



# L'Immutabilité en Java

# Qu'est-ce que l'immutabilité ?

## Définition

Un objet dont l'état ne peut pas changer après sa création.  
Aucune méthode ne modifie les champs internes.

## Différence clé

Objets mutables : modifiables après création. Objets immuables : valeurs constantes tout au long du cycle de vie.

### État constant

Après création

### Aucune modification

Pas de méthodes mutantes

### Thread-safe

Par nature

### Partage sécurisé

Entre threads

# Les avantages de l'immutabilité



## Sécurité thread-safe

Naturellement thread-safe sans synchronisation, éliminant les problèmes de concurrence.



## Possibilité de cache

Objets mis en cache et réutilisés sans risque, améliorant les performances.



## Simplicité de conception

Code plus facile à comprendre et maintenir car l'état ne change jamais de manière inattendue.



## Moins d'erreurs

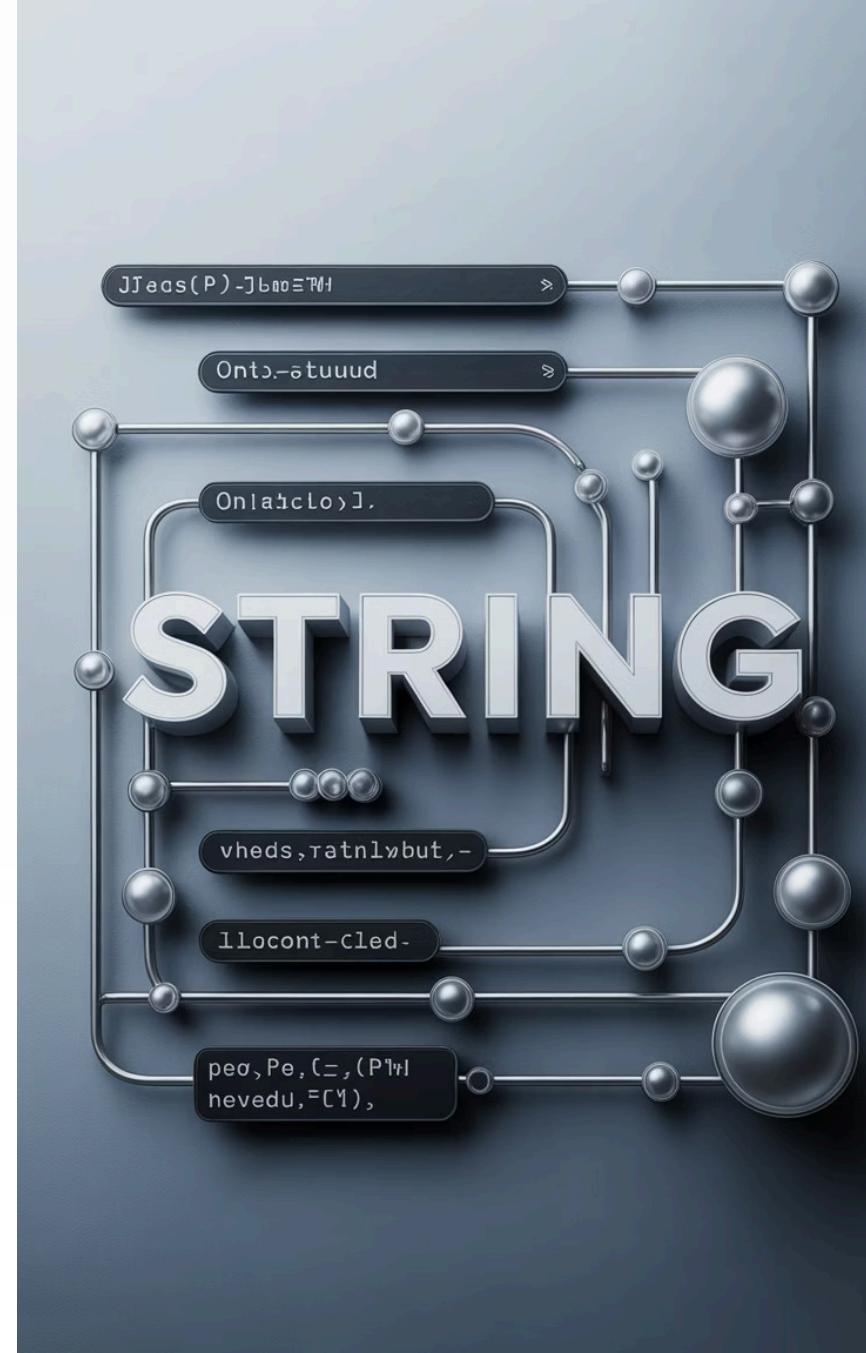
Élimine toute une catégorie de bugs liés aux modifications d'état non contrôlées.

# String : l'exemple parfait

La classe String est l'exemple le plus connu d'objet immuable en Java. Chaque opération crée une nouvelle instance.

« L'immutabilité de String permet à Java d'optimiser le stockage via le String Pool, où les chaînes identiques partagent la même instance mémoire. »

Exemple : `toUpperCase()` retourne une nouvelle String, l'original reste inchangé.



# Les principes fondamentaux

01

## Déclarer la classe final

Empêche l'héritage qui pourrait introduire de la mutabilité

02

## Rendre tous les champs final

Garantit que les champs ne peuvent être assignés qu'une seule fois

03

## Champs privés uniquement

Évite l'accès direct et la modification des champs internes

04

## Pas de méthodes setters

Aucune méthode ne doit permettre de modifier l'état de l'objet

05

## Copie défensive

Créer des copies des objets mutables passés au constructeur

06

## Retours immutables

Les getters doivent retourner des copies ou des objets immuables



# Anatomie d'une classe immuable

## Règles de conception

- Classe déclarée **final** pour empêcher l'héritage
- Tous les champs **private final**
- Aucune méthode modifiant l'état interne
- Constructeur initialise tous les champs définitivement

## Copies défensives

Le constructeur crée des copies des objets mutables (collections, dates).

Les getters retournent des copies défensives pour les champs mutables.

# Exemple AVANT : Classe Person mutable

```
public class Person {  
    private String name;  
    private int age;  
    private Date birthDate;  
    private List<String> hobbies;  
  
    public Person(String name, int age, Date birthDate, List<String> hobbies) {  
        this.name = name;  
        this.age = age;  
        this.birthDate = birthDate;  
        this.hobbies = hobbies;  
    }  
  
    // Setters permettent la modification  
    public void setName(String name) { this.name = name; }  
    public void setAge(int age) { this.age = age; }  
  
    // Getters exposent les objets mutables  
    public Date getBirthDate() { return birthDate; }  
    public List<String> getHobbies() { return hobbies; }  
}
```

- ❑ **Problèmes identifiés :** Classe entièrement mutable. Les setters permettent de modifier l'état, et les getters exposent des références directes à des objets mutables (Date, List).

# Exemple APRÈS : Classe Person immuable

```
public final class Person {  
    private final String name;  
    private final int age;  
    private final Date birthDate;  
    private final List<String> hobbies;  
  
    public Person(String name, int age, Date birthDate, List<String> hobbies) {  
        this.name = name;  
        this.age = age;  
        // Copie défensive pour Date  
        this.birthDate = new Date(birthDate.getTime());  
        // Copie défensive pour List  
        this.hobbies = new ArrayList<>(hobbies);  
    }  
  
    public String getName() { return name; }  
    public int getAge() { return age; }  
  
    // Retourne des copies pour préserver l'immutabilité  
    public Date getBirthDate() { return new Date(birthDate.getTime()); }  
    public List<String> getHobbies() { return new ArrayList<>(hobbies); }  
}
```

- ❑ **Améliorations :** Classe final, champs final, pas de setters, copies défensives dans le constructeur et les getters.

# Impact sur la sécurité du code

## Avec mutabilité

```
Person person = new Person(...);  
// Quelqu'un peut modifier  
person.setAge(100);  
person.getHobbies().clear();
```

```
Date date = person.getBirthDate();  
date.setYear(1900);  
// Modifie l'objet interne !
```

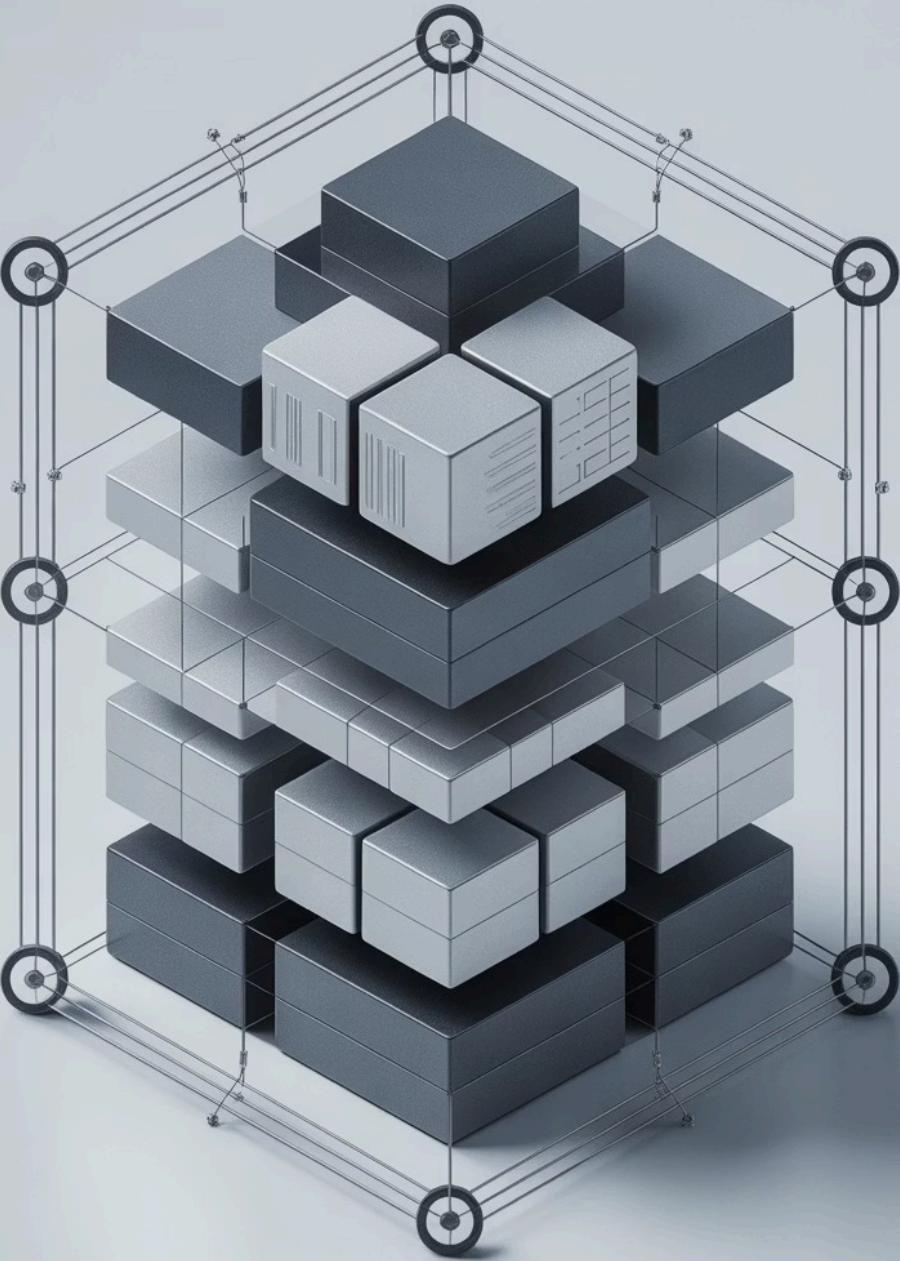
Code mutable permet des modifications imprévisibles, bugs difficiles à détecter en multi-thread.

## Avec immutabilité

```
Person person = new Person(...);  
// Impossible de modifier  
// person.setAge(100); N'existe pas  
person.getHobbies().clear();  
// N'affecte pas l'original
```

```
Date date = person.getBirthDate();  
date.setYear(1900);  
// N'affecte pas l'objet interne
```

L'immutabilité garantit un état cohérent, éliminant effets de bord et bugs.



# Pattern : Builder pour objets immuables

Le pattern Builder crée des objets immuables complexes avec de nombreux paramètres. API fluide et lisible tout en maintenant l'immutabilité.



## Builder mutable

Pendant construction

## Méthode build()

Validation centralisée

## Objet final immuable

Garantie d'immutabilité

Résout le "telescoping constructor anti-pattern" avec une construction claire étape par étape.

# Implémentation du Builder Pattern

```
public final class Person {  
    private final String name;  
    private final int age;  
    private final String email;  
    private final String phone;  
  
    private Person(Builder builder) {  
        this.name = builder.name;  
        this.age = builder.age;  
        this.email = builder.email;  
        this.phone = builder.phone;  
    }  
  
    public static class Builder {  
        private String name;  
        private int age;  
        private String email;  
        private String phone;  
  
        public Builder name(String name) {  
            this.name = name;  
            return this;  
        }  
  
        public Builder age(int age) {  
            this.age = age;  
            return this;  
        }  
  
        public Person build() {  
            if (name == null || age < 0) {  
                throw new IllegalStateException("Invalid person data");  
            }  
            return new Person(this);  
        }  
    }  
}
```

# Utilisation élégante du Builder

## Construction fluide

```
Person person = new Person.Builder()  
    .name("Alice Dupont")  
    .age(30)  
    .email("alice@example.com")  
    .phone("+33 6 12 34 56 78")  
    .build();
```

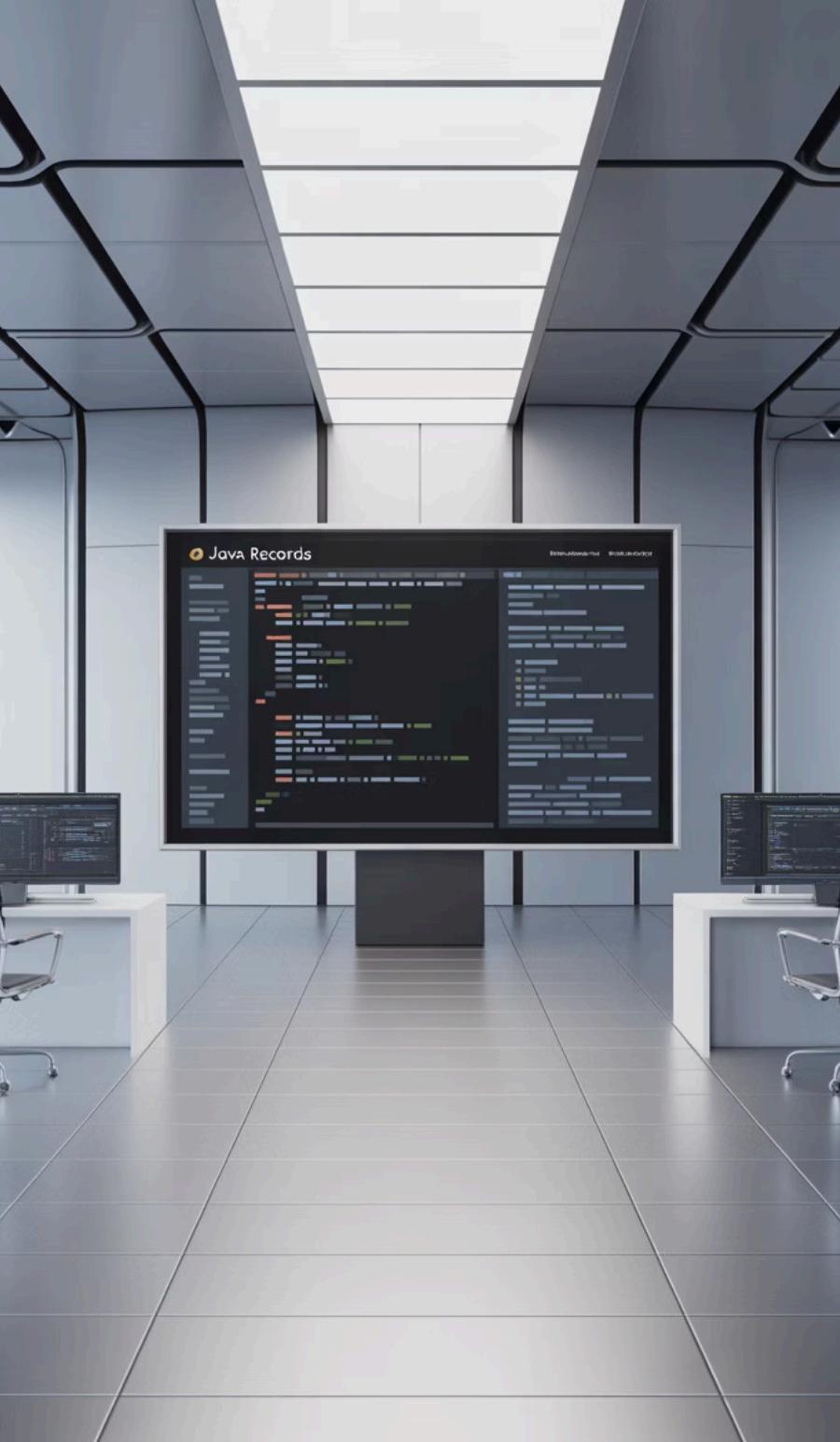
API fluide rend le code lisible et expressif. Chaînage des appels.

## Paramètres optionnels

```
Person person = new Person.Builder()  
    .name("Bob Martin")  
    .age(45)  
    .build();
```

Paramètres optionnels omis facilement. Builder gère valeurs par défaut et validation.

« Le pattern Builder offre le meilleur des deux mondes : flexibilité pendant la construction et immutabilité garantie pour l'objet final. »



# Pattern : Records Java (depuis Java 14)

Java 14 introduit les Records : classes spéciales pour porteurs de données immuables. Le compilateur génère automatiquement constructeur, getters, equals(), hashCode() et toString().



## Moins de code

Élimine le boilerplate traditionnel



## Immutable par défaut

Implicitement final, champs final



## Solution moderne

Idiomatique pour DTO et objets domaine

# Records : Avant et Après

## AVANT (classe traditionnelle)

```
public final 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 instanceof Point)) 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 + "]";  
}
```

## APRÈS (Record)

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

**C'est tout !** Le compilateur génère automatiquement tout le code nécessaire.

### Avantages :

- 90% moins de code
- Pas d'erreurs de copier-coller
- Intention claire et concise
- Maintenance simplifiée

# Records avec validation personnalisée

```
public record Email(String address) {  
    public Email {  
        if (address == null || !address.contains("@")) {  
            throw new IllegalArgumentException("Invalid email address");  
        }  
        // Normalisation  
        address = address.toLowerCase().trim();  
    }  
}  
  
public record Temperature(double value, String unit) {  
    public Temperature {  
        if (unit == null || (!unit.equals("C") && !unit.equals("F"))) {  
            throw new IllegalArgumentException("Unit must be C or F");  
        }  
        if (value < -273.15 && unit.equals("C")) {  
            throw new IllegalArgumentException("Temperature below absolute zero");  
        }  
    }  
}  
  
public Temperature toCelsius() {  
    if (unit.equals("C")) return this;  
    return new Temperature((value - 32) * 5/9, "C");  
}
```

- ❑ **Constructeur compact :** Les Records supportent un constructeur compact permettant validation et transformation sans répéter les paramètres.

# Anti-Patterns

Pièges courants à éviter lors de la création de classes immuables



# Anti-Pattern : Exposition de collections mutables

Piège le plus courant : exposer des références directes à des collections mutables. Même avec champ final, le contenu peut être modifié.

## Code problématique

```
public final class ShoppingCart {  
    private final List<Item> items;  
  
    public ShoppingCart(List<Item> items) {  
        this.items = items; // DANGER !  
    }  
  
    public List<Item> getItems() {  
        return items; // DANGER !  
    }  
  
    // Utilisation  
    List<Item> myItems = new ArrayList<>();  
    myItems.add(new Item("Book"));  
    ShoppingCart cart = new ShoppingCart(myItems);  
  
    // L'immutabilité est brisée !  
    myItems.clear();  
    cart.getItems().add(new Item("Evil"));
```

## Solution correcte

```
public final class ShoppingCart {  
    private final List<Item> items;  
  
    public ShoppingCart(List<Item> items) {  
        // Copie défensive  
        this.items = List.copyOf(items);  
    }  
  
    public List<Item> getItems() {  
        // Retourne une vue immuable  
        return items;  
    }  
  
    // Utilisation  
    List<Item> myItems = new ArrayList<>();  
    myItems.add(new Item("Book"));  
    ShoppingCart cart = new ShoppingCart(myItems);  
  
    myItems.clear(); // N'affecte pas cart  
    // cart.getItems().add(...);  
    // Lance UnsupportedOperationException
```



# Collections immuables en Java

1

## `Collections.unmodifiableList`

Vue immutable, mais liste originale peut changer

2

## `List.copyOf`

Copie immutable, isolée de l'original

3

## `List.of`

Création directe de liste immutable

4

## `Guava ImmutableList`

Plus de fonctionnalités, builder pattern

# Anti-Pattern : Héritage compromettant l'immédiateté

Classe immuable non-final peut être étendue. Une sous-classe peut introduire mutabilité en ajoutant champs mutables ou surchargeant méthodes.

## Classe parent "immuable"

```
// PAS FINAL - ERREUR !
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; }
}
```

## Sous-classe brisant l'immédiateté

```
public class MutablePoint extends Point {
    private int x;
    private int y;

    public MutablePoint(int x, int y) {
        super(x, y);
        this.x = x;
        this.y = y;
    }

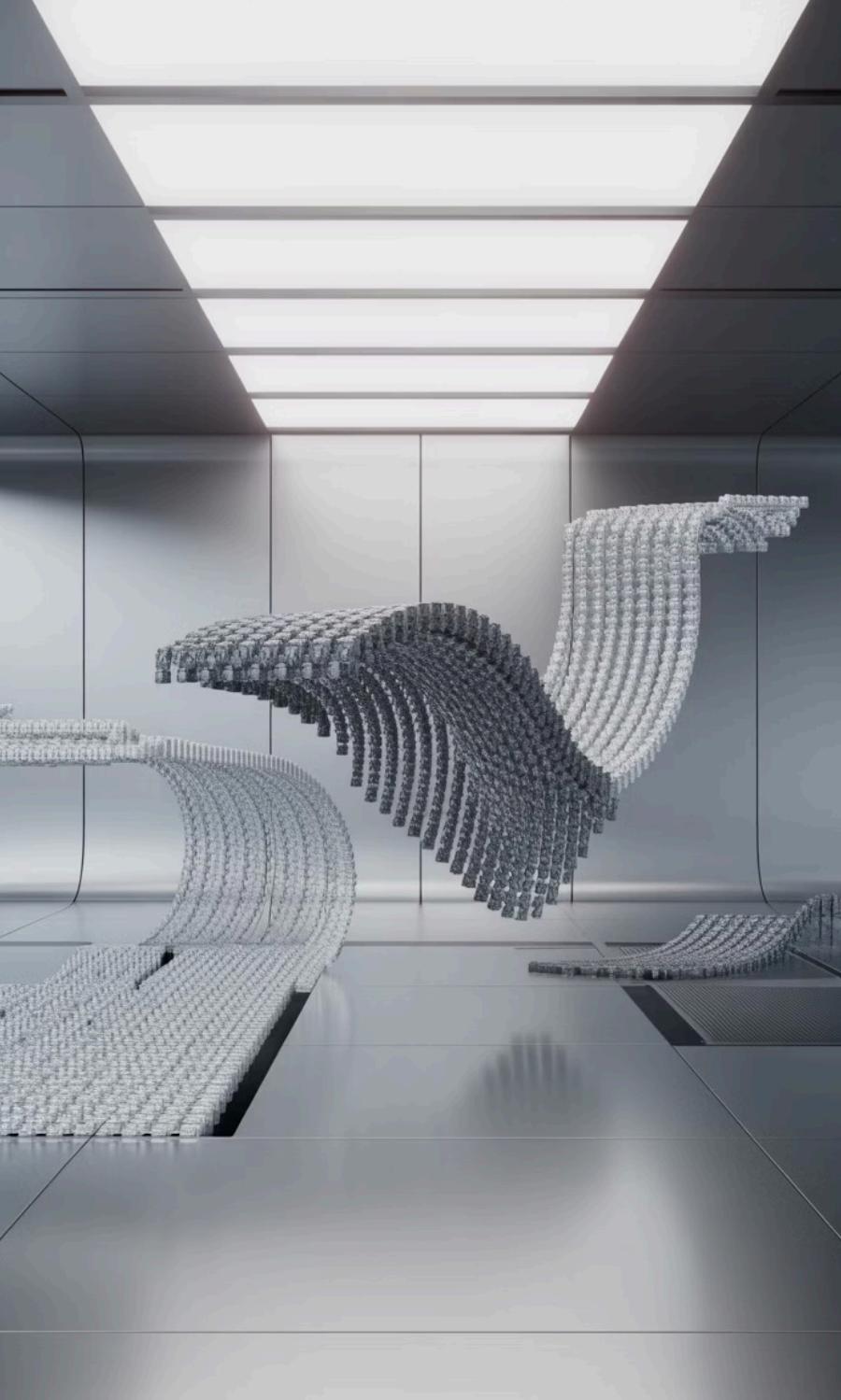
    @Override
    public int getX() { return x; }

    @Override
    public int getY() { return y; }

    public void setX(int x) { this.x = x; }
    public void setY(int y) { this.y = y; }
}

// Le polymorphisme rend cela dangereux
Point p = new MutablePoint(1, 2);
((MutablePoint) p).setX(10); // Modifié !
```

- ☐ **Solution :** Toujours déclarer les classes immuables comme **final** pour empêcher l'héritage.



# Pattern : Objets immuables avec modifications

Comment "modifier" un objet immuable ? Créer un nouvel objet basé sur l'ancien avec les modifications souhaitées. Pattern "copy-and-modify".



Méthodes préfixées par "with" indiquent création d'une variante de l'objet.

# Méthodes "with" pour modification immuable

```
public final class Person {  
    private final String name;  
    private final int age;  
    private final String email;  
  
    public Person(String name, int age, String email) {  
        this.name = name;  
        this.age = age;  
        this.email = email;  
    }  
  
    // Méthodes "with" retournant de nouvelles instances  
    public Person withName(String name) {  
        return new Person(name, this.age, this.email);  
    }  
  
    public Person withAge(int age) {  
        return new Person(this.name, age, this.email);  
    }  
  
    public Person withEmail(String email) {  
        return new Person(this.name, this.age, email);  
    }  
  
    public Person withIncrementedAge() {  
        return new Person(this.name, this.age + 1, this.email);  
    }  
}
```

**Utilisation :** Person aliceNewEmail = alice.withEmail("alice.new@example.com");

# Immutabilité et performance

0

1x

100%

## Synchronisation

Coût de synchronisation pour objets immuables - thread-safe par nature

## Allocation mémoire

Facteur d'allocation - création d'objets modernes en Java extrêmement rapide

Coût de performance négligeable, voire amélioration dans applications multi-thread. JVM moderne optimise efficacement l'allocation d'objets courts-vivants.

## Sécurité garantie

Élimination complète des bugs de concurrence liés à l'état partagé mutable



# Immutabilité dans les architectures modernes

## Microservices

Objets immuables parfaits pour DTO, garantissant intégrité des données lors du transfert entre services.

## Event Sourcing

Événements immuables garantissent audit trail fiable et reconstruction d'état cohérente.

## Programmation Réactive

Immutabilité fondamentale dans frameworks réactifs comme Reactor et RxJava.

## Caching Distribué

Objets immuables idéaux pour cache, partagés entre threads et serveurs sans synchronisation.

# Cas pratique : Système de gestion de commandes

Système e-commerce gérant commandes, articles, totaux, remises et historique. L'immutabilité améliore conception, sécurité et maintenabilité.

## Problème avec mutabilité

Bugs de concurrence, modifications accidentnelles d'état, difficultés de débogage et d'audit

## Solution avec immutabilité

Builder pattern, méthodes "with", système sûr, testable et facile à maintenir



# Principes clés à retenir

## Immutabilité = Sécurité

Élimine bugs de concurrence et modifications d'état imprévues. Code robuste et prévisible.

## Règles de conception strictes

Classe final, champs final, pas de setters, copies défensives, retours immuables.

## Patterns modernes

Records Java, pattern Builder, méthodes "with" pour APIs élégantes et immuables.

## Attention aux collections

List.copyOf(), Set.of(), éviter exposition de collections mutables. Piège le plus fréquent.

## Performance acceptable

Coût négligeable comparé aux bénéfices en maintenabilité et sécurité.

## Adopter progressivement

Commencer par DTOs, objets de domaine et configuration. Immutabilité totale pas toujours nécessaire.

« L'immutabilité transforme la complexité accidentelle en simplicité essentielle, rendant votre code plus facile à comprendre, à tester et à maintenir. »