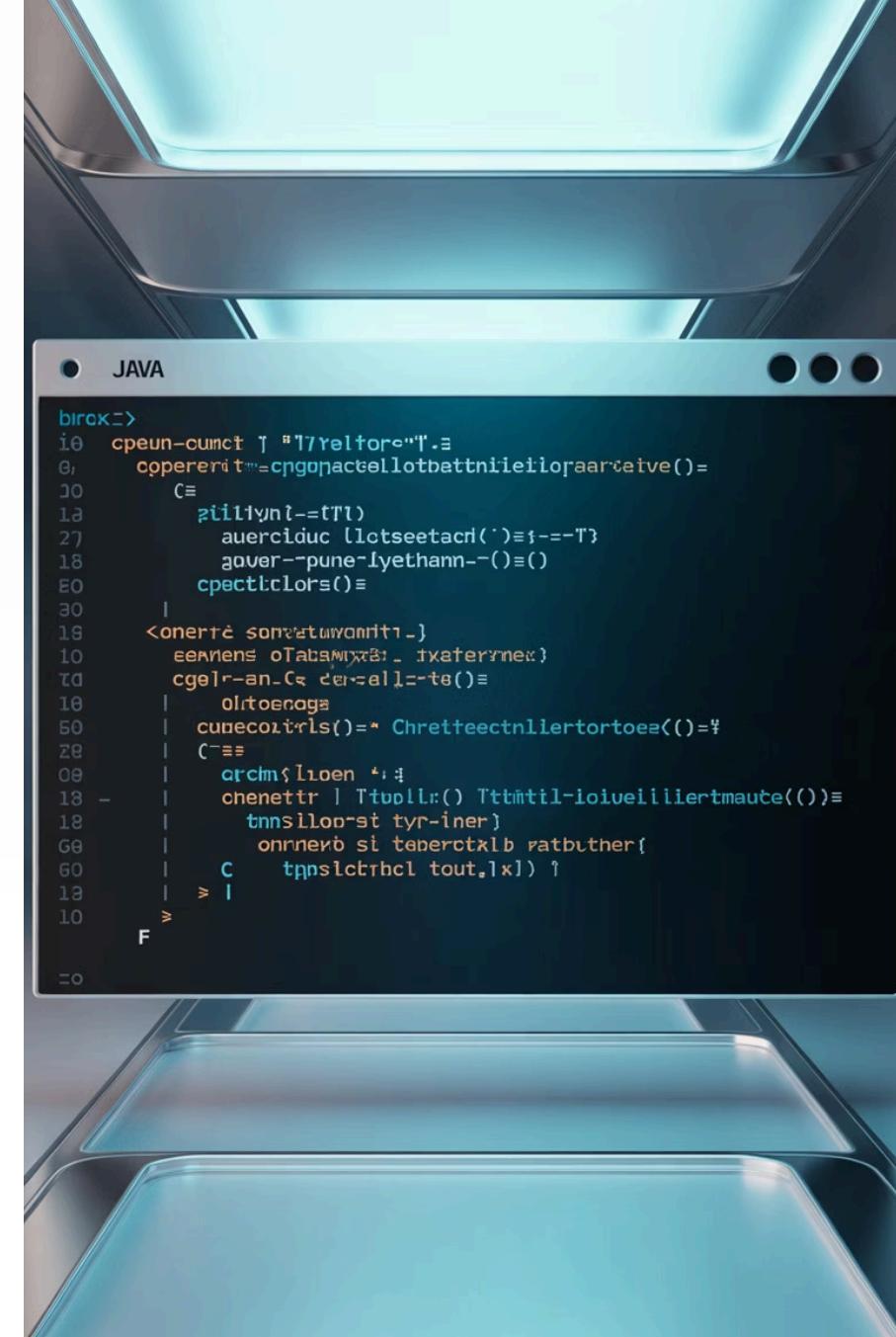


# La Généricité en Java



A close-up photograph of a computer monitor displaying a Java code editor window. The window has a dark theme with light-colored text. The code is heavily obfuscated, consisting of random characters and symbols. The title bar of the window says "JAVA". The code is as follows:

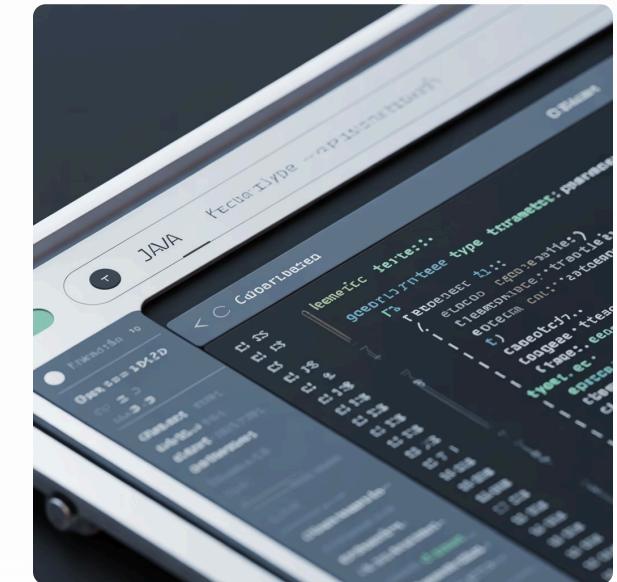
```
birck=>
10  cpeun-cumct | "l7relfore"|-.
0,   copererit=""cpogpactellotbatnlieilopaarceve()=-
20   C≡
12   ptillynl-=T1)
27   auerciduc (lctseetach(')=;--T}
18   gover--pune-lyethann--()=()
E0   cpectclors()=-
30   |
18   <onerté somewtuwanriti_-
10   eemniens oTabamixx8: _ ixaterymek)
7D   cgelr-an-Cs der-sell=-ts()=-
10   oltoenaga
50   cuuecoitrls()=" Chretteectnlertortoee()=%
C-≡≡
28   |
09   arcim\$ lloen *:#;
18   chenettir | Ttuboll:() Tttmtil-loiveillertmaute()=-
18   tnnslloo-st tyr-iner)
G8   onnerò si teberctklb ratbuther{
60   C   tpslctrhcl tout,lx) |
12   > |
10   F
=0
```

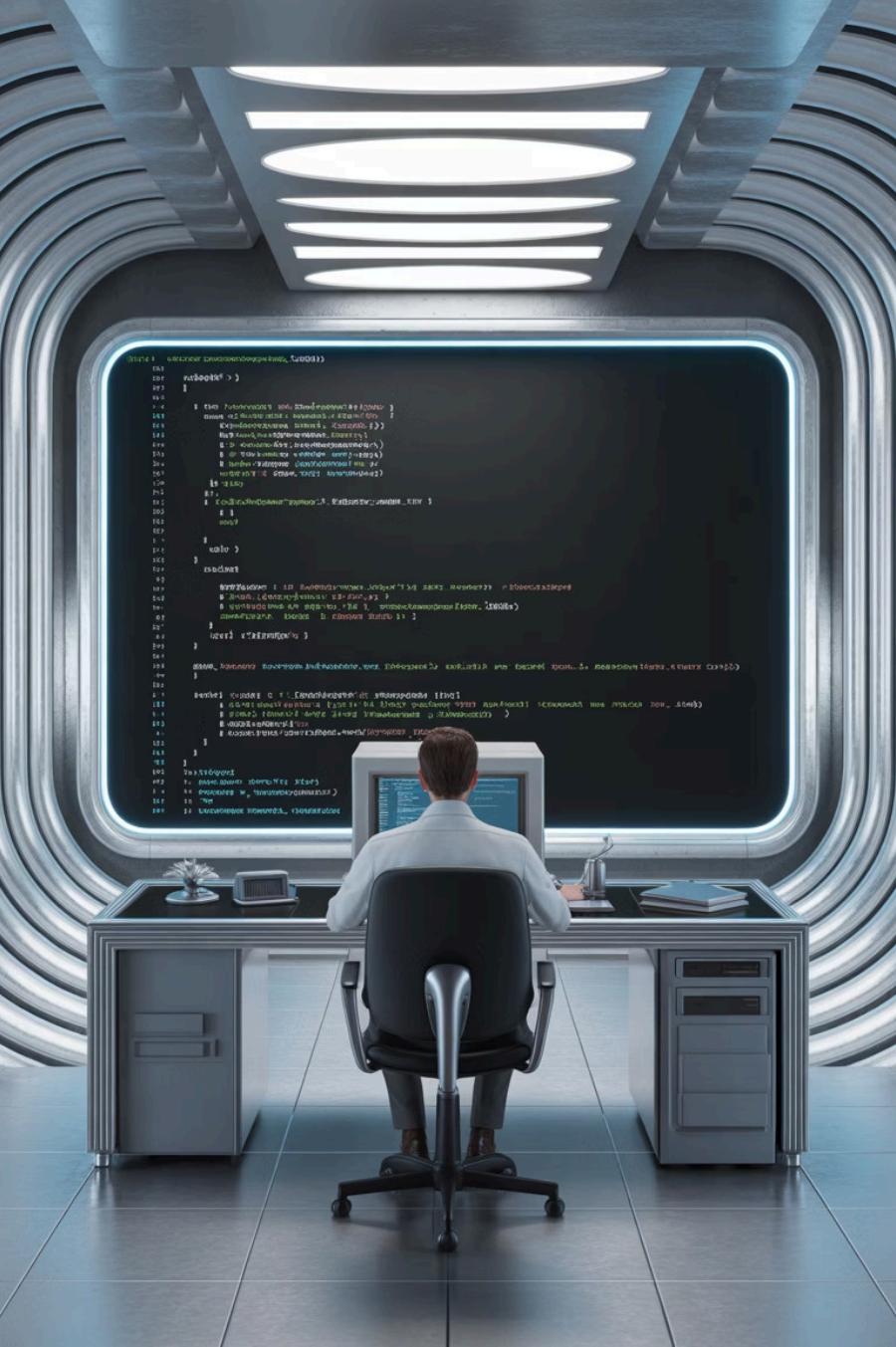


# Qu'est-ce que la Généricité ?

Mécanisme puissant introduit en Java 5 permettant de créer des classes, interfaces et méthodes avec des types paramétrés.

Transforme la façon d'écrire du code réutilisable et type-safe.





# L'Ère Avant la Généricité

## Absence de Type Safety

Collections acceptant n'importe quel type, erreurs d'exécution imprévisibles

## Casts Explicites

Conversions manuelles obligatoires, risques de ClassCastException

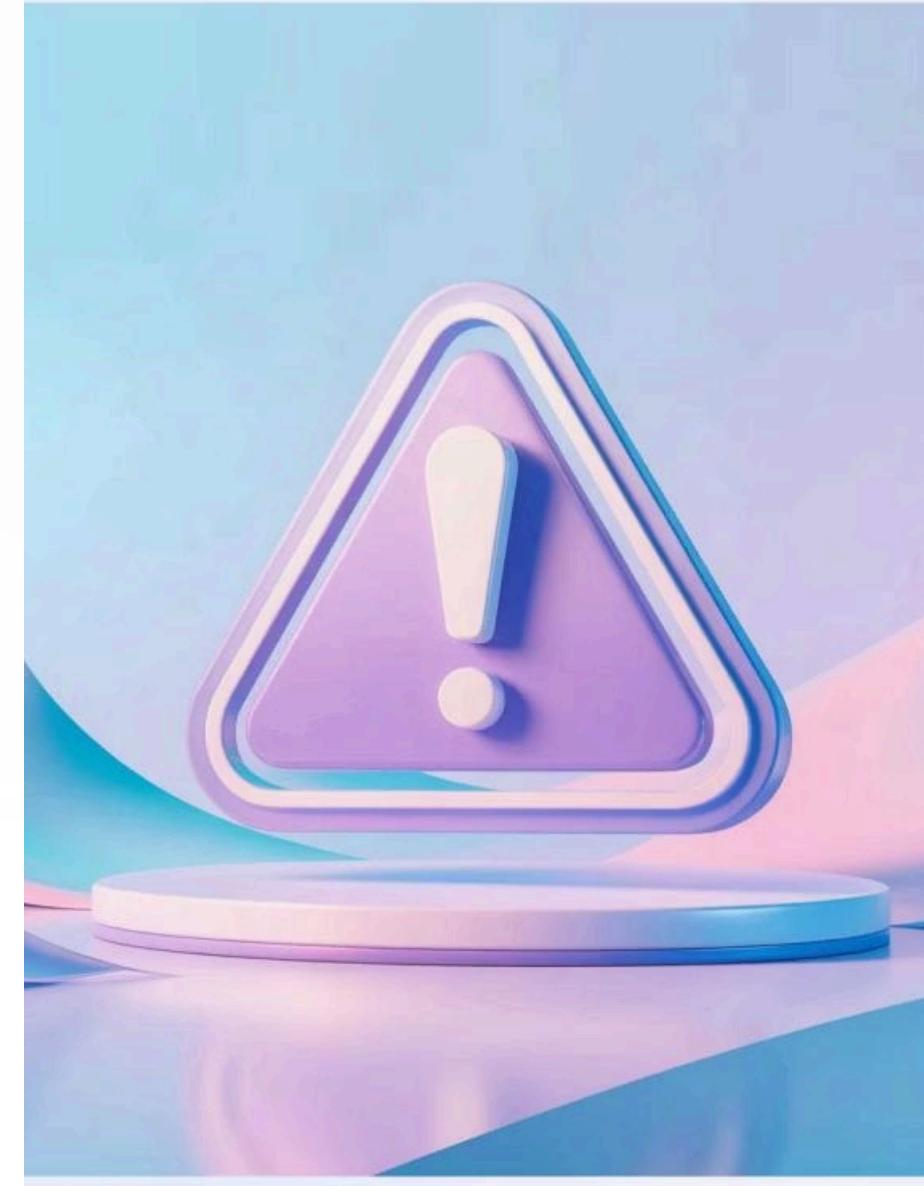
## Code Verbeux

Vérifications manuelles, complexité accrue, maintenance difficile

# Exemple : Collections Sans Généricité

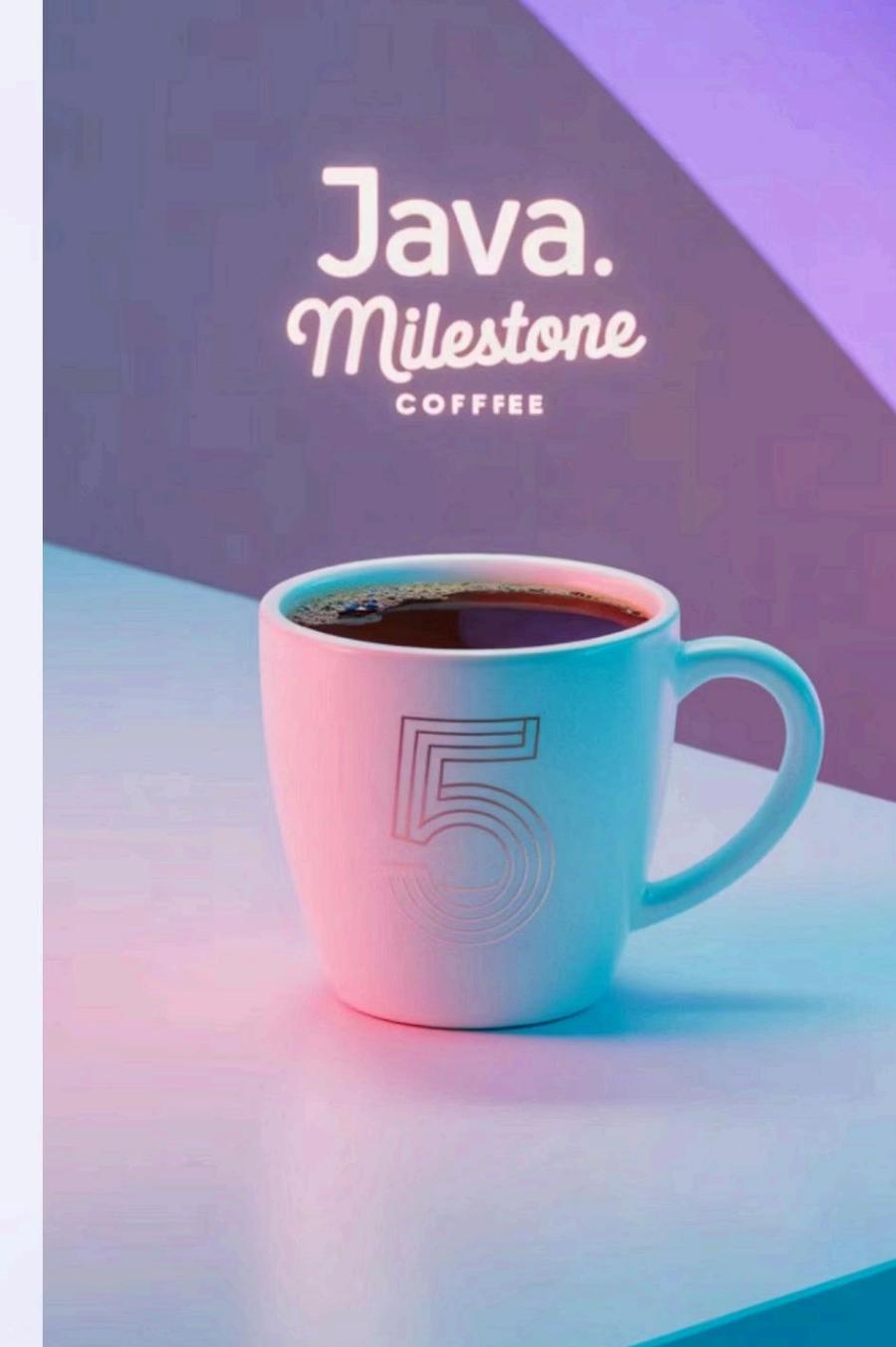
```
// Avant Java 5 - Code problématique
List list = new ArrayList();
list.add("Hello");
list.add(42); // Aucune erreur !
String str = (String) list.get(0);
String str2 = (String) list.get(1); // ClassCastException !
```

- ❑ Le véritable danger : le compilateur ne détecte pas ces incohérences. Les erreurs se manifestent uniquement à l'exécution.

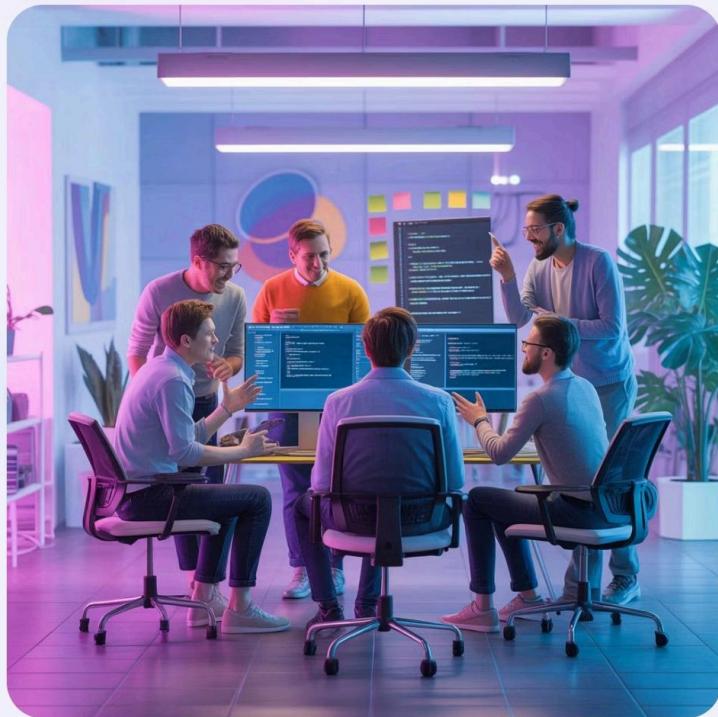


# L'Arrivée de Java 5 en 2004

- 1 2001-2002  
Début des discussions JSR 14 pour ajouter la généricité
- 2 2003  
Développement intensif de la spécification et du prototype
- 3 Sept 2004  
Lancement officiel Java 5 avec la généricité comme fonctionnalité phare
- 4 2004-2005  
Adoption progressive par la communauté et migration des bibliothèques



# Les Pionniers : JSR 14



Java Specification Request 14 : travail collaboratif d'experts dirigé par Gilad Bracha

**Innovation majeure :** concept d'effacement de type (type erasure) permettant rétrocompatibilité binaire et sécurité à la compilation

Résolution de défis complexes : compatibilité, performances, syntaxe intuitive

# La Révolution du Type Safety

```
// Avec Java 5 et généricité  
List<String> list = new  
ArrayList<>();  
list.add("Hello");  
list.add(42); // ERREUR  
COMPILATION !  
  
String str = list.get(0); // Pas de  
cast
```



## Sécurité Garantie

Erreurs détectées à la compilation



## Code Plus Propre

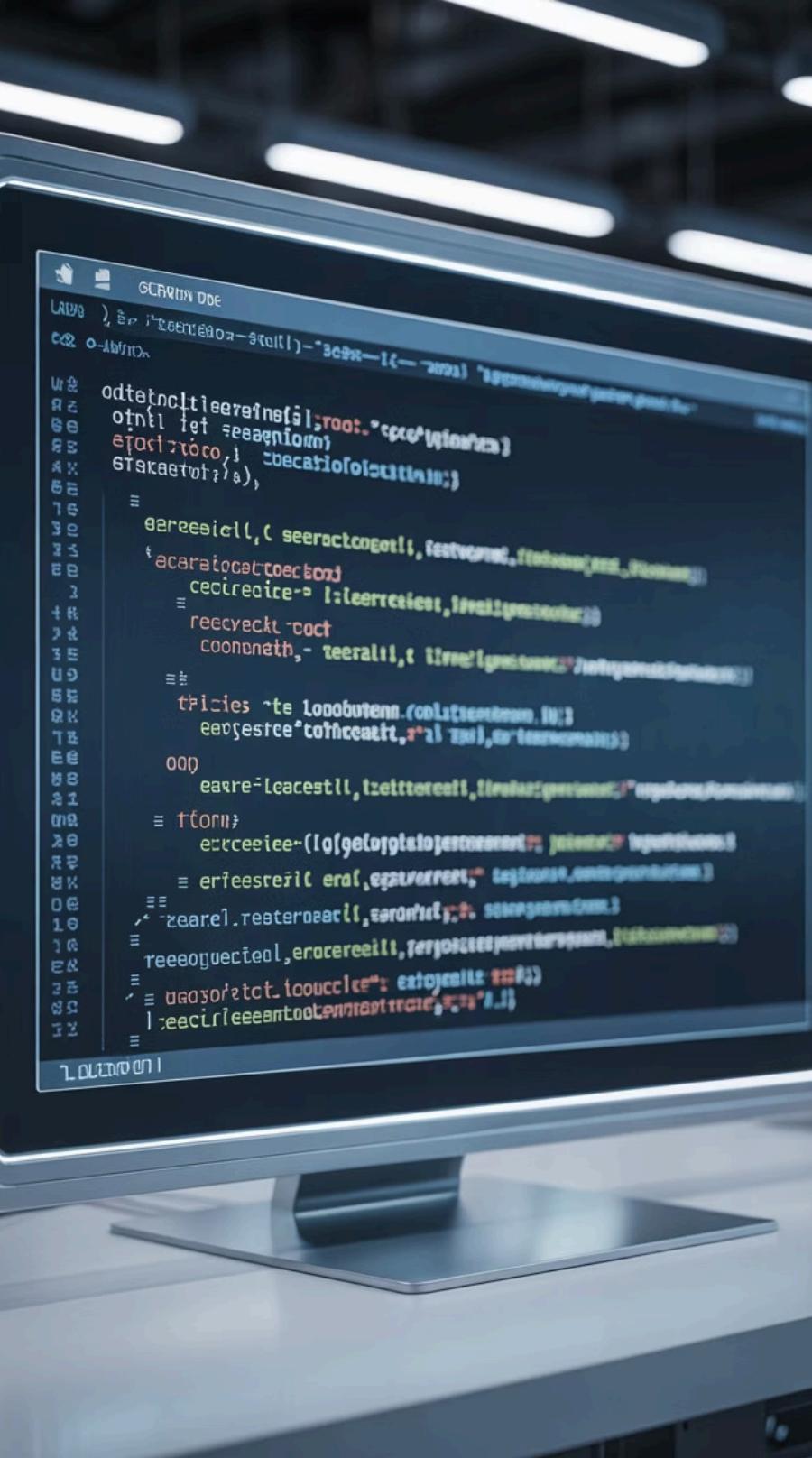
Élimination des casts explicites



## Productivité Accrue

Meilleure autocomplétion IDE





# Les Avantages Fondamentaux

01

---

## Sécurité des Types

Vérification à la compilation, élimination des erreurs de cast runtime

02

---

## Élimination des Casts

Code concis et lisible sans conversions redondantes

03

---

## Réutilisabilité Maximale

Une implémentation pour tous les types, moins de duplication

04

---

## Documentation Implicite

Paramètres de type auto-documentent le code

# Sécurité des Types : Un Pilier Essentiel



## Détection Précoce

Identification des incompatibilités avant l'exécution

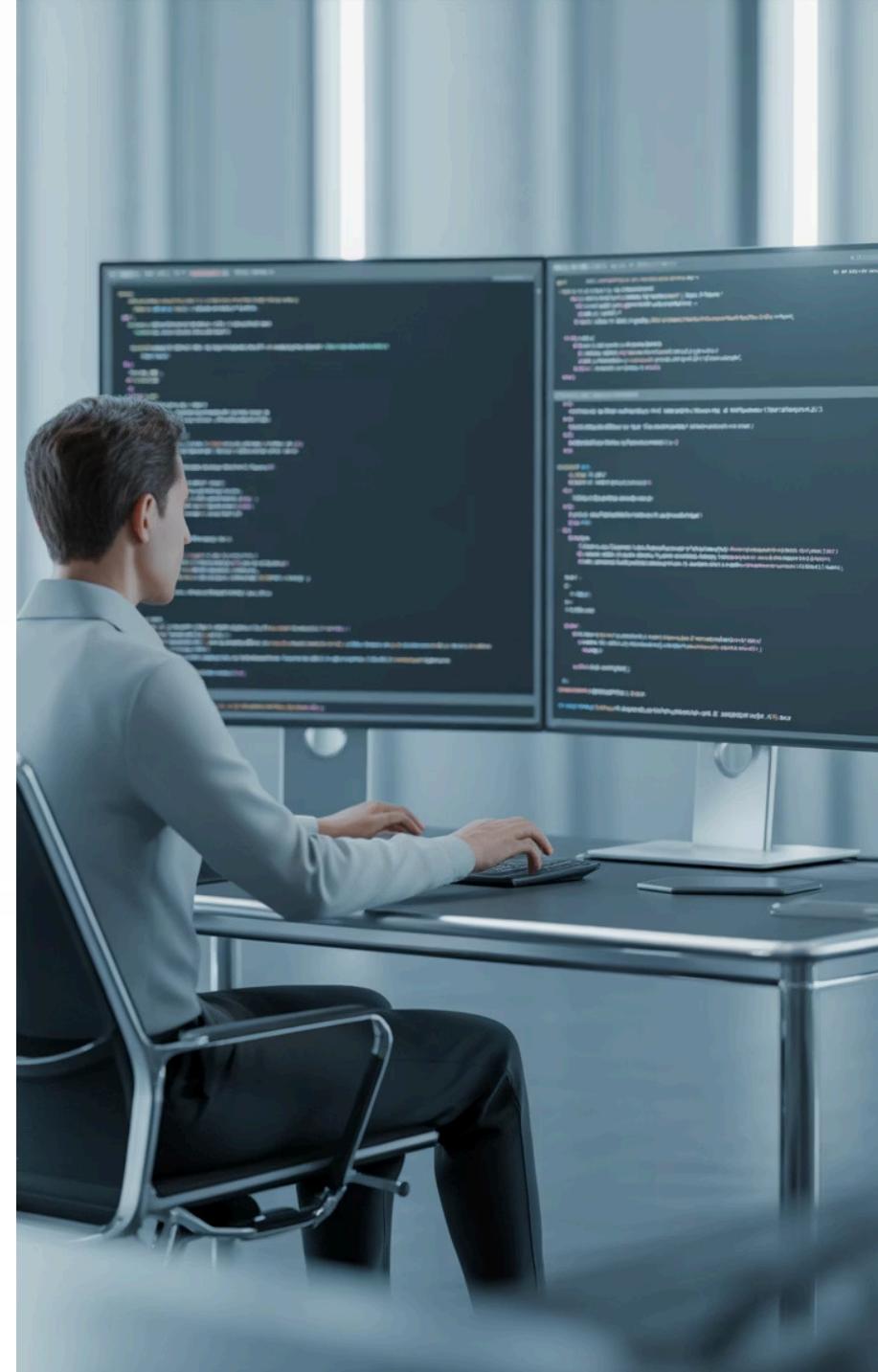
Réduction drastique du temps de débogage



## Confiance et Prévisibilité

Développement plus rapide avec assurance des types corrects

Amélioration de la collaboration en équipe



# Réutilisabilité du Code



## Composants Universels

Structures fonctionnant avec n'importe quel type

## Moins de Duplication

Une implémentation remplace des dizaines de versions

## Bibliothèques Puissantes

Frameworks génériques réutilisables dans tous les projets

La réutilisabilité transforme l'architecture logicielle, encourageant des composants abstraits et flexibles qui s'adaptent aux besoins futurs.



# Lisibilité et Maintenabilité

## Code Clair et Expressif

```
Map<String, Customer> customers;  
List<Order> pendingOrders;  
Set<Product> inventory;
```

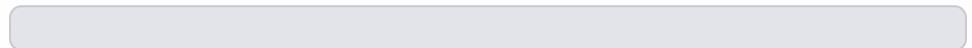
Les types génériques servent de documentation vivante,  
toujours à jour

## Code Ancien et Ambigu

```
Map customers; // Contient quoi ?  
List orders; // Quel type ?  
Set inventory; // Vraiment ?
```

Sans généricité, consultation de documentation nécessaire  
pour comprendre

# Performance et Optimisations



## Surcoût à l'Exécution

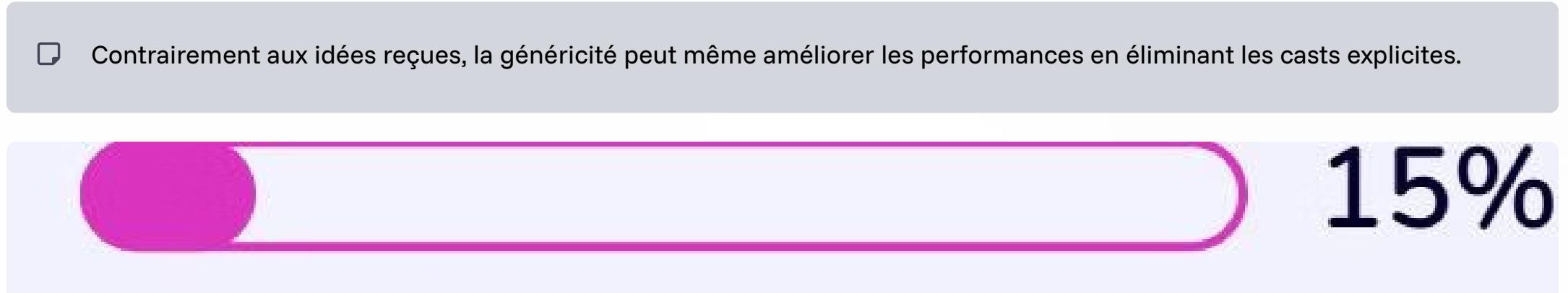
Grâce à l'effacement de type, aucun coût de performance ajouté



## Optimisation JIT

Compilateur optimise le code générique aussi efficacement que le non-générique

- Contrairement aux idées reçues, la généricité peut même améliorer les performances en éliminant les casts explicites.



# Les Limites de la Généricité

# Contraintes Principales

## Effacement de Type

Informations de type supprimées à l'exécution, empêchant certaines opérations

## Types Primitifs Non Supportés

Impossible d'utiliser int, double directement, nécessite des wrappers

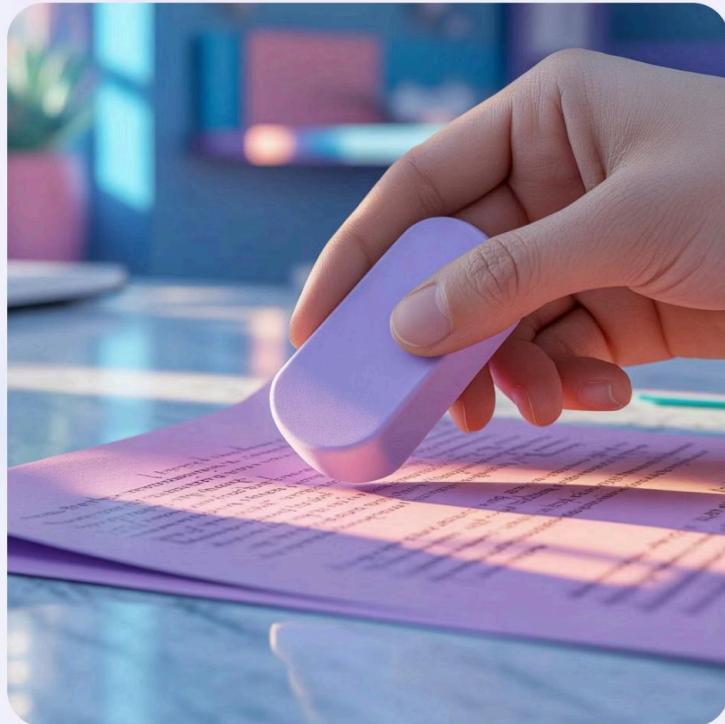
## Tableaux Génériques Limités

Création de tableaux de types génériques impossible

## Complexité Syntaxique

Wildcards et bounds peuvent rendre le code difficile à lire

# L'Effacement de Type : Compromis Technique



Technique utilisée pour implémenter la généricité tout en maintenant la compatibilité binaire avec code pré-Java 5

**Concrètement :** `List<String>` et `List<Integer>` deviennent simplement `List` dans le bytecode

Compromis permettant adoption en douceur mais imposant des limitations

# Conséquences de l'Effacement

```
// Impossible de déterminer le type à l'exécution  
List<String> strings = new ArrayList<>();  
List<Integer> integers = new ArrayList<>();  
strings.getClass() == integers.getClass() // true
```

```
// Impossible d'instancier un type générique  
class Box<T> {  
    T item = new T(); // ERREUR  
}
```

```
// instanceof ne fonctionne pas  
if (obj instanceof List<String>) // ERREUR
```

01

## Compilation

Types génériques complets avec toutes les informations

02

## Effacement

Remplacement par raw types et insertion automatique des casts

03

## Exécution

Aucune trace des types génériques dans le bytecode final

# Types Primitifs et Autoboxing

## Problème

```
// Illégal en Java  
List<int> numbers;  
Map<double, String> coords;  
  
// Solution obligatoire  
List<Integer> numbers;  
Map<Double, String> coords;
```

Types primitifs ne sont pas des objets et ne peuvent pas être stockés dans structures génériques

Autoboxing convertit automatiquement mais avec coût en performance et mémoire



# Impact de l'Autoboxing

**5x**

## Consommation Mémoire

Un Integer occupe 5 fois plus de mémoire qu'un int primitif

**3x**

## Ralentissement Potentiel

Opérations sur millions d'éléments 2-3x plus lentes avec autoboxing

**100M**

## Seuil Critique

Au-delà de 100M opérations, impact mesurable et significatif

- ☐ Pour systèmes haute performance, considérez bibliothèques spécialisées comme Trove ou FastUtil

# Tableaux et Généricité

```
// Interdit par le compilateur  
List<String>[] arrayOfLists = new List<String>[10]; // ERREUR  
T[] array = new T[10]; // ERREUR dans classe générique
```

```
// Workarounds communs  
@SuppressWarnings("unchecked")  
List<String>[] arrayOfLists = new List[10];
```

```
// Ou mieux : utiliser une List de Lists  
List<List<String>> listOfLists = new ArrayList<>();
```

## Problème : Covariance

Tableaux Java covariants entrent en conflit avec invariance des génériques

## Solution : Collections

Privilégier collections génériques : plus de flexibilité et sécurité



# Complexité Syntaxique

```
// Exemple de complexité syntaxique
public <T extends Comparable<? super T>> void sort(List<T> list)

class EnumMap<K extends Enum<K>, V> extends AbstractMap<K, V>

<T extends Object & Comparable<? super T>> T max(Collection<? extends T> coll)
```

## Le Défi de la Lisibilité

Signatures complexes peuvent obscurcir l'intention du code

Courbe d'apprentissage abrupte pour débutants

## Trouver l'Équilibre

Parfois, interface moins générique mais plus claire est préférable

Sacrifier flexibilité théorique pour gagner en clarté

# Création et Exploitation

# Classe Générique Simple

```
// Classe générique simple et élégante
public class Box<T> {
    private T content;

    public void set(T content) {
        this.content = content;
    }

    public T get() {
        return content;
    }

    public boolean isEmpty() {
        return content == null;
    }
}

// Utilisation
Box<String> stringBox = new Box<>();
stringBox.set("Hello Generics");
String value = stringBox.get(); // Pas de cast !
```

# Paramètres de Type Multiples

```
// Classe avec deux paramètres
public class Pair<K, V> {
    private final K key;
    private final V value;

    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public K getKey() { return key; }
    public V getValue() { return value; }
}

// Utilisation pratique
Pair<String, Integer> nameAge =
    new Pair<>("Alice", 30);
```

## Type K pour Key

Représente le type de la clé

## Type V pour Value

Désigne le type de la valeur

## Type T pour Type

Paramètre générique unique



# Méthodes Génériques

```
public class Utils {  
    // Méthode générique statique  
    public static <T> T getFirst(List<T> list) {  
        if (list == null || list.isEmpty()) return null;  
        return list.get(0);  
    }  
  
    // Méthode avec bound  
    public static <T extends Comparable<T>> T max(T a, T b) {  
        return a.compareTo(b) > 0 ? a : b;  
    }  
  
    // Plusieurs paramètres de type  
    public static <K, V> Map<V, K> invert(Map<K, V> map) {  
        Map<V, K> inverted = new HashMap<>();  
        for (Map.Entry<K, V> entry : map.entrySet()) {  
            inverted.put(entry.getValue(), entry.getKey());  
        }  
        return inverted;  
    }  
}
```

# Bounded Type Parameters

01

## Upper Bound avec extends

Restreint le type à une classe spécifique ou ses sous-classes

Exemple : <T extends Number> accepte Integer, Double, etc.

02

## Multiple Bounds

Paramètre peut avoir plusieurs bounds séparés par &

Classe en premier, suivie des interfaces

03

## Utilisation dans le Corps

Appel des méthodes définies par les bounds sur instances du type paramétré

```
// Exemple complet avec bound
public class NumberBox<T extends Number> {
    private T value;

    public double doubleValue() {
        return value.doubleValue(); // Possible car Number
    }
}

// Multiple bounds
public <T extends Comparable<T> & Serializable> void process(T item)
```

# Wildcards : Le ? Mystérieux

1

**Unbounded Wildcard : ?**

Représente n'importe quel type

```
public void printList(List<?>  
list)
```

2

**Upper Bounded : ?  
extends Type**

Type ou n'importe quel sous-type

```
public void  
processNumbers(List<?  
extends Number> list)
```

3

**Lower Bounded : ? super Type**

Type ou n'importe quel super-type

```
public void addIntegers(List<? super Integer> list)
```



# PECS : Producer Extends Consumer Super

## Producer Extends

Si vous **lisez** des éléments (structure produit), utilisez ?  
extends T

```
public void processAll(List<? extends Number>
                      numbers) {
    for (Number n : numbers) {
        System.out.println(n);
    }
}
```

## Consumer Super

Si vous **écrivez** des éléments (structure consommé), utilisez ?  
super T

```
public void addNumbers(List<? super Integer> list) {
    list.add(42);
    list.add(100);
}
```

# Bonnes Pratiques Essentielles



## Nommage des Paramètres

T (Type), E (Element), K/V (Key/Value), N (Number)

Conventions courtes préférées pour réduire verbosité



## Préférer Interfaces aux Classes

Utiliser List au lieu de ArrayList dans signatures

Maximise flexibilité et facilite tests



## Diamond Operator

Utiliser <> pour inférence de type (Java 7+)

Élimine redondance et rend code plus concis



## Éviter Raw Types

Toujours spécifier paramètres de type

Raw types suppriment sécurité de type

# Collections Java et Généricité



## List<E>

Collections ordonnées permettant doublons

ArrayList, LinkedList, Vector



## Set<E>

Collections sans doublons

HashSet, TreeSet, LinkedHashSet



## Map<K,V>

Structures associatives clé-valeur

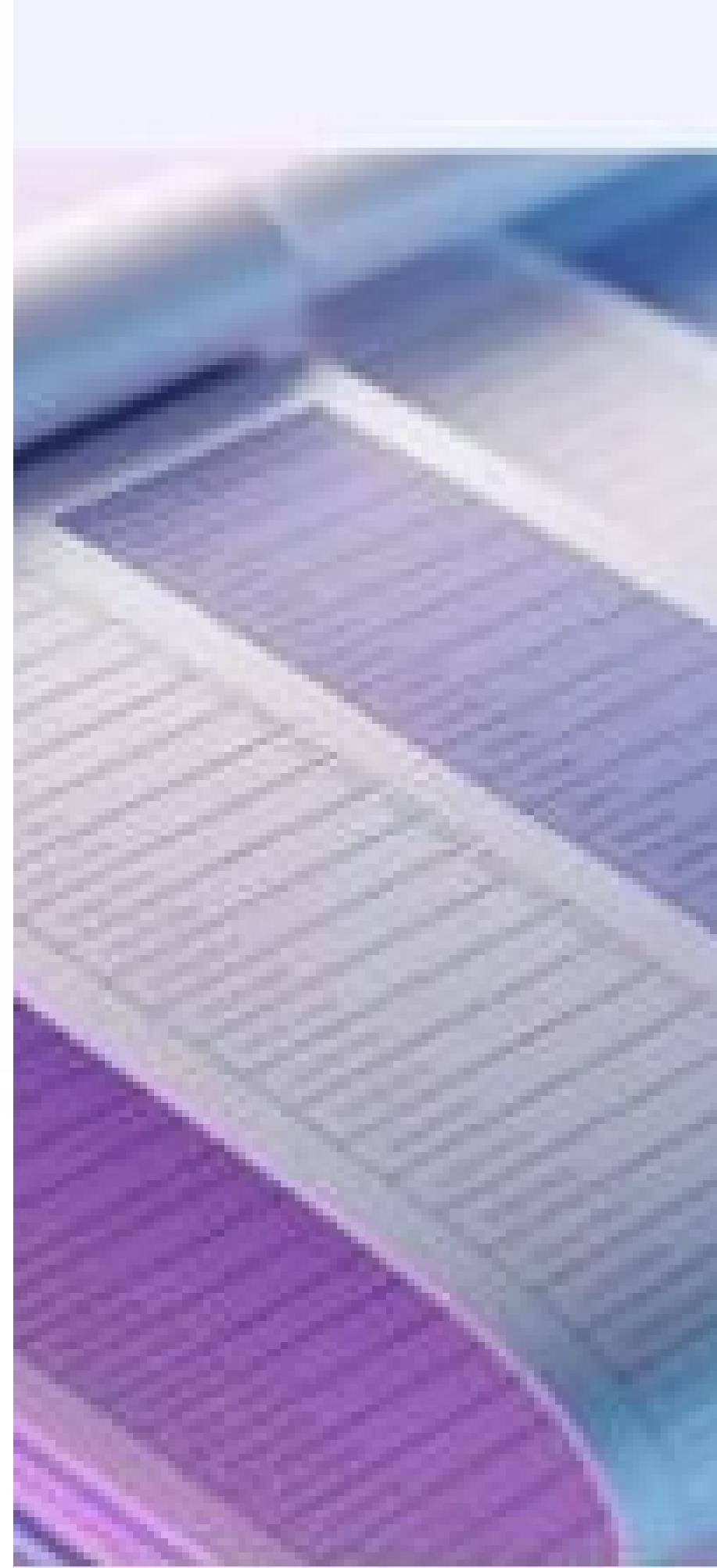
HashMap, TreeMap, LinkedHashMap



## Queue<E>

Collections avec ordre de traitement

PriorityQueue, LinkedList



# Conclusion : Maîtriser la Généricité

La généricité est bien plus qu'une fonctionnalité - c'est un paradigme qui transforme la qualité et la maintenabilité de votre code

## Outil, Pas une Fin

Utiliser judicieusement en gardant clarté et maintenabilité comme priorités

## Pratique Continue

Développer l'intuition pour savoir quand appliquer et comment concevoir APIs élégantes

## Apprentissage Continu

Explorer bibliothèques natives, étudier code source, expérimenter dans chaque projet

