un objet non immuable ne pose pas de problème.

6.3 Bâtisseurs imbriqués

Les classes `Date` et `DateBuilder` développées plus haut sont intimement liées, dans le sens où la seconde n'a aucun sens sans la première. Dès lors, il serait bien de pouvoir les définir en commun, ce qui peut se faire en Java par imbrication d'une classe dans une autre.

Pour imbriquer le bâtisseur de date `DateBuilder` dans la classe `Date`, il suffit de :

- déplacer la classe `DateBuilder` à l'intérieur de la classe `Date`,
- lui attacher l'attribut `static`, pour en faire ce que l'on nomme une **classe imbriquée statiquement**,
- la renommer en `Builder`, le préfixe `Date` étant redondant suite à l'imbrication.

En effectuant les opérations susmentionnées, on obtient le résultat suivant :

```
public final class Date {
  // … comme avant
  public final static class Builder {
    // … comme avant
  }
}
```

A noter que depuis l'extérieur de la classe `Date`, il faut utiliser la notation pointée pour désigner la classe `Builder` imbriquée, en écrivant p.ex. `new Date.Builder(…)`. C'est la raison pour laquelle nous avons renommé cette classe.

La possibilité d'imbriquer des classes en Java est utile dans d'autres cas que celui des bâtisseurs, et sera examinée plus tard dans toute sa généralité. Pour l'instant, le seul type d'imbrication qui nous intéresse est celui utilisé ci-dessus, à savoir l'imbrication dite statique.

Une classe imbriquée statiquement dans une autre est très similaire à une classe non imbriquée. Les différences principales sont :

1. de l'extérieur de la classe englobante, le nom d'une classe imbriquée statiquement est précédé de celui de sa classe englobante (`Date.Builder` dans notre cas),
2. une classe imbriquée statiquement a accès aux membres privés statiques (`private static`) de sa classe englobante, ainsi qu'à ses constructeurs privés,

3. une classe imbriquée statiquement peut être déclarée privée (`private`) ou protégée (`protected`), affectant sa visibilité depuis l'extérieur selon les règles usuelles.

Attention, une classe imbriquée statique ne peut accéder qu'aux membres *statiques* de sa classe englobante !

6.4 Exemple : `StringBuilder`

Des bâtisseurs sont parfois utilisés dans la bibliothèque Java pour faciliter la construction d'instances de classes immuables.

Un exemple important est la classe `StringBuilder`, un bâtisseur de chaînes de caractères. Pour mémoire, la classe Java des chaînes de caractères, `String`, est immuable — comme il se doit.

Le fonctionnement de la classe `StringBuilder` est assez similaire à celui du bâtisseur de dates décrit plus haut, si ce n'est que sa méthode de construction se nomme `toString` et pas `build`. L'extrait de programme ci-dessous illustre son utilisation :

```
StringBuilder b = new StringBuilder("[");
for (int i = 1; i <= 10; ++i) {
  b.append(i);
  if (i < 10)
    b.append(", ");
}
String s = b.append("]").toString();
// s vaut "[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]"
```

(On notera que cet exemple s'écrirait de manière encore plus simple au moyen de la classe `StringJoiner`, qui est un bâtisseur de chaînes spécialisé au cas où la chaîne à bâtir est composée de différentes parties séparées par une chaîne de séparation, ici la virgule. Il n'en reste pas moins que la classe `StringBuilder` est utile dans un grand nombre de situations.)

6.5 Définition de bâtisseurs

En conclusion, un bâtisseur peut être une addition bienvenue à une classe immuable dans les cas où il est utile de construire des instances par étapes. La règle ci-dessous énonce cela et résume la marche à suivre pour définir un bâtisseur.

Règle des bâtisseurs : S'il peut être utile de construire par étapes des instances d'une classe immuable, associez-lui un bâtisseur.

De plus :

- nommez la classe du bâtisseur `Builder`,

- imbriquez-la statiquement dans la classe dont elle bâtit des instances,
- nommez sa méthode de construction `build`, et
- retournez `this` de toutes les méthodes de modification pour permettre les appels chaînés.

7 Références

- *Effective Java (3rd ed.)* de Joshua Bloch, en particulier :
  - la règle 17, *Minimize mutability* sur l'intérêt des classes immuables.