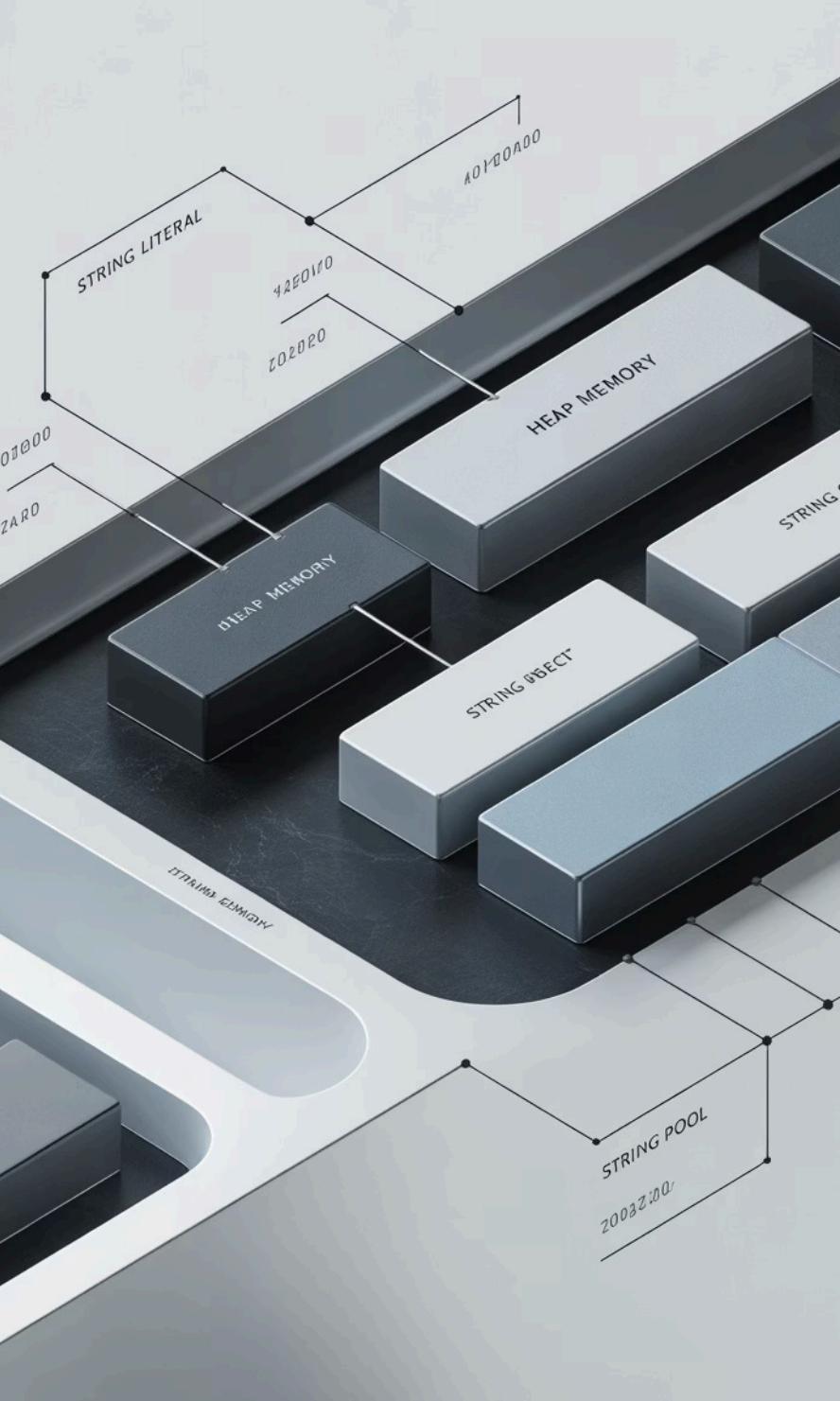


La Classe String en Java



Qu'est-ce que la classe String ?

Immutable

Une fois créée, son contenu ne change jamais

Classe complète

Hérite de Object, implémente Serializable, Comparable, CharSequence

String Pool

Zone mémoire spéciale pour réutilisation efficace

Thread-safe

Sécurisée automatiquement pour usage concurrent



L'immutabilité : Concept fondamental

Avantages

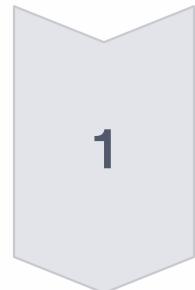
- **Thread-safety** : accès concurrent sans risque
- **Sécurité** : clés HashMap sans modification
- **Optimisation** : réutilisation via String Pool

Coût

- Chaque modification crée un nouvel objet
- Surcharge mémoire potentielle
- Impact performance si manipulations intensives

L'immutabilité garantit la sécurité et la simplicité au prix de la création fréquente d'objets

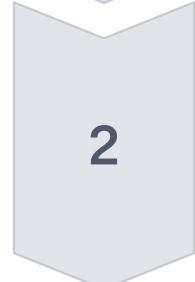
Le String Pool : Optimisation mémoire



1 Littéral

```
String s1 = "Hello";
```

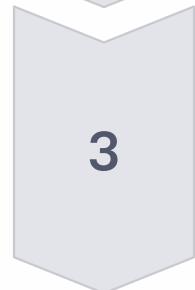
Réutilise l'objet du pool



2 new String()

```
String s3 = new String("Hello");
```

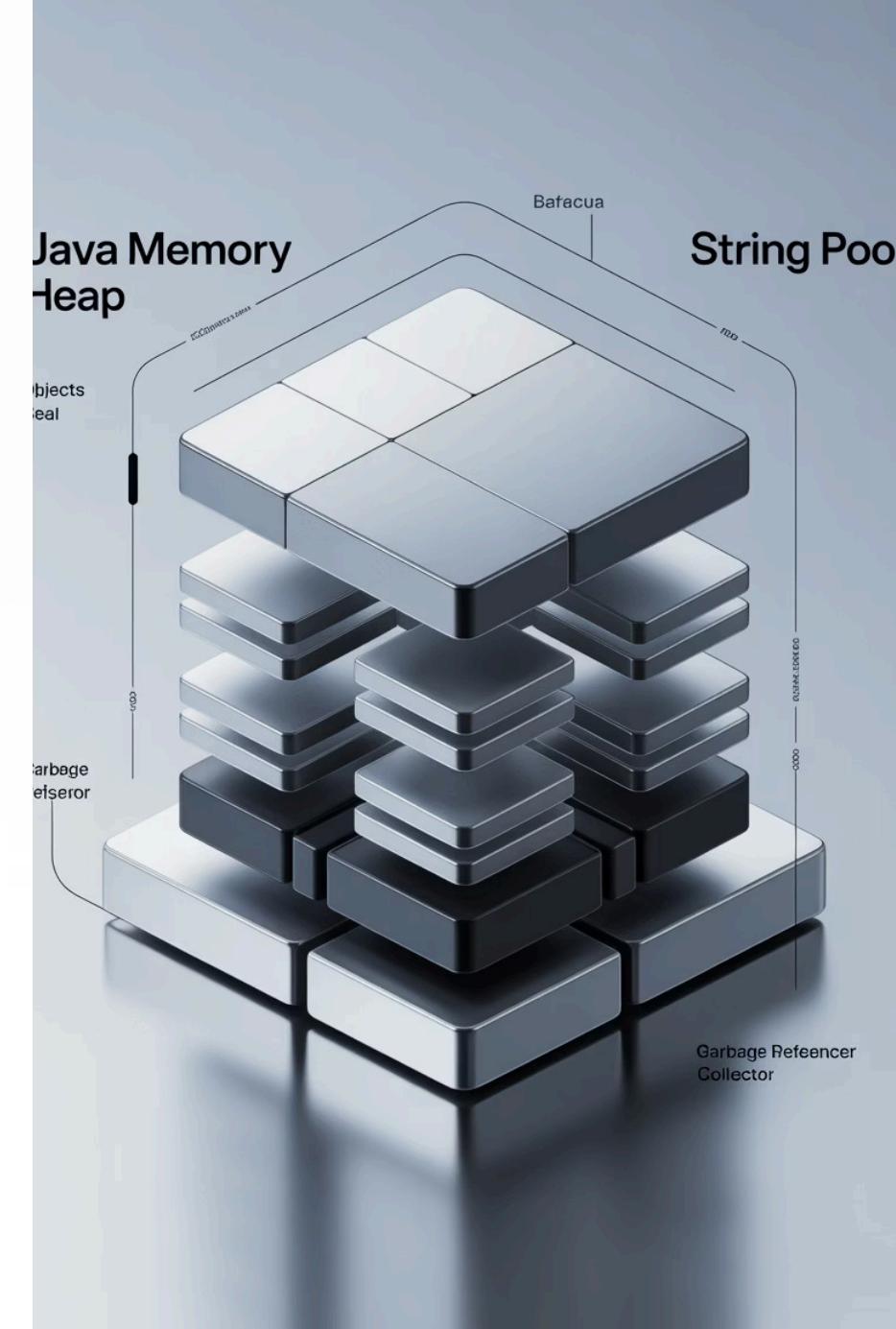
Crée nouvel objet en heap

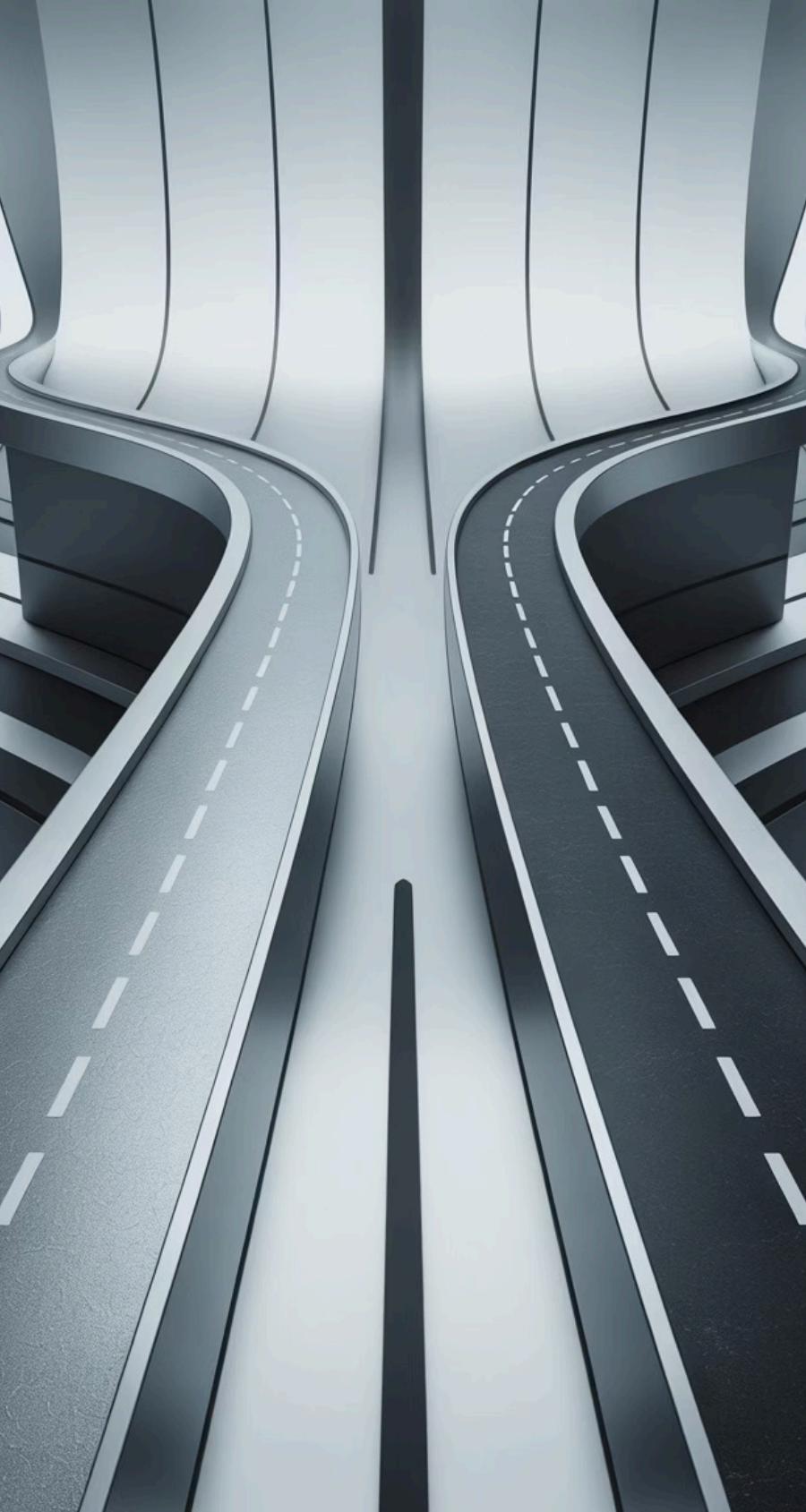


3 intern()

```
String s4 = s3.intern();
```

Force ajout/récupération du pool





`==` vs `equals()`

Opérateur `==`

Compare les **références mémoire**, pas le contenu

```
String s1 = "Java";
String s3 = new String("Java");
s1 == s3 // false
```

Antipattern : Utiliser pour comparer des String de sources différentes

Méthode `equals()`

Compare le **contenu** caractère par caractère

```
s1.equals(s3) // true
"Java".equals(s1) // Pattern sûr
```

: Toujours utiliser `equals()`, littéral en premier

Patterns de création : Bonnes pratiques

01

Privilégier les littéraux

String message = "Bonjour";

Optimise et utilise le String Pool

02

Éviter new String()

Contourne le pool, crée objets inutiles

03

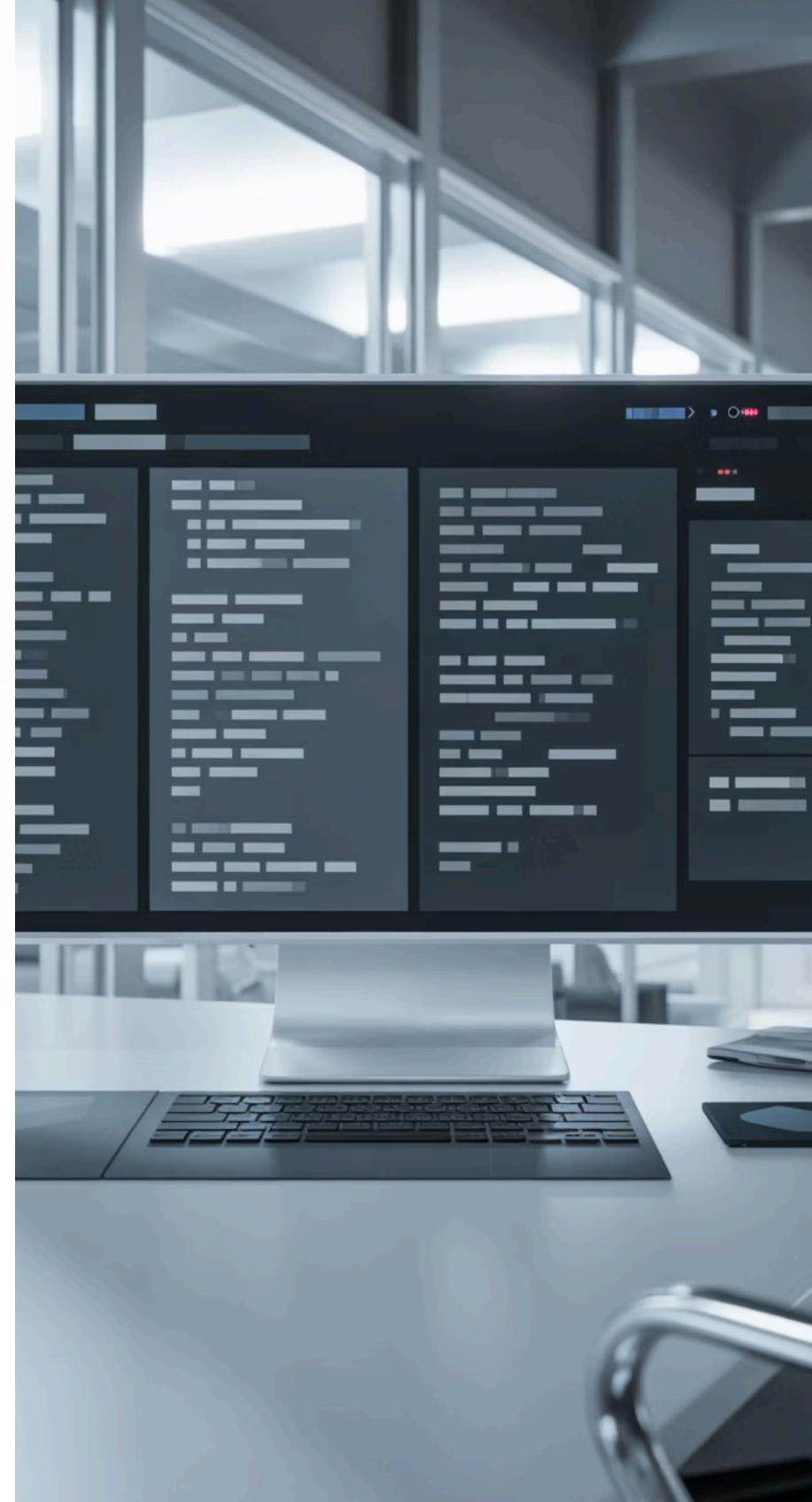
Utiliser String.valueOf()

Gère correctement les valeurs null

04

Constantes en UPPER_CASE

public static final String DEFAULT_ENCODING = "UTF-8";



Antipattern : Concaténation en boucle

✗ ANTIPATTERN

```
String result = "";  
for (int i = 0; i < 1000; i++) {  
    result += "élément " + i;  
}  
// Crée 1000+ objets  
temporaires
```

Complexité $O(n^2)$, surcharge mémoire considérable

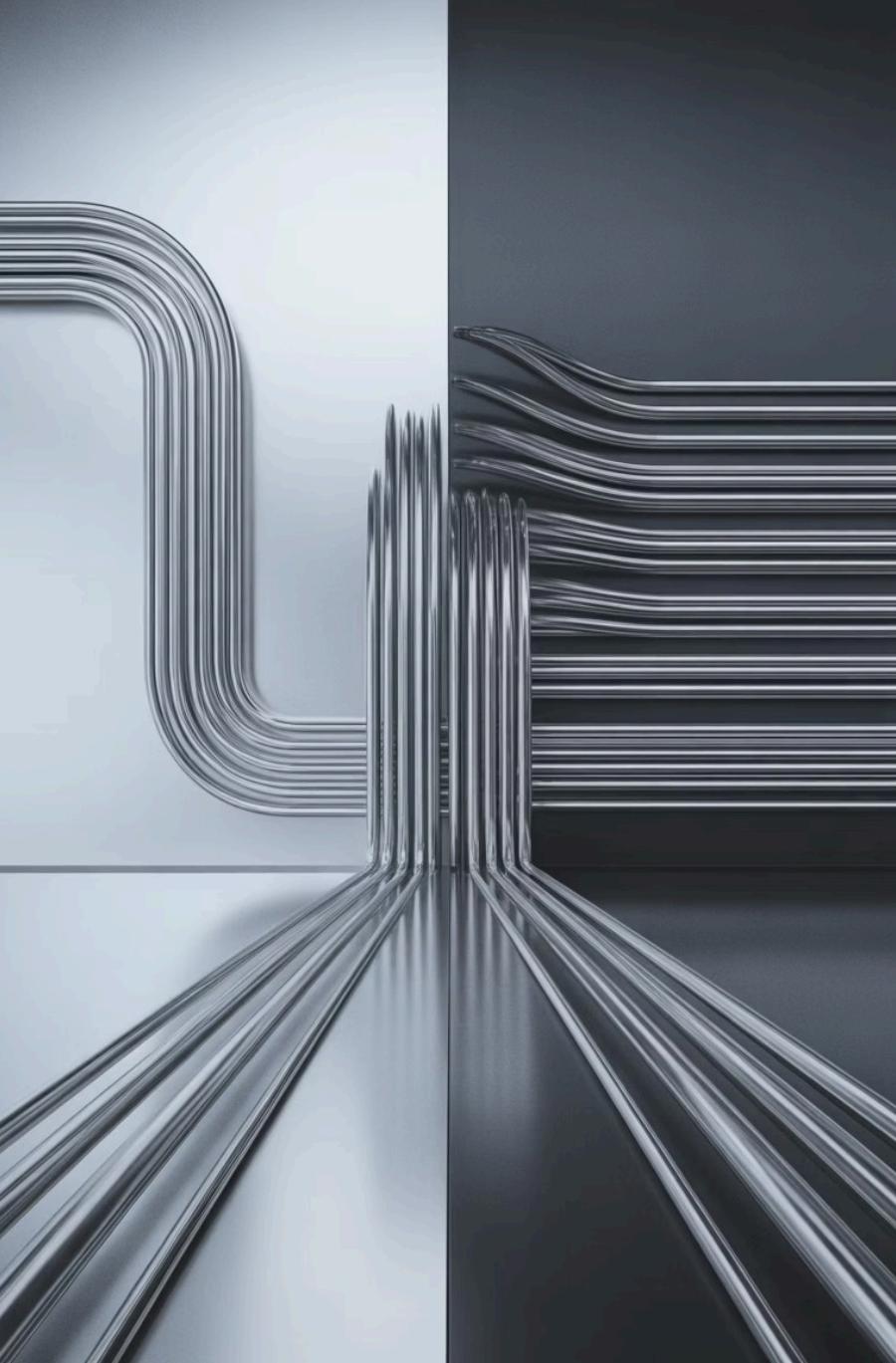
✓ PATTERN

```
StringBuilder builder =  
    new StringBuilder();  
for (int i = 0; i < 1000; i++) {  
    builder.append("élément  
").append(i);  
}  
String result =  
builder.toString();
```

Un seul objet final, 100-1000x plus rapide

- ❑ **Règle d'or :** Dès que vous concaténez dans une boucle, utilisez `StringBuilder`





StringBuilder vs StringBuffer

StringBuilder

- **Non synchronisé** - Plus rapide
- **Thread-unsafe** - Contexte monothread
- **Java 5+** - Alternative moderne

Recommandé dans 95% des cas

StringBuffer

- **Synchronisé** - Méthodes protégées
- **Thread-safe** - Usage concurrent
- **Java 1.0** - Historique

Uniquement si multi-threading requis

Méthodes essentielles : Recherche

indexOf() / lastIndexOf()

```
text.indexOf("a"); // 1  
text.lastIndexOf("a"); // 9  
text.indexOf("a", 5); // 7
```

Retourne -1 si non trouvé

substring()

```
text.substring(0, 5); //  
"Hello"  
text.substring(6); // "World"
```

Index de fin exclusif

contains() / startsWith()

```
text.contains("velop"); // true  
text.startsWith("dev"); // true  
text.endsWith("ment"); // true
```

Tests sensibles à la casse



Méthodes de transformation



Casse

```
text.toLowerCase(); // "java"  
text.toUpperCase(); // "JAVA"
```

Attention aux locales pour certains caractères



Découpage

```
String[] parts = csv.split(",");  
// ["a", "b", "c"]
```

Divise selon pattern regex



Remplacement

```
text.replace("World", "Java");  
text.replaceAll("\\s+", "_");
```

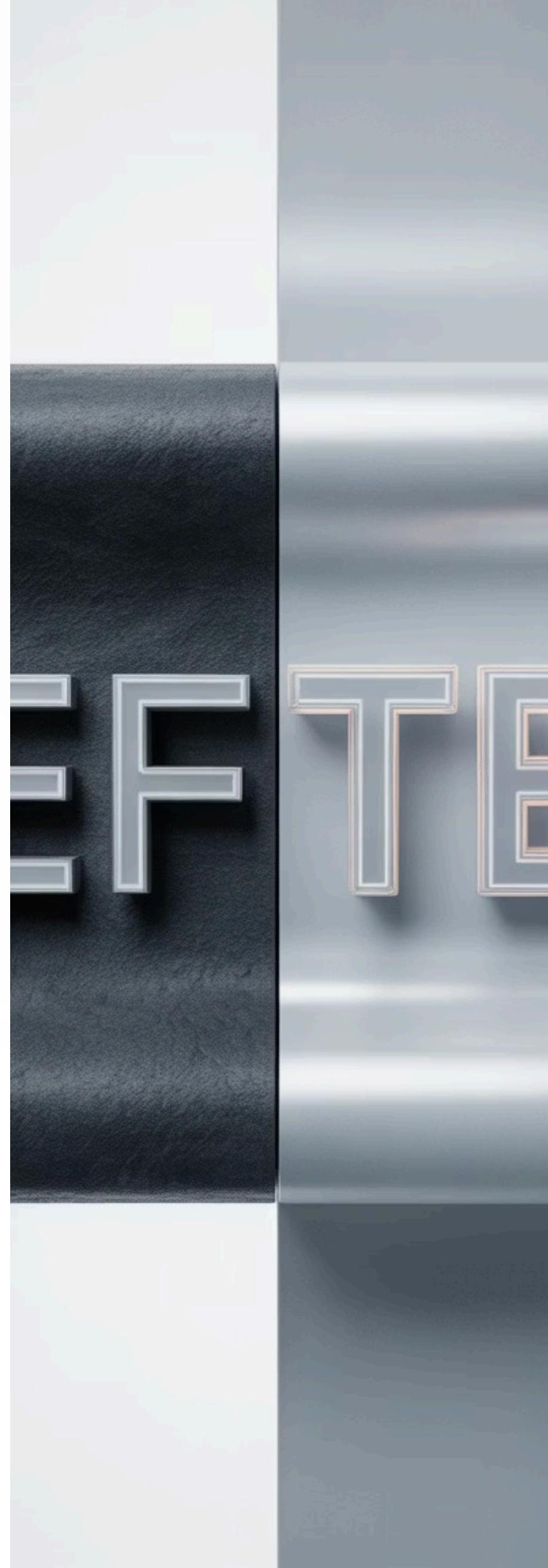
replace() littéral, replaceAll() regex



Nettoyage

```
text.trim(); // "Java"  
text.strip(); // Java 11+
```

strip() gère mieux Unicode



Pattern : Validation efficace



Vérifier null en premier

Éviter NullPointerException



Tester isEmpty() ou isBlank()

isEmpty() vérifie longueur zéro, isBlank() (Java 11+) vérifie aussi espaces



Normaliser les données

Appliquer trim(), toLowerCase() pour uniformiser



Valider le format

Utiliser regex ou méthodes spécialisées

```
public boolean isValidEmail(String email) {  
    if (email == null || email.isBlank()) return false;  
    email = email.trim().toLowerCase();  
    return email.contains "@" && email.indexOf "@" > 0;  
}
```





Expressions régulières : Puissance et précaution

Antipattern

```
for (String input : inputs) {  
    if  
(input.matches("\\d{3}")) {  
        // Recompile à chaque  
        fois  
    }  
}
```

Pattern optimisé

```
Pattern pattern =  
    Pattern.compile("\\d{3}");  
for (String input : inputs) {  
    if (pattern.matcher(input)  
        .matches()) {  
        // Réutilise le pattern  
    }  
}
```

Compilation coûteuse

Compiler une fois, réutiliser : amélioration 10-100x

Double échappement

\\d pour chiffre, \\. pour point littéral

Attention ReDoS

Valider et limiter longueur des entrées utilisateur

Format Strings et printf

1

Spécificateurs de base

%s (String), %d (entier), %f
(décimal)

```
String.format("Nom: %s,  
Age: %d", nom, age);
```

2

Options avancées

%10s (largeur), %-10s
(alignement gauche), %010d
(zéros)

```
String.format("| %-15s |  
%8.2f |", "Article", 129.50);
```

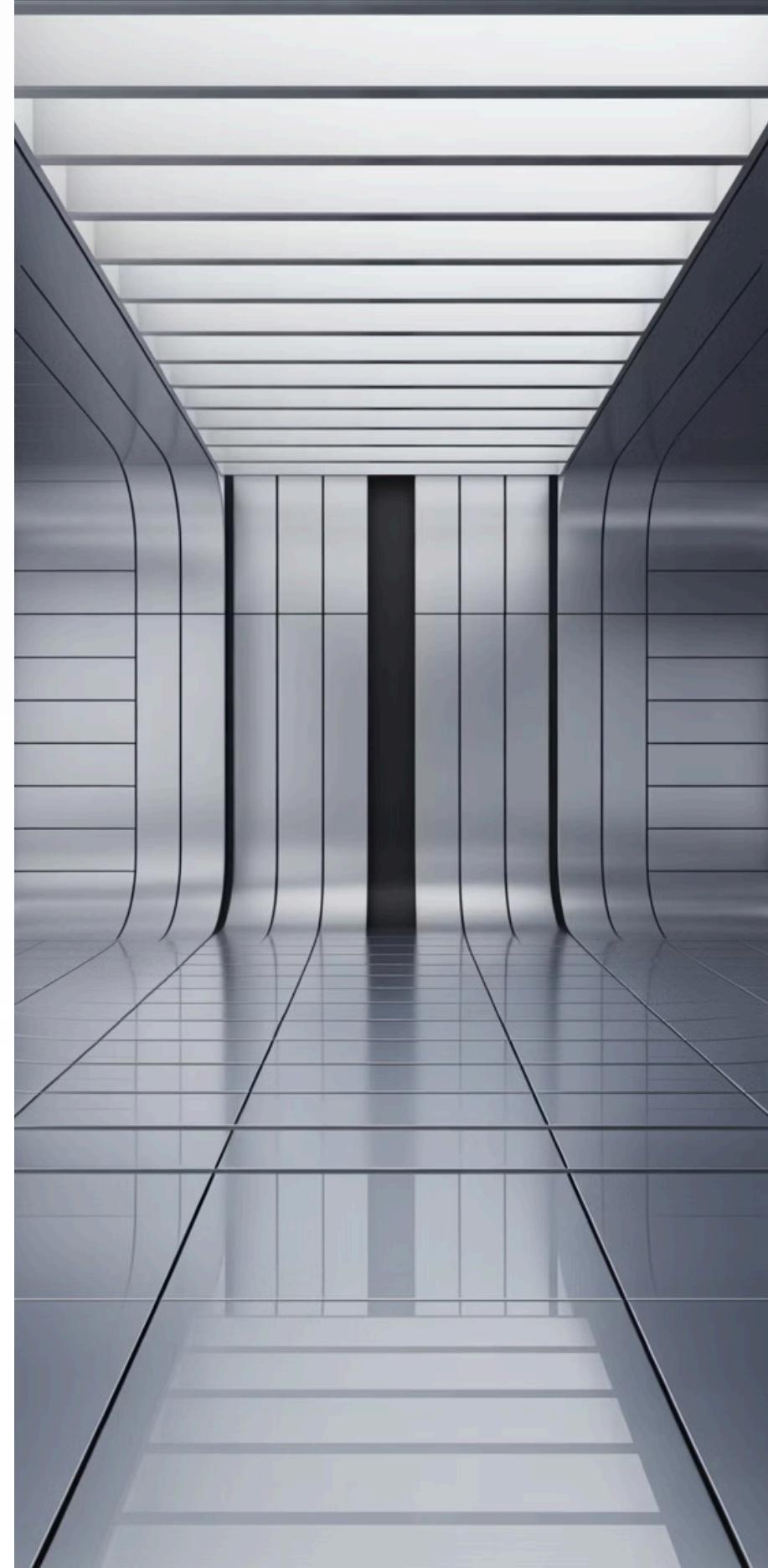
3

Indices positionnels

Réordonner les arguments

```
String.format("%2$s aime %1$s", "Java", "Alice");  
// "Alice aime Java"
```

- ❑ String.format() a un coût de performance. Pour boucles serrées,
préférez StringBuilder



Text Blocks (Java 15+)

Avant

```
String json = "{\n    \"nom\": \"Dupont\",\\n    \"age\": 30\\n}\n";
```

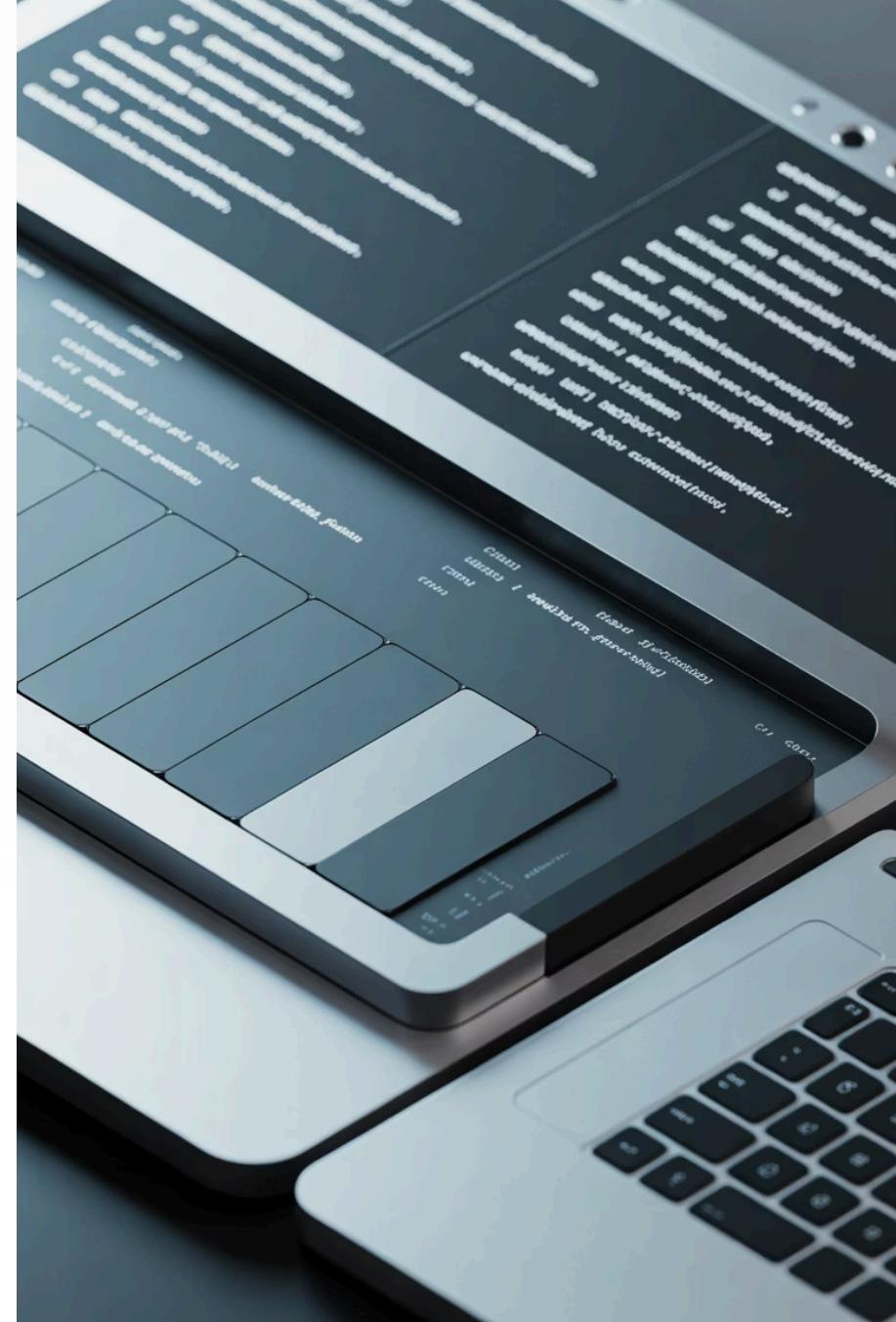
Concaténation lourde,
échappements multiples, difficile à
maintenir

Indentation commune automatiquement supprimée. Parfait pour SQL, JSON,
HTML, XML

Avec Text Blocks

```
String json = """"\n    \"nom\": \"Dupont\",\\n    \"age\": 30\n""";
```

Syntaxe naturelle, lisibilité
améliorée, pas d'échappement





Performance : Analyse et optimisation

100x

Gain StringBuilder

Dans boucles de concaténation intensives vs opérateur

+

30%

Coût String.format()

Surcoût par rapport à StringBuilder pour formats simples

64KB

Taille String Pool

Capacité par défaut (configurable avec -XX:StringTableSize)

2x

Allocation mémoire

Surcoût approximatif String vs tableau chars équivalent

Profilez votre application pour identifier les vrais goulots avant d'optimiser prématûrement

Internationalisation et Locale



Problèmes de casse

```
String turkish = "TITLE";
turkish.toLowerCase();
// "title" en turc!
turkish.toLowerCase(Local
e.ENGLISH);
// "title"
```

Toujours spécifier Locale pour opérations critiques



Comparaison culturelle

```
Collator collator =
Collator.getInstance(
Locale.FRENCH);
collator.compare("été",
"etude");
```

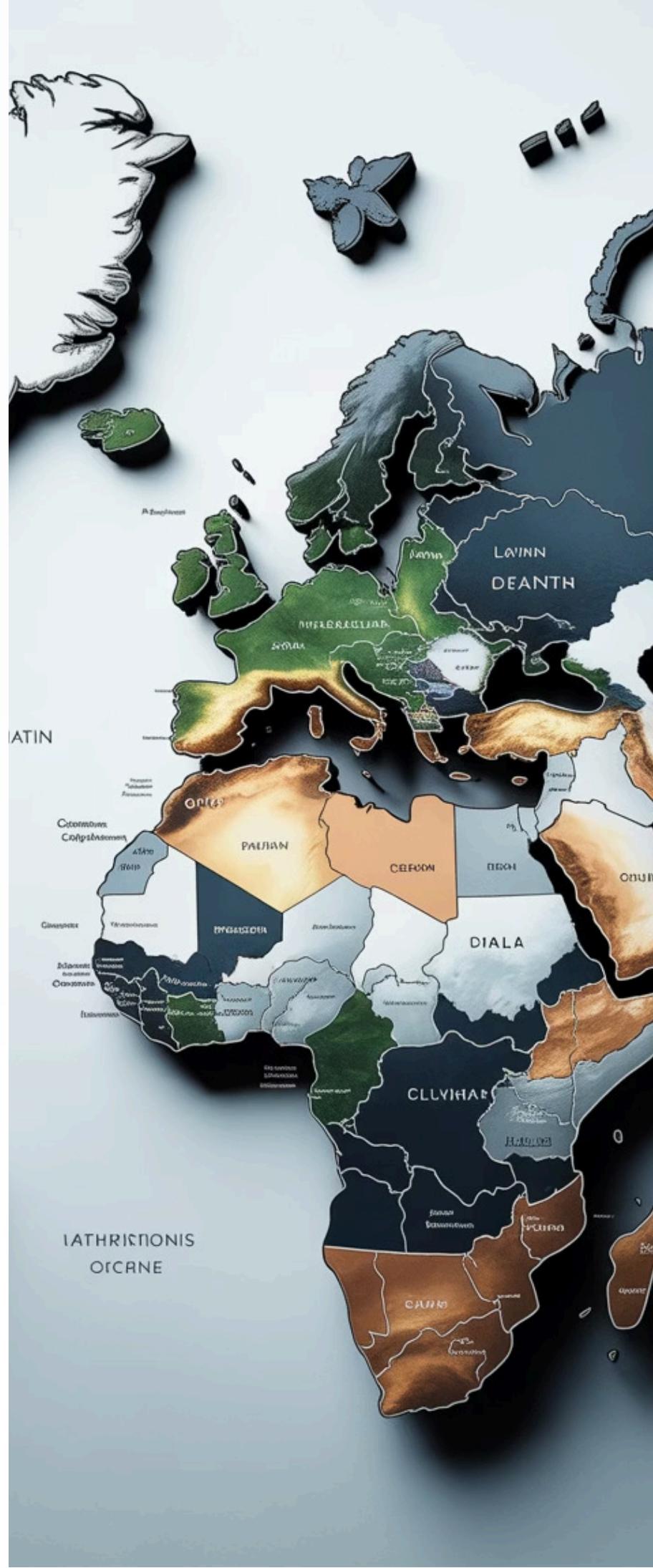
Utiliser Collator, pas compareTo()



ResourceBundle

```
ResourceBundle bundle =
ResourceBundle.getBundle(
"messages", locale);
String msg = bundle.getString(
"greeting");
```

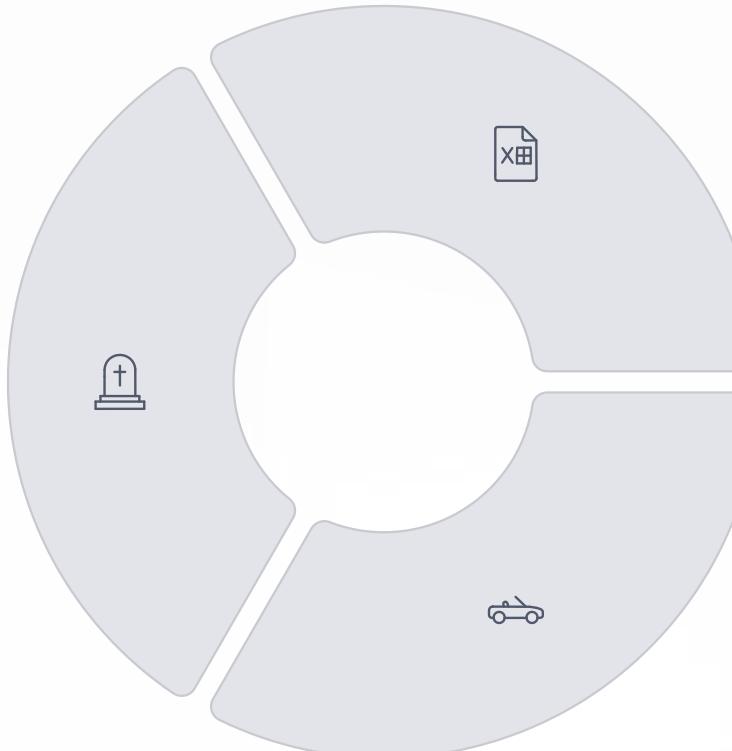
Externaliser les String traduisibles



Encodage : UTF-8, UTF-16

UTF-16 (interne)

Java utilise UTF-16 en mémoire. 2 bytes par caractère BMP, 4 bytes pour caractères rares (emojis)



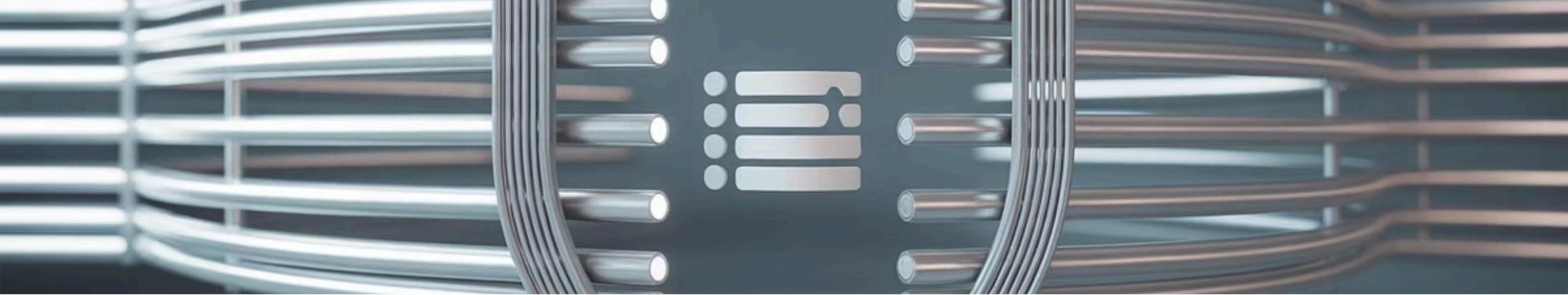
UTF-8 (I/O)

Standard pour fichiers et réseaux. Taille variable : 1-4 bytes. Plus compact pour texte occidental

Conversion

getBytes() et new String(bytes, charset).
Toujours spécifier
StandardCharsets.UTF_8

```
byte[] bytes = text.getBytes(StandardCharsets.UTF_8);
String decoded = new String(bytes, StandardCharsets.UTF_8);
```



Sécurité : Injections et validations

Injection SQL

1 **Dangereux** : "SELECT * FROM users WHERE name = '" + username + "'"

Sécurisé : PreparedStatement avec paramètres

Cross-Site Scripting (XSS)

2 **Dangereux** : Affichage direct de userInput

Sécurisé : StringEscapeUtils.escapeHtml4()

Path Traversal

3 **Dangereux** : new File(baseDir + "/" + filename)

Sécurisé : Validation et normalisation avec Path

□ **Principe de défense en profondeur** : Validation en entrée, échappement en sortie, API sécurisées



Pattern : String immutable pour clés

Pourquoi String est idéale

- **Immutabilité :**
hashCode ne change jamais
- **hashCode() optimisé :**
Calcul efficace et mis en cache
- **equals() fiable :**
Comparaison correcte
- **Thread-safe :** Sans synchronisation

```
Map cache =  
new HashMap<>();  
cache.put("user123", userObject);
```

```
String key = "config.database.url";  
int hash = key.hashCode();  
// Mis en cache
```

```
// ÉVITER objets mutables  
// StringBuilder sb =  
// new StringBuilder("key");  
// map.put(sb, value);  
// sb.append("X"); // DANGER!
```



Comparaison lexicographique : compareTo()

1

Résultat négatif

String appelante avant le paramètre

```
"apple".compareTo("banana")  
// -1
```

2

Résultat zéro

String exactement identiques

```
"test".compareTo("test")  
// 0
```

3

Résultat positif

String appelante après le paramètre

```
"zebra".compareTo("apple")  
// 25
```

```
// Tri insensible à la casse  
Collections.sort(names, String.CASE_INSENSITIVE_ORDER);
```

```
// Tri culturellement correct  
Collator collator = Collator.getInstance(Locale.FRENCH);  
names.sort(collator);
```



Antipattern : Mots de passe en String

✗ Dangereux

```
public boolean authenticate(  
    String password) {  
    return  
    checkPassword(password);  
    // Impossible d'effacer  
}  
  
String userPassword =  
"secret123";  
// Persiste en mémoire  
// Visible dans heap dumps
```

String reste en mémoire jusqu'au GC, créant fenêtre d'exposition au risque

✓ Sécurisé

```
public boolean authenticate(  
    char[] password) {  
    try {  
        return  
        checkPassword(password);  
    } finally {  
        Arrays.fill(password, '\0');  
    }  
  
    char[] pwd = getPassword();  
    try {  
        authenticate(pwd);  
    } finally {  
        Arrays.fill(pwd, '\0');  
    }
```

char[] peut être explicitement effacé, minimisant exposition



Joining et Collectors : Java 8+

String.join() - API statique

```
String result = String.join(", ",  
    "Java", "Python", "C++");  
// "Java, Python, C++"
```

```
List langs =  
    List.of("Java", "Python");  
String joined =  
    String.join(" | ", langs);
```

Idéal pour cas simples avec délimiteur fixe

Collectors.joining() - Avec Streams

```
String result = languages.stream()  
.collect(Collectors.joining(", "));
```

```
String list = languages.stream()  
.collect(Collectors.joining(  
    ", ", "[", "]"));  
// "[Java, Python, C++]"
```

Puissant pour pipelines de transformation complexes

Pattern : Null-safe operations



Yoda conditions

```
if  
("admin".equals(userInpu  
t)) {  
    // Retourne false si null  
}
```



Objects.equals()

```
if (Objects.equals(str1,  
str2)) {  
    // Gère les deux null  
}
```



Valeurs par défaut

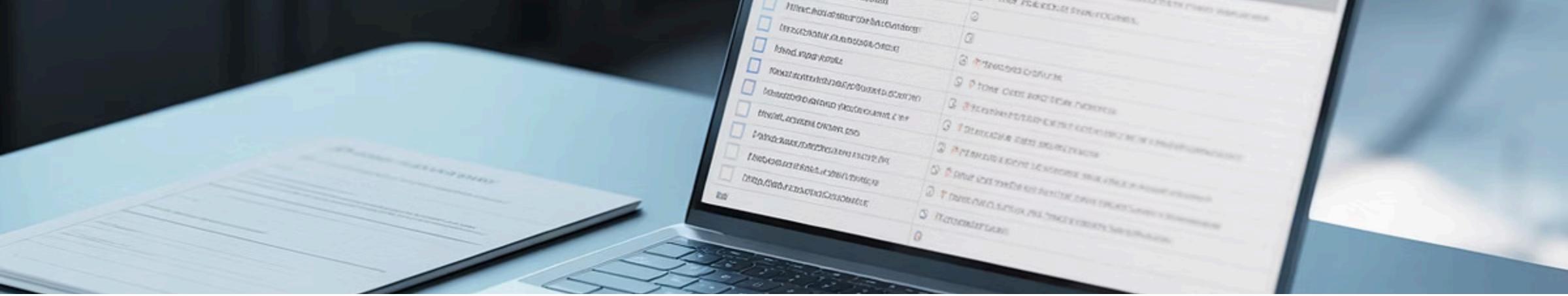
```
String safe =  
    Objects.requireNonNullEl  
    se(  
        input, "défaut");
```



Optional pour API

```
public Optional  
findUsername(int id) {  
    return  
        Optional.ofNullable(  
            database.lookup(id));  
}
```

Apache Commons Lang : StringUtils.isEmpty(), StringUtils.isBlank(), StringUtils.defaultString()



Tests unitaires : Best practices

01

Cas normaux

Scénarios typiques avec valeurs représentatives

02

Cas limites

Null, vide, espaces uniquement, très long, caractères spéciaux

03

Cas erronés

Entrées invalides, formats incorrects, valeurs hors limites

04

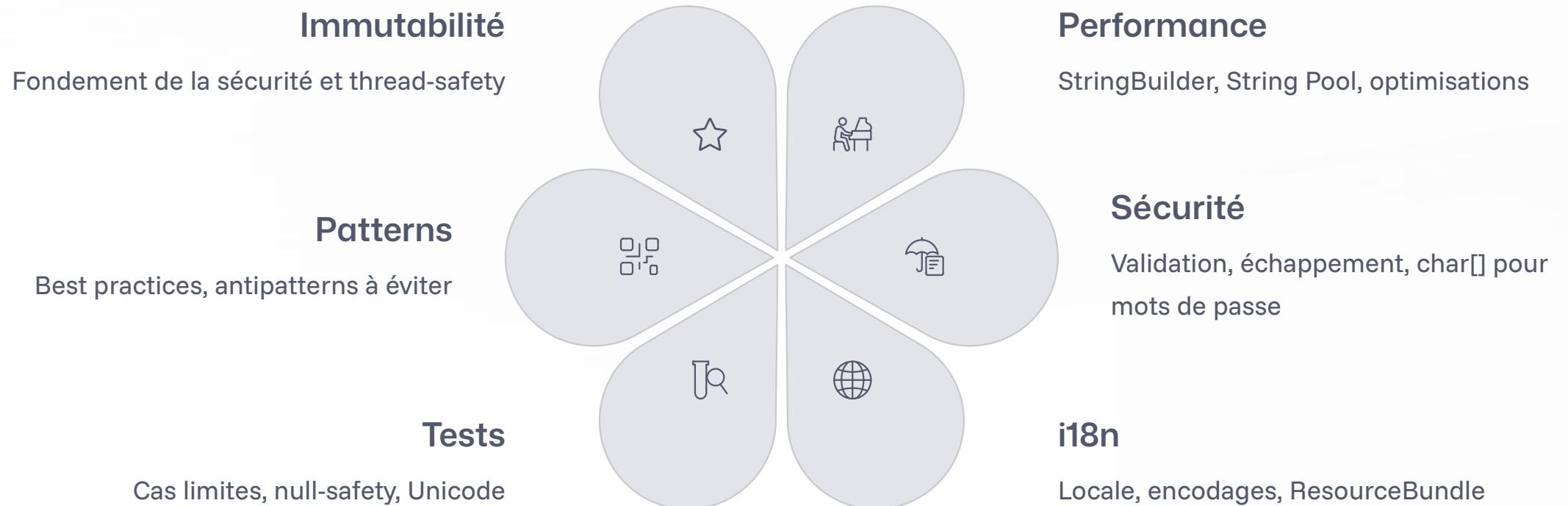
Cas d'encodage

Caractères Unicode variés, emojis, langues diverses

```
@Test  
public void testEmailValidation() {  
    assertTrue(serializer.isValidEmail("user@example.com"));  
    assertFalse(serializer.isValidEmail(null));  
    assertFalse(serializer.isValidEmail(""));  
    assertTrue(serializer.isValidEmail("用户@例子.jp"));  
}
```

- ❑ **Règle :** Testez avec null, "", " " (espaces), et valeurs très longues (> 10 000 caractères)

Conclusion : Vers la maîtrise de String



La maîtrise de String transforme un développeur intermédiaire en expert Java capable d'écrire un code robuste, performant et maintenable

• Approfondissement

Documentation Oracle, Effective Java, JEP

• Pratique

Appliquer dans projets réels, refactorer code legacy

• Partage

Code reviews, standards d'équipe, formation