Immuabilité

Pratique de la programmation orientée-objet

Michel Schinz – 2015-02-16

Classe des dates

On désire écrire une classe représentant une date du calendrier grégorien :

```
public final class Date {
  private int y, m, d;
  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;
  }
}
```

Classe des personnes

La classe `Date` écrite, on désire l'utiliser pour en écrire une autre représentant une personne :

```
public final class Person {
  private final String name;
  private final Date bdate;
  public Person(String name, Date bdate) {
    this.name = name;
    this.bdate = bdate;
  }
  public String name() { return name; }
  public Date birthdate() { return bdate; }
}
```

Création de personnes

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

```
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() + " est né le " + j.birthdate());
System.out.println(
  l.name() + " est né le " + l.birthdate());
```

Question : qu'affiche ce programme et pourquoi ?

Copie défensive

Etant 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 le champ `bdate`.
Une telle copie est appelée **copie défensive** (*defensive copy*) car son but est de se défendre contre les éventuelles modifications que pourraient subir l'objet que l'on stocke.

Copie de date

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);
  }
}
```

Copie de date

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 bdate) {
    this.name = name;
    this.bdate = new Date(bdate);
  }
}
```

La classe `Person` est-elle désormais correcte ?

Exercice

Question : qu'affiche le programme ci-dessous ?

```
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() + " est né le " + j.birthdate());
System.out.println(
  l.name() + " est né le " + l.birthdate());
```

Copie défensive (bis)

Le fait que la méthode `birthdate` retourne la date de naissance stockée en interne permet à un client de modifier celle-ci, ce qui 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);
  }
}
```

Copies défensives

Ces copies défensives sont problématiques pour deux raisons :

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

Conclusion : il est préférable d'éviter totalement ces problèmes en définissant, autant que possible, des classes immuables.

Règle de l'immuabilité

Dans la mesure du possible, écrivez des classes immuables.

Dates immuables

La classe des dates immuables ressemble beaucoup à celle des dates non immuables, les différences étant :

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

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.

Dates immuables

```
public final class Date {
  private final int y, m, d;
  public Date(int y, int m, int d) {
    this.y = y; this.m = m; this.d = d;
  }
  public int year() { return y; }
  public int month() { return m; }
  public int day() { return d; }
}
```

Dates dérivées

Un avantage de la classe `Date` non immuable est qu'il était facile d'obtenir une date ressemblant à une autre en ne changeant qu'une partie des valeurs :

```
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.

Dates dérivées

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 y2) {
    return new Date(y2, m, d);
  }
  // … idem pour withMonth et withDay
}
```

Dates dérivées

Les méthodes de dérivation de nouvelles dates écrites, 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,
- d2 représente le 28 décembre 1969.

Avantages de l'immuabilité

Comparées aux classes modifiables, 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 des applications concurrentes.

Inconvénients de l'immuabilité

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

1. 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,
2. 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.

Remarque historique

Ce n'est que récemment que l'utilisation de données immuables est devenue réaliste en programmation, pour plusieurs raisons. En particulier, 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 en exemple !
(P.ex. `java.util.Date` est terriblement mal conçue, entre autres car elle n'est pas immuable).

Immuabilité en informatique

Ce « virage vers l'immuabilité » est l'une des grandes tendances de l'informatique moderne. Dans beaucoup de domaines, les avantages d'une approche immuable (aussi appelée **non-destructive**) commencent à être reconnus :

- les logiciels de retouche d'images modernes (p.ex. Lightroom ou iPhoto) produisent une nouvelle image plutôt que de modifier l'image existante comme le faisaient leurs ancêtres (p.ex. Photoshop),
- certaines bases de données moderne (p.ex. Datomic) ne modifient jamais les données, contrairement à leurs ancêtres (p.ex. MySQL, PostgreSQL),
- etc.

Terminologie

Une classe est **immuable** si ses instances ne peuvent pas changer d'état une fois créées.
Une classe est **non modifiable** 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.
Attention : même si une classe immuable n'est jamais modifiable, l'inverse n'est pas forcément vrai.

Immuable/non modifiable

La classe `Int` 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 Int {
  protected int i = 0;
  public int get() { return i; }
}
class ModifiableInt extends Int {
  public void set(int newI) { i = newI; }
}
```

Un objet de type `Int` n'est pas forcément immuable pour autant, car il peut s'agir d'une instance de `ModifiableInt`.

Classe immuable

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

1. tous ses champs sont finaux (`final`), initialisés lors de la construction et jamais modifiés par la suite,
2. toute valeur non immuable fournie à son constructeur doit être copiée en profondeur avant d'être stockée dans un de ses champs,
3. aucune valeur non immuable stockée dans un de ses champs ne doit être fournie à l'extérieur : il faut soit la rendre non modifiable au préalable, soit fournir une copie profonde.

Classe immuable

Les conditions précitées 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.

Tableaux et immuabilité

Un tableau Java « normal » (pas dynamique) 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.

unmodifiableList

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` (abrégée `UOE`).

Exemple :

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

unmodifiableList

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 par `unmodifiableList` puis stockés ainsi dans des champs,
- 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.

Exemple

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

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

Bâtisseurs

Construction d'objets

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 ?

Bâtisseur

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, modifiable, dont le seul but est de permettre la construction progressive d'instances d'une classe immuable.
Une telle classe s'appelle un **bâtisseur** (*builder*).

Bâtisseur de dates

Pour illustrer le concept de bâtisseur, nous allons en développer un pour la classe `Date` immuable présentée précédemment.
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 plus loin étant faciles à adapter aux cas pour lesquels un bâtisseur est véritablement utile, par exemple pour une classe de vecteurs ou de matrices immuables.

Bâtisseur de dates

La classe d'un bâtisseur ressemble à la classe dont les instances sont à bâtir. En particulier, un bâtisseur a généralement les mêmes champs que la classe qu'il bâtit, mais ceux-ci sont – dans la plupart des cas – modifiables.
Un bâtisseur de dates contient donc trois champs : un pour l'année, un pour le mois, un pour le jour. A la différence des champs de la classe `Date`, ceux-ci sont modifiables via des méthodes (`setYear`, `setMonth`, `setDay`).
La valeur initiale de ces trois champs est ici spécifiée à la construction, mais dans certains cas on peut imaginer utiliser des valeurs par défaut (p.ex. la date du jour).
Finalement, la méthode `build` construit une instance de la classe à bâtir, ici une date.

Bâtisseur de dates

```
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);
  }
}
```

Appels chaînés

Plutôt que de ne rien retourner (type `void`), les méthodes de modification du bâtisseur (`set...`) 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;
}
```

Cela permet de chaîner les appels :

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

Bâtisseur/classe modifiable

Le bâtisseur de dates ressemble fortement à la version modifiable de `Date`… Ne retrouve-t-on donc pas les mêmes problèmes ?
Non, car le bâtisseur n'est utilisé que pour construire des dates, pas pour les représenter. Par exemple, la classe `Person` stocke une date immuable – avec tous les avantages que cela comporte – et pas un bâtisseur de date.
De manière générale, un bâtisseur ne sera utilisé que très localement et ne sera jamais 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.

Règle des bâtisseurs

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

De plus :

- nommez la classe du bâtisseur `Builder`,
- imbriquez-la statiquement dans la classe dont elle bâtit des instances (voir plus loin),
- nommez sa méthode de construction `build`, et
- retournez `this` de toutes les méthodes de modification (`set...`) pour permettre les appels chaînés.

StringBuilder

La bibliothèque Java offre la classe `StringBuilder` pour bâtir des chaînes de caractères (`String`), qui sont immuables. Ce bâtisseur est généralement conçu selon les règles précitées, si ce n'est que sa méthode de construction s'appelle `toString`. Exemple d'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]"
```

Classes imbriquées statiques

Bâtisseur de dates

Les classes `Date` et `DateBuilder` 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 qui offre la possibilité d'imbriquer une classe dans une autre.

Bâtisseur imbriqué

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

1. déplacer la classe `DateBuilder` à l'intérieur de la classe `Date`,
2. lui attacher l'attribut `static` – car il s'agit d'une classe imbriquée statique,
3. la renommer en `Builder`, le préfixe `Date` étant redondant comme nous le verrons.

Bâtisseur imbriqué

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.

Classes imbriquées en Java

Une **classe imbriquée** (*nested class*) est une classe définie à l'intérieur d'une autre.

Java offre deux types de classes imbriquées :

- les **classes imbriquées statiques** (*static nested classes*), décrites ici,
- les classes imbriquées non statiques, souvent appelées *inner classes* en anglais – que l'on pourrait traduire par **classes intérieures** – qui seront décrites ultérieurement.

Classes imbriquées statiques

Une classe imbriquée statique 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 statique a accès aux membres privés statiques de sa classe englobante,
3. une classe imbriquée statique peut être déclarée privée (`private`) ou protégée (`protected`).

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