Vers un code Immuable

Mohamed / Cyril / Gaëtan / Lucas

Histoire de l’immuabilité

Pas associé à de date spécifique (ce n’est pas une innovation technologique introduite à un moment spécifique dans l’histoire).

Concept avec différentes manières de mise en oeuvre (besoin du développeur/exigences du projet)

Évolution au fil du temps (en fonction des besoins).

Différentes stabilité/contextes

Amélioration (stabilité, prévisibilité,sécurité)

Pas design pattern-> Pas de GOF ni SOLID

Qu’est-ce qu’un code muable

Facilité de modification (évolutions, exigences, besoins) -> pertinence et fonctionnalité continue muabilité (mise a jour, fonctionnalités, performance et personnalisation par utilisateur)

Méthode agile -> encourage la capacité du code à s’adapter facilement, qualité reconnue dans le développement logiciel (agilité et pérennité des applications)

En java : attributs modifiable (setname -> name) = classe mutable

Qu’est-ce qu’un code immuable ?

Pas modifiable (après écriture/compilation)

Parties du code (variables, fonction ou structures)

-> non modifiables/altérables après création initiale

Exemple :

Classe pas immuable car la valeur de ses attributs peuvent changer -> setters

```java
package fr.unilim.iut;

public class Date {
    private int year, month, day;

    public Date(int year, int month, int day) {
        this.year = year; this.month = month; this.day = day;
    }
    /*--------GETTERS-----------*/
    public int getYear() {
        return year;
    }
    public void setYear(int year) {
        this.year = year;
    }
    public int getMonth() {
        return month;
    }
    /*--------SETTERS-------------*/
    public void setMonth(int month) {
        this.month = month;
    }
    public int getDay() {
        return day;
    }
    public void setDay(int day) {
        this.day = day;
    }

    @Override
    public String toString() {
        return year + "-" + month + "-" + day;
    }
}
```

L'intérêt de l'immuabilité

Classe date -> écrire une autre représentant une **personne** (nom, date de naissance)

```java
package fr.unilim.iut;

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

Classes terminer on programme la suite :

```java
package fr.unilim.iut;

public class Main {

    public static void main(String[] args) {
        Date firstDate = new Date(2004, 5, 5);
        Personne firstPerson = new Personne("Mohamed Mesri", firstDate);
        firstDate.setYear(2002);
        Personne secondPerson = new Personne("Cyril Romero", firstDate);

        System.out.println(firstPerson.name() + ", né le " + firstPerson.birthdate());
        System.out.println(secondPerson.name() + ", né le " + secondPerson.birthdate());

    }

}
```

Affichage (attendus) :

Mohamed Mesri né le 2004-5-5

Cyril Romero né le 2002-5-5

On obtient

```
Mohamed Mesri, né le 2002-5-5
Cyril Romero, né le 2002-5-5
```

L'intérêt de l'immuabilité

Pourquoi ?

setYear modifie la date de naissance stockée dans l'objet représentant Mohamed Mesri.

Correction :

Classe(date) non immuable -> copier date reçue dans le constructeur de Person avant le stockage(attribut)

(=defensive copy) -> défendre des modifications

Avant correction : ajouter un moyen de copier une date dans Date(classe)-> via constructeur

Correction du constructeur de Person -> stock une copie de la date reçue

```
public Date(Date date) {
        this(date.year, date.month, date.day);
}
```

```
public Personne(String name, Date birthdate) {
  this.name = name;
  this.birthdate = new Date(birthdate);
}
```

L'intérêt de l'immuabilité

Classe Person correcte ? non -> demandons-nous ce qu'affiche la variante du programme précédent

```
package fr.unilim.iut;

public class Main {

    public static void main(String[] args) {
        Date firstDate = new Date(2004, 5, 5);
        Personne firstPerson = new Personne("Mohamed Mesri", firstDate);
        Date secondDate = firstPerson.getBirthdate();
        secondDate.setYear(2002);
        Personne secondPerson = new Personne("Cyril Romero", secondDate);

        System.out.println(firstPerson.getName() + ", né le " + firstPerson.getBirthdate());
        System.out.println(secondPerson.getName() + ", né le " + secondPerson.getBirthdate());

    }

}
```

Affichage -> le même que précédemment (*setYear* modifie la date de Mohamed Mesri

Raison différente : méthode *getBirthday* retourne la date stockée (Person) -> client peut modifier (=faille d'encapsulation)

Correction : copie defensive de *getBirthday*

```
public Date getBirthdate(){
    return new Date(birthdate);
}
```

```
Mohamed Mesri, né le 2004-5-5
Cyril Romero, né le 2002-5-5
```

Pourquoi faire appel à l'immuabilité

Devoir faire ces copies défensives est problématique pour deux raisons :

1- les faires par tous les utilisateurs de la classe non immuable

2- facilement oubliable -> engendre des problèmes (données corrompues) difficilement diagnosticable

Définir le plus possible des classes immuables pour éviter les copies défensives

Exemple de code immuable

Pour rendre la classe Date immuable :

- Modification des attributs year, month et day en final
- Suppression de setYear, setMonth et setDay
- Suppression constructeur de copie

```java
public Date withYear(int newY) {
    return new Date(newY, month, day);
}

public Date withMonth(int newM) {
    return new Date(year, newM, day);
}

public Date withDay(int newD) {
    return new Date(year, month, newD);
}
```

Ajout de 3 méthodes :

- withYear
- withMonth
- withDay

```java
package fr.unilim.iut;

public final class Date {
    private final int year, month, day;

    public Date(int year, int month, int day) {
        this.year = year; this.month = month; this.day = day;
    }

    /*--------GETTERS-----------*/
    public int getYear() {
        return year;
    }
    public int getMonth() {
        return month;
    }
    public int getDay() {
        return day;
    }

    @Override
    public String toString() {
        return year + "-" + month + "-" + day;
    }
}
```

```java
package fr.unilim.iut;

public class Main {

    public static void main(String[] args) {
        Date firstDate = new Date(2004, 5, 5);
        Date secondDate = firstDate.withYear(2002);
        Personne firstPerson = new Personne("Mohamed Mesri", firstDate);
        Personne secondPerson = new Personne("Cyril Romero", secondDate);

        System.out.println(firstPerson.getName() + ", né le " + firstPerson.getBirthdate());
        System.out.println(secondPerson.getName() + ", né le " + secondPerson.getBirthdate());

    }

}
```

→

```
Mohamed Mesri, né le 2004-5-5
Cyril Romero, né le 2002-5-5
```

Avantages Immuabilité

Les classes immuables -> avantage notable prévisibilité et gestion de la mémoire. Caractéristique principale -> état de leurs instances inchangé tout au long cycle de vie. Cette stabilité -> bénéfices :

- 1. **Raisonnement aisé** : Leur état constant -> compréhension simplifié -> meilleure maintenance du code.

- 2. **Élimination des copies défensives** : Pas besoin de créer des copies

- 3. **Pas besoin de copier des instances** -> simplifie la manipulation d'objets et réduit la complexité du code.

- 4. **Partage entre fils d'exécution** : partage sécurisé sans nécessiter de mécanismes de synchronisation complexes.

Classes immuables -> stabilité, simplicité et sécurité.
Facilite développement/maintenance/gestion apps.

Inconvénients Immuabilité

Malgré les avantages des classes immuables, elles ont quelques inconvénients :

- 1. Création fréquente de nouvelles instances pour les valeurs qui changent beaucoup ce qui peut impacter les 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.

Malgré ces inconvénients, l’utilisation de classes immuables reste généralement préférable.

Immuabilité VS non modifiable

Important : pas confondre deux notions proches : l'**immuabilité** et « **non modifiabilité** ».
Ainsi, on dit qu'une classe est :

- **immuable** si ses instances ne peuvent pas changer d'état une fois créées.
- **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.

Dès lors, une classe immuable est toujours non modifiable, mais l'inverse n'est pas forcément vrai.

Immuabilité VS non modifiable

```java
package fr.unilim.iut;

public class CompteBancaire {
    private final int numeroCompte;
    private double solde;

    public CompteBancaire(int numeroCompte, double soldeInitial) {
        this.numeroCompte = numeroCompte;
        this.solde = soldeInitial;
    }

    public int getNumeroCompte() {
        return numeroCompte;
    }

    public double getSolde() {
        return solde;
    }

    public void deposer(double montant) {
        if (montant > 0) {
            solde += montant;
        }
    }

    public void retirer(double montant) {
        if (montant > 0 && montant <= solde) {
            solde -= montant;
        }
    }
}
```

Classe BankAccount ->compte bancaire avec un numéro de compte et un solde initial. Solde (retrait/depot) -> modification mais structure baseclasse -> inchangée. Classe ->"**non modifiable**" -> sens structure fondamentale stable. Pas immuable -> solde peut être modifié.

La non-modifiabilité ici -> structure classe Immuabilité -> 'impossibilité changer état interne de l'objet une fois créé.

Dans cet exemple, la structure de base de la classe BankAccount est conçue pour rester inchangée, mais l'état du solde peut être modifié en utilisant les méthodes de dépôt et de retrait.

Recette classe Immuable

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 .

- Une copie approfondie des objets doit être effectuée dans les méthodes getter pour renvoyer une copie plutôt que de renvoyer la référence réelle de l’objet).

Recette classe Immuable

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.
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 valeurs non immuables entrent en jeu.

```java
public class Point {
    private double x;
    private double y;
    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }
    public double getX() {
        return x;
    }
    public void setX(double x) {
        this.x = x;
    }
    public double getY() {
        return y;
    }
    public void setY(double y) {
        this.y = y;
    }
    public void move(double deltaX, double deltaY) {
        this.x += deltaX;
        this.y += deltaY;
    }
    @Override
    public String toString() {
        return "Point [x=" + x + ", y=" + y + "]";
    }
}
```

1- Attributs finaux

2-Valeurs fourni dans la méthode move non immuable-> copie

3-Pas de setters

4-Classe Final

```java
public final class ImmuablePoint {
    private final double x;
    private final double y;

    public ImmuablePoint(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public double getX() {
        return x;
    }

    public double getY() {
        return y;
    }

    public ImmuablePoint move(double deltaX, double deltaY) {
        return new ImmuablePoint(this.x + deltaX, this.y + deltaY);
    }

    @Override
    public String toString() {
        return "ImmutablePoint [x=" + x + ", y=" + y + "]";
    }
}
```

Quid des collections ?

Tableau java “**normal**” (c.a.d pas **ArrayList**) -> toujours modifiable.
-> utilisation de tableau dans classe immuable complique la classe (copies défensives des tab ext avant stockage et copie défensives des tab int avant de fournir à l’ext)

Tableau dynamique -> solution (unmodifiableList de java.util.Collections) = version non modifiable d’un tableau dynamique (toute methodes de modif lèvent l’exception)

UnsupportedOperationException. Exemple :

```
ArrayList<String> listOfStrings = new ArrayList<String>();
listOfStrings.add("un");listOfStrings.add("deux");
List<String> uListOfStrings = Collections.unmodifiableList(listOfStrings);
uListOfStrings.add("trois");
```

Quid des collections ?

```java
package fr.unilim.iut;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        ArrayList<String> listOfStrings = new ArrayList<String>();
        listOfStrings.add("un");listOfStrings.add("deux");
        List<String> uListOfStrings = Collections.unmodifiableList(listOfStrings);
        uListOfStrings.add("trois");
    }
}
```

unmodifiableList = moyen simple pour définir une classe immuable utilisant un tableau

les tableaux <- construction = copiés défensivement,rendus non modifiables au moyen de la méthode unmodifiableList puis stockés ainsi dans des attributs.
Ils sont retournés-> méthodes d'accès.

UnmodifiableList = type **List** (une interface) et pas ArrayList.

```
Exception in thread "main" java.lang.UnsupportedOperationException
        at java.base/java.util.Collections$UnmodifiableCollection.add(Collections.java:1067)
        at Employee/fr.unilim.iut.Main.main(Main.java:12)
```

Quid des collections ?

Collections.unmodifiableSet -> vue non modifiable -> “lecture seule” -> java.lang.UnsupportedOperationException

Collections.unmodifiableMap-> vue lecture seule carte (map) existante -> empêcher modification et permet aux autres parties ->accéder aux données sécurisée -> java.lang.UnsupportedOperationException

Collections.singletonList -> liste immuable d’un seul objet -> impossible d’ajouter/supprimer ->java.lang.UnsupportedOperationException

Collections.singletonMap -> carte (map) immuable contenant une seule paire clé-valeur. Non Modifiable après sa création -> java.lang.UnsupportedOperationException

Notion de record

Introduit par Java 14 en preview un nouveau type nommé Record décrit dans la JEP 359: Records

Record = nouveau type -> simplifier la création de classe qui encapsule simplement de manière immuable des données. Forme de restreintes de déclaration de classe similaire aux énumérations introduites en Java 5

Objectif : étendre la syntaxe -> création (plus simple) un type immuable qui encapsule des données

Déclaration minimaliste = informations utiles et nécessaire au compilateur -> génération de byte-code d'une classe

Les Records sont introduits en mode preview dans Java 14, ce qui implique que :

- la fonctionnalité peut changer voire même être retirée dans la ou les versions futures de Java,
- pour pouvoir les utiliser, il faut activer l'option --enable-preview des outils javac, java et jshell

Notion de record

```java
package fr.unilim.iut;

public record Livre(String titre, String auteur, int anneePublication) {

    public String description() {
        return titre + " par " + auteur + " (Publié en " + anneePublication + ")";
    }

    public static Livre createLivreParDefaut() {
        return new Livre("Titre Inconnu", "Auteur Inconnu", 0);
    }

    private static boolean anneeValide(int annee) {
        return annee >= 0 && annee <= 2023;
    }

    public Livre {
        if (!anneeValide(anneePublication)) {
            throw new IllegalArgumentException("Année de publication invalide.");
        }
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        Livre livre1 = new Livre("La Belle et la Bête", "Billy Kimber", 1979);

        Livre livre2 = Livre.createLivreParDefaut();
        System.out.println(livre1.description());
        System.out.println(livre2.description());
    }
}
```

Propriétés record -> automatiquement déclarées final et sont en lecture seule -> aucune modification possible après la création de l'objet.

Principalement utilisés pour représenter des données immuables (valeurs simple).

Simplifient la création de classes pour de telles structures de données évitant la rédaction fastidieuse de méthodes d'accès, equals, hashCode, et toString -> facilite la création de code propre et robuste.

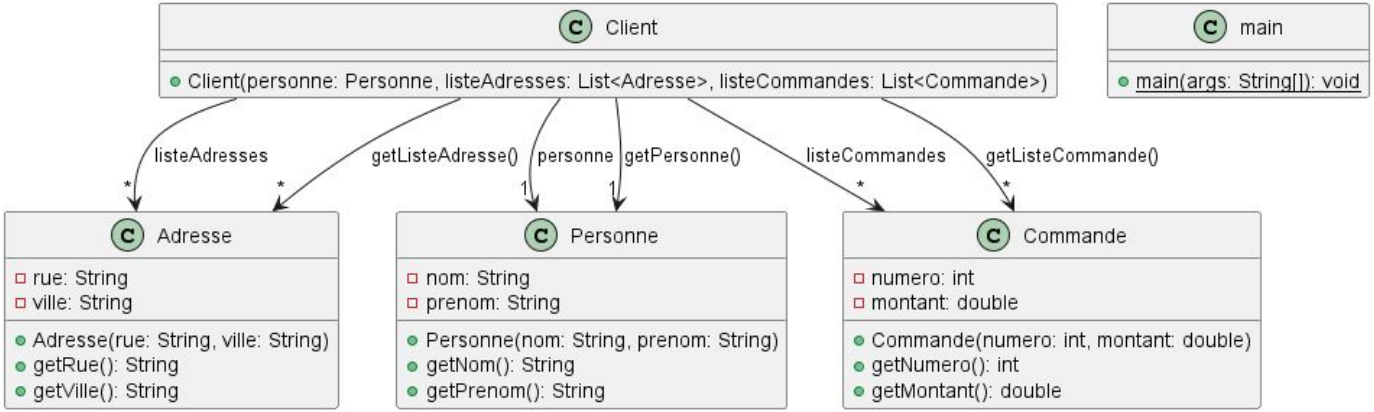
Lien entre record et immuabilité

En résumé, un record en Java est étroitement lié à l'immuabilité, car il est conçu pour stocker des données de manière sécurisée et prévisible.

Les champs finaux, les méthodes d'accès en lecture et les méthodes de comparaison automatiquement générées encouragent les bonnes pratiques de programmation axées sur l'immuabilité.

Cependant, il est essentiel de noter que bien que les records encouragent l'immuabilité, il est possible de les rendre mutables en utilisant la réflexion Java, mais cela va à l'encontre de leur intention initiale de stocker des données de manière immuable.

Live Coding


Client
Client(personne: Personne, listeAdresses: List<Adresse>, listeCommandes: List<Commande>)
main
main(args: String[]): void
listeAdresses
getListeAdresse()
personne
getPersonne()
listeCommandes
getListeCommande()
Adresse
rue: String
ville: String
Adresse(rue: String, ville: String)
getRue(): String
getVille(): String
Personne
nom: String
prenom: String
Personne(nom: String, prenom: String)
getNom(): String
getPrenom(): String
Commande
numero: int
montant: double
Commande(numero: int, montant: double)
getNumero(): int
getMontant(): double

QCM

https://create.kahoot.it/share/immuabilite-du-code/99ee2cba-6e0f-480e-9652-35d3b3d2758f

Sources

- https://www.devuniversity.com/blog/immutable-vs-mutable-un-concept-majeur-en-programmation
- https://blog.oxiane.com/2020/05/19/java-14-les-records/#:~:text=Les%20records%20sont%20un%20nouveau,%C3%A9num%C3%A9rations%20introduites%20en%20Java%205
- https://gfx.developpez.com/tutoriel/java/immuables/
- https://www.ukonline.be/cours/java/apprendre-java/chapitre5-6
- https://fr.wikipedia.org/wiki/Objet_immuable
- https://www.moquillon.fr/index.php/post/2017/03/11/L-immutabilit%C3%A9-en-Java
- https://www.jmdoudoux.fr/java/dej/chap-records.htm