

# **VERS UN CODE IMMUTABLE**

SALMON ALEXIS  
GUY-SOULARD GABRIEL  
HOLLANDE BENJAMIN  
SIP KYLLIAN

# SOMMAIRE

**01**

QU'EST CE QU'UN CODE  
IMMUABLE ?

**02**

L'INTÉRÊT D'UN CODE  
IMMUABLE

**03**

COMMENT RENDRE UNE CLASSE  
IMMUABLE

**04**

COLLECTIONS IMMUABLES

**05**

LES RECORDS

**06**

LES RECORDS ET LEUR LIEN  
AVEC L'IMMUABILITÉ

**07**

LIVE CODING

**08**

CONCLUSION

**09**

BIBLIOGRAPHIE

# **QU'EST CE QU'UN CODE IMMUABLE**

**UN OBJET DIT IMMUABLE EST UNE INSTANCE DE CLASSE DONT LES  
MEMBRES EXPORTÉS NE PEUVENT ÊTRE MODIFIÉS APRÈS CRÉATION.**

# L'INTÉRÊT D'UN CODE IMMUABLE

- ILS SONT GARANTIS THREAD-SAFE
- ILS PEUVENT ÊTRE MIS EN CACHE
- ILS N'ONT BESOIN NI DE CONSTRUCTEUR PAR COPIE NI D'IMPLEMENTATION DE L'INTERFACE CLONEABLE
- LEURS INVARIANTS SONT TESTÉS À LA CRÉATION SEULEMENT
- ILS CONSTITUENT D'EXCELLENTES CLÉS POUR LES MAP ET SET

# EXAMPLE

```
public class Person {  
    private String name;  
    private int age;  
    private List<String> friends;  
  
    public Person(String name, int age, List<String> friends) {  
        this.name = name;  
        this.age = age;  
        this.friends = friends; // Pas de copie défensive  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    public int getAge() {  
        return age;  
    }  
  
    public void setAge(int age) {  
        this.age = age;  
    }  
  
    public List<String> getFriends() {  
        return friends;  
    }  
  
    public void setFriends(List<String> friends) {  
        this.friends = friends;  
    }  
}
```

# EXAMPLE

## Person

```
- name : name  
- age : int  
- friends : List<String>  
  
+ Person(name: String, age: int, friends : List<String>)  
+ getName(): String  
+ setName(name: String): void  
+ getAge(): int  
+ setAge(age: int): void  
+ getFriends(): List<String>  
+ setFriends(friends: List<String>): void
```

# EXAMPLE

```
public static void main(String[] args) {  
    List<String> friends = new ArrayList<>();  
    friends.add("Alice");  
    friends.add("Bob");  
  
    Person person = new Person("John", 30, friends);  
  
    // Modification des attributs après la création  
    person.setName("Mike");  
    person.setAge(35);  
  
    // Modification directe de la liste des amis  
    person.getFriends().add("Charlie");  
  
    System.out.println(person.getName());      // Affiche "Mike"  
    System.out.println(person.getAge());        // Affiche "35"  
    System.out.println(person.getFriends());    // Affiche [Alice, Bob, Charlie]  
}
```

# COMMENT RENDRE UNE CLASSE IMMUABLE

- DECLARER LA CLASSE FINAL
- RENDRE TOUS LES CHAMPS PRIVÉS ET FINAUX
- INITIALISER LES CHAMPS DANS UN CONSTRUCTEUR
- PAS DE SETTER
- FOURNIR UNIQUEMENT DES MÉTHODES "GETTER" POUR ACCÉDER AUX CHAMPS
- FAIRE DES COPIES DANS LE CONSTRUCTEUR SI NÉCESSAIRE
- RETOURNEZ DES COPIES POUR LES OBJETS MUTABLES

# COMMENT RENDRE UNE CLASSE IMMUABLE

```
import java.util.ArrayList;
import java.util.List;

public final class Person {
    private final String name;
    private final int age;
    private final List<String> friends;

    public Person(String name, int age, List<String> friends) {
        this.name = name;
        this.age = age;
        // Copie défensive pour éviter que la liste mutable soit modifiée
        this.friends = new ArrayList<>(friends);
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    public List<String> getFriends() {
        return new ArrayList<>(friends);
    }

    // Pas de setter, la classe est immuable
}
```

# COMMENT RENDRE UNE CLASSE IMMUBLE

<<Final>>  
Person

- name : name <<Final>>  
- age : int <<Final>>  
- friends : List<String> <<Final>>

+ Person(name: String, age: int, friends : List<String>)  
+ getName(): String  
+ getAge(): int  
+ getFriends(): List<String>

# COMMENT RENDRE UNE CLASSE IMMUABLE

```
public static void main(String[] args) {  
    List<String> friends = new ArrayList<>();  
    friends.add("Alice");  
    friends.add("Bob");  
  
    Person person = new Person("John", 30, friends);  
  
    // La tentative de modifier la liste originale n'affectera pas la classe Person  
    friends.add("Charlie");  
  
    // La liste à l'intérieur de Person n'a pas changé  
    System.out.println(person.getFriends()); // Affiche [Alice, Bob]  
  
    // Impossible de modifier l'état de l'objet Person  
    // person.setName("NewName"); -> Erreur de compilation  
}
```

# COLLECTIONS IMMUABLES

- UTILISATION DE COLLECTIONS.UNMODIFIABLE()

```
public static void main(String[] args) {  
    import java.util.Collections;  
    import java.util.List;  
    import java.util.ArrayList;  
  
    List<String> mutableList = new ArrayList<>();  
    mutableList.add("A");  
    mutableList.add("B");  
  
    // Créer une collection immuable à partir de mutableList  
    List<String> immutableList = Collections.unmodifiableList(mutableList);  
  
    // Tentative de modification provoquera une exception  
    // immutableList.add("C"); // Throws UnsupportedOperationException  
}
```

# COLLECTIONS IMMUABLES

- UTILISATION DES MÉTHODES LIST.OF(), SET.OF(), MAP.OF()

```
public static void main(String[] args) {  
    import java.util.List;  
    import java.util.Set;  
    import java.util.Map;  
  
    // Créer une liste immuable  
    List<String> immutableList = List.of("A", "B", "C");  
  
    // Créer un ensemble immuable  
    Set<String> immutableSet = Set.of("X", "Y", "Z");  
  
    // Créer une map immuable  
    Map<Integer, String> immutableMap = Map.of(1, "One", 2, "Two", 3, "Three");  
  
    // Tentative de modification lève une exception  
    // immutableList.add("D"); // UnsupportedOperationException  
}
```

# LES RECORDS EN JAVA

## INTRODUCTION AUX RECORDS :

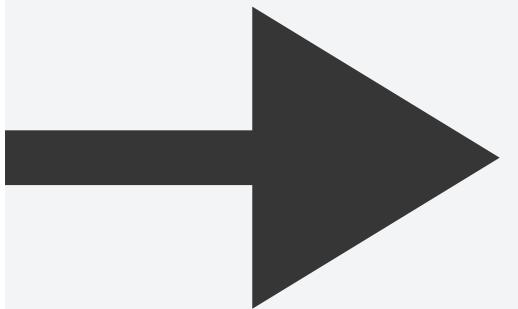
### RECORDS :

- Fonctionnalité introduite dans Java à partir de la version 14,
- Avec pour objectif de simplifier la création de classes immuables,
- Conteneur de données immutable

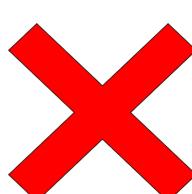
# LES RECORDS EN JAVA

## EXEMPLE :

```
public class Point {  
    private final int x;  
    private final int y;  
  
    public Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    public int getX() {  
        return x;  
    }  
  
    public int getY() {  
        return y;  
    }  
  
    @Override  
    public boolean equals(Object o) {  
        if (this == o) return true;  
        if (o == null || getClass() != o.getClass()) return false;  
        Point point = (Point) o;  
        return x == point.x && y == point.y;  
    }  
  
    @Override  
    public int hashCode() {  
        return Objects.hash(x, y);  
    }  
  
    @Override  
    public String toString() {  
        return "Point{x=" + x + ", y=" + y + "}";  
    }  
}
```



```
public record Point(int x, int y) {}
```



# POURQUOI EST-CE LIÉ À L'IMMUABILITÉ ?

- Naturellement immuables
- Tous les champs sont déclarés comme final.
- Ils simplifient la création de classes immuables, réduisant ainsi le code "boilerplate" nécessaire.

```
Point point = new Point(3, 5);
```

# LIVE CODING

**<https://youtu.be/dl6xjPjFgeM>**

STRATEGY N°3

# CONCLUSION



# BIBLIOGRAPHIE

- <https://www.devuniversity.com/blog/immutable-vs-mutable-un-concept-majeur-en-programmation>
- Immuabilité Java (youtube.com)
- Thread safety — Wikipédia (wikipedia.org)
- Classes et objets immuables. (developpez.com)
- Valeurs fixes en Java : finales, constantes et immuables (codegym.cc)
- Comment créer une classe immutable en Java - toptips.fr
- <https://claude.ai/>
- <https://blog.oxiane.com/2020/05/19/java-14-les-records/>
- <https://www.jmdoudoux.fr/java/dej/chap-records.htm>
- <https://chatgpt.com/>