

# La Classe Object et ses Méthodes en Java



# Object : Fondation du Système de Types

Racine de la hiérarchie des classes Java. Toutes les classes héritent d'Object, bénéficiant de ses méthodes fondamentales.

Comprendre ces méthodes est **essentiel** : elles définissent comparaison, hash codes, représentation textuelle et clonage.

01

---

## equals()

Comparaison d'égalité

02

---

## hashCode()

Calcul de hash

03

---

## toString()

Représentation textuelle

04

---

## clone()

Copie d'objets



# equals() : Fondements

Définit l'égalité entre objets. Par défaut : égalité de référence (==). En pratique : égalité basée sur le **contenu**.

## Crucial pour Collections

HashSet, HashMap, ArrayList dépendent d>equals() pour recherche et élimination des doublons

## Contrat Strict

5 propriétés obligatoires : réflexivité, symétrie, transitivité, cohérence, comparaison null

# Pattern : Implémentation Correcte d>equals()



## Vérification de référence

this == obj pour optimiser cas identiques



## Vérification de null

obj == null retourne false



## Vérification de type

instanceof pour gérer sous-classes



## Cast et comparaison

Objects.equals() pour champs significatifs

```
@Override  
public boolean equals(Object obj) {  
    if (this == obj) return true;  
    if (obj == null) return false;  
    if (!(obj instanceof Person)) return false;  
    Person other = (Person) obj;  
    return Objects.equals(name, other.name) &&  
        Objects.equals(age, other.age);  
}
```



# Antipattern : Violations du Contrat

## Violation de symétrie

`parent.equals(enfant) ≠ enfant.equals(parent)` brise le contrat

## Oubli de vérification null

`NullPointerException` au lieu de retourner `false`

## Comparaison `getClass()`

Empêche polymorphisme et viole principe de substitution

## Champs mutables

Objets perdus dans `HashSet` après modification

# hashCode() : Principe Fondamental

Retourne un entier représentant un "code de hachage". Utilisé par HashMap, HashSet, Hashtable pour organiser et retrouver objets efficacement.

**Distribution uniforme** minimise collisions et optimise performances.



- ❑ **Contrat critique :** Si equals() retourne true, hashCode() DOIT être identique. Violer ce contrat produit bugs difficiles à diagnostiquer.



# Le Contrat Sacré

1

**equals() retourne true**

Deux objets considérés égaux selon logique métier

2

**hashCode() DOIT être identique**

Même code de hachage obligatoire

3

**Placement dans HashMap**

Objets stockés et retrouvés correctement

Ce contrat n'est **pas optionnel** - c'est une exigence absolue du langage Java.

# Pattern : Implémentation Robuste de hashCode()

## Approche Manuelle

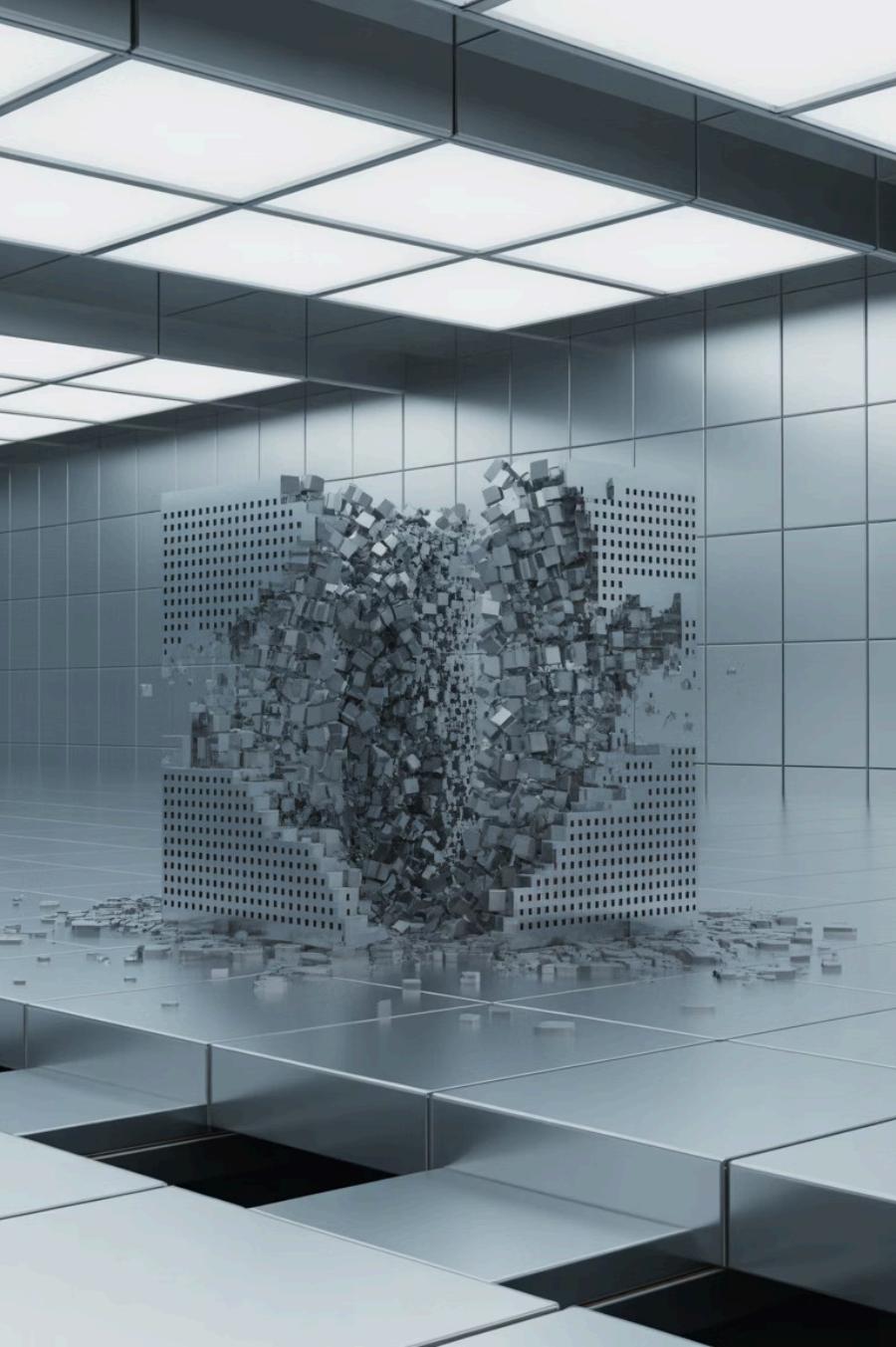
```
@Override  
public int hashCode() {  
    int result = 17;  
    result = 31 * result +  
        (name != null ?  
            name.hashCode() : 0);  
    result = 31 * result + age;  
    return result;  
}
```

Méthode traditionnelle avec nombre premier (31) pour combiner hash codes

## Approche Moderne (Java 7+)

```
@Override  
public int hashCode() {  
    return Objects.hash(  
        name, age, email  
    );  
}
```

Recommandé : simplifie code, garantit algorithme correct, gère null automatiquement



# Antipatterns hashCode() à Éviter

## Retourner une constante

Retourner toujours 0 ou 1  
détruit performances :  
complexité passe de O(1) à O(n)

## Utiliser Random

Hash aléatoire viole contrat de cohérence : appels multiples doivent retourner même valeur

## Inclure champs exclus d>equals()

Objets égaux auront hash codes différents, violant le contrat

# Performance et Distribution

$O(1)$

**Complexité HashMap**

Avec bon hashCode(), opérations en temps constant

**31**

**Nombre Premier**

Excellent compromis distribution/performance

**2-5%**

**Gain Lazy Init**

Cacher hashCode pour objets immutables

Qualité d'un algorithme se mesure par sa **distribution uniforme**. Nombres premiers comme 31 offrent bonne distribution statistique.

Pour objets immutables : pattern de lazy initialization avec mise en cache. Attention : ne jamais cacher pour objets mutables.

# toString(): Représentation Textuelle

Fournit représentation textuelle pour débogage et logging. Implémentation par défaut : nom classe + hash code hexadécimal (rarement utile).

Bonne implémentation : **concise, informative, lisible.** Inclure champs significatifs sans verbosité excessive.

- Pas de contrat strict
- Modifiable sans casser application
- Uniquement pour affichage



# Patterns pour `toString()` Efficace

## Format Simple et Lisible

```
"Person{name=\"" + name +  
    "\", age=" + age + "}"
```

Concaténation traditionnelle, simple mais crée objets String temporaires

## StringBuilder pour Performance

```
new StringBuilder()  
.append("Person{name=\"")  
.append(name)  
.toString()
```

Plus performant pour nombreux champs, évite Strings intermédiaires

## Approche Moderne avec Helpers

```
String.format(  
    "Person{name='%s', age=%d}",  
    name, age  
)
```

Code lisible et maintenable avec `String.format()`

# Bonnes Pratiques `toString()`



## Champs Pertinents

Sélectionner champs définissant identité ou état important. Exclure métadonnées internes.



## Éviter Boucles Infinies

Attention références circulaires : A→B→A cause StackOverflowError. Limiter profondeur.



## Protéger Données Sensibles

Jamais mots de passe, tokens en clair. Utiliser masques (\*\*\*\*) pour sécurité logs.



## Optimiser si Nécessaire

Limiter sortie pour collections volumineuses : "... (X more items)"

# Antipatterns `toString()` Dangereux



## Calculs Lourds

Jamais calculs coûteux ou appels base de données.  
Débogueurs appellent automatiquement, ralentit application.



## Modification d'État

`toString()` doit être lecture seule. Ne jamais modifier état objet ou dépendances.



## Lancer des Exceptions

Éviter exceptions. Gérer null gracieusement avec "field=null" plutôt que crasher.



# clone() : Complexité et Pièges

Méthode la plus controversée d'Object. Design présente problèmes fondamentaux.

Nombreux experts recommandent d'**éviter complètement** clone() en faveur d'alternatives plus sûres.



## Interface Cloneable requise

Sans elle, CloneNotSupportedException



## Copie superficielle par défaut

Objets internes partagés entre original et clone



## Copie profonde complexe

Clonage récursif fragile et source de bugs



# Antipattern : Utilisation Naïve de clone()

```
// MAUVAIS : Copie superficielle problématique
public class Person implements Cloneable {
    private String name;
    private List<String> hobbies;

    @Override
    public Person clone() {
        try {
            return (Person) super.clone(); // Shallow copy!
        } catch (CloneNotSupportedException e) {
            throw new AssertionError();
        }
    }
}
```

- ❑ **Problème majeur :** p1 et p2 partagent la même liste hobbies.  
Modifier via p2 affecte aussi p1 !

# Pattern : Copie Profonde vs Alternative

## Implémentation Manuelle

```
@Override  
public Person clone() {  
    try {  
        Person cloned =  
            (Person) super.clone();  
        // Deep copy mutables  
        cloned.hobbies =  
            new ArrayList<>(this.hobbies);  
        return cloned;  
    } catch (...) {  
        throw new AssertionError();  
    }  
}
```

## Alternative Recommandée

```
// Constructeur de copie  
public Person(Person other) {  
    this.name = other.name;  
    this.age = other.age;  
    this.hobbies =  
        new ArrayList<>(other.hobbies);  
}  
  
// Utilisation  
Person p2 = new Person(p1);
```

Préféré : explicite, type-safe, pas de try-catch

# Alternatives Modernes à clone()

## Constructeur de Copie

Pattern le plus recommandé.  
Explicite, type-safe, sans  
exceptions checked.

## Static Factory Method

Méthode statique copyOf() ou  
from(). Plus de flexibilité et  
naming explicite.

## Sérialisation

Copie profonde complète  
d'objets complexes. Coûteux  
en performance.

## Bibliothèques Dédiées

ModelMapper, MapStruct,  
Dozer automatisent copie avec  
mappings configurables.



# Immutabilité : La Meilleure Alternative

Solution la plus élégante : rendre objets **immutables**. Objet ne peut être modifié après création, élimine besoin de clonage.

## Principes Immutabilité

- Classe final
- Champs final et private
- Aucun setter
- Initialisation via constructeur
- Copies défensives

## Avantages

- Thread-safety automatique
- Clés HashMap sûres
- Code simplifié
- Moins de bugs
- État partagé sans risque

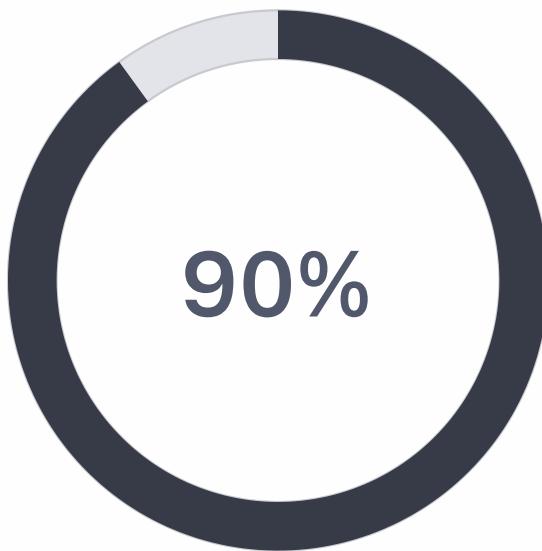
Pattern Builder excellent pour objets immutables complexes. Lombok génère avec `@Builder`.

# Records Java : Solution Moderne

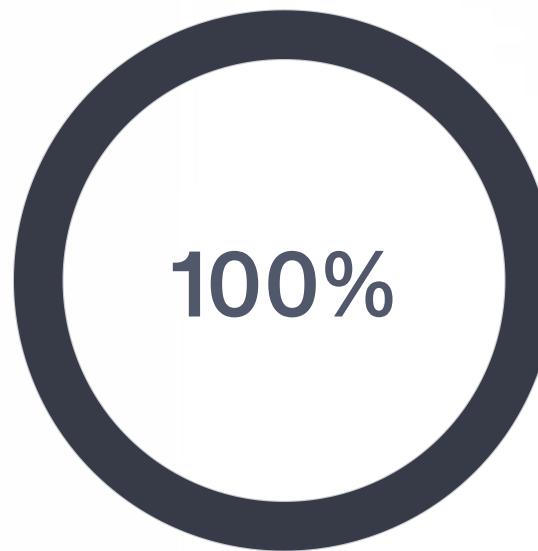
```
// Record immutable avec génération automatique  
public record Person(  
    String name,  
    int age,  
    String email  
) {}
```

// Équivalent à ~50 lignes de code avec :

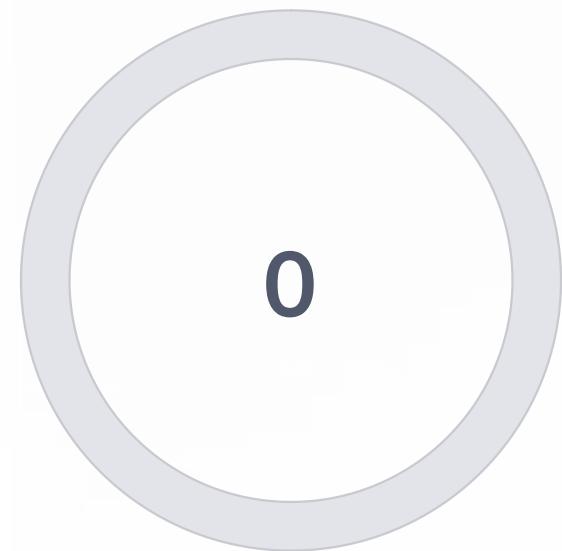
```
// - Constructeur  
// - Getters (name(), age(), email())  
// - equals() et hashCode() corrects  
// - toString() informatif  
// - Immutabilité garantie
```



Réduction du boilerplate code



Immutabilité garantie



Risque d'erreur implémentation

Records Java (Java 14+) : classes spéciales pour porteurs de données immutables. Génération automatique optimisée.

# Comparaison : Class vs Record

## Classe Traditionnelle

- Flexible, mutable par défaut
- Écriture manuelle equals(), hashCode(), toString()
- Risque erreurs et incohérences
- Bon pour comportement complexe et état mutable

## Record Java

- Syntaxe ultra-concise
- Immutable par défaut
- Génération automatique toutes méthodes
- Parfait pour données pures, DTOs, clés collections



# Génération Automatique : IDEs et Lombok



## IntelliJ IDEA

Générateurs automatiques equals(),  
hashCode(), toString()

## Eclipse

Code correct et cohérent, maintenu  
automatiquement

## Lombok

Annotation processing, génération  
bytecode compilation

```
// Lombok : une annotation élimine le boilerplate
@Data
public class Person {
    private String name;
    private int age;
    private String email;
}
// Génère automatiquement : getters, setters,
// equals(), hashCode(), toString(), constructeur
```

@Value crée classe immutable similaire aux records. Avec Java 14+, records natifs souvent meilleure option.

# Testing des Méthodes d'Object



## Tests pour equals()

- Réflexivité : `x.equals(x) == true`
- Symétrie : `x.equals(y) == y.equals(x)`
- Transitivité : chaîne d'égalité
- Cohérence : appels répétés identiques
- Comparaison null : `x.equals(null) == false`



## Tests pour hashCode()

- Objets égaux ont même hash code
- Cohérence : appels répétés identiques
- Distribution acceptable
- Test avec collections (`HashSet.contains()`)



## Tests pour toString()

- Contient champs principaux
- Ne lance pas d'exception
- Format cohérent et lisible
- Gère valeurs null gracieusement

**EqualsVerifier** automatise complètement tests d>equals() et hashCode() avec une seule ligne de code.

# Performance et Optimisations



100%



31%



5%

## Complexité HashMap

Avec bon hashCode(), opérations O(1) grâce à distribution uniforme

## Facteur Premier

Nombre 31 offre excellent compromis distribution/optimisation compilateur

## Impact Lazy Init

Cacher hashCode pour immutables améliore performances 2-5% dans collections intensives

Pour objets critiques en performance, **profilez** votre application. Mauvais hashCode() peut causer ralentissements majeurs en production.

# Checklist : Implémentation Correcte

1

## Redéfinir equals() ET hashCode() ensemble

Ne jamais redéfinir l'un sans l'autre, vous violeriez le contrat fondamental

2

## Utiliser Objects.equals() et Objects.hash()

Méthodes utilitaires gèrent null et simplifient code

3

## Inclure mêmes champs dans equals() et hashCode()

Tout champ utilisé dans equals() doit être dans hashCode(), et vice-versa

4

## Tester avec EqualsVerifier

Test unitaire pour valider tous les contrats automatiquement

5

## Préférer immutabilité ou records

Objets immutables éliminent nombreux problèmes et sont thread-safe

6

## Éviter clone(), utiliser constructeurs de copie

Mécanisme clone() fragile et complexe, préférer alternatives