

# Bugs en java

Philippe Prados

[pp@philippe.prados.name](mailto:pp@philippe.prados.name)



*Préservez l'environnement,  
n'imprimez pas ce document*

## TABLE DES MATIERES

1.	Invocation d'une méthode non final dans un constructeur. ....	3
2.	Comparaison de chaîne et valeur null .....	3
3.	Comparaison de chaîne à l'aide de ==.....	4
4.	Surcharge de equals() .....	4
5.	Conteneur statique non multi-tâches.....	4
6.	Catch Throwable .....	4
7.	Propagation des exceptions.....	5
8.	Short.....	5
9.	Serialisation .....	5
10.	Static final.....	5
11.	Affectation .....	5
12.	Conversions de format .....	6
13.	Classe utilitaire.....	6
14.	Création de Strings en cascade.....	7
15.	Initialisation de String.....	7
16.	Concaténation de String.....	7
17.	Libération des ressources.....	8
18.	Utilisation de type générique .....	8

## Avant-propos

*Ce document reprend les erreurs classiques rencontrées dans les applications java.*

Lors de différents audits d'applications, j'ai eu l'occasion d'identifier de nombreuses maladroites dans le code Java, générant généralement des problèmes de performances ou des bugs aléatoires. Des rédactions malheureuses sont parfois symptomatiques d'une conception risquée. Nous allons reprendre certains points pour expliquer en quoi le code est erroné ou faible en performance et comment y remédier.

### 1. INVOCATION D'UNE METHODE NON FINAL DANS UN CONSTRUCTEUR.

Une méthode non finale peut être surchargée dans une sous classe. Si un constructeur invoque une méthode non finale, il y a un risque d'erreur subtile lors de l'invoque de la méthode surchargée.

```
public class A
{
    public A() // Constructeur
    {
        setAttribut("MaValeur");
    }
    public void setAttribut(String attr)
    {
        ...
    }
}
class B
{
    Hashtable cache;
    B()
    {
        super();
        initCache();
    }
    private void initCache()
    {
        ...
    }
    public void setAttribut(String attr)
    {
        // Use cache...
    }
}
```

Le constructeur de la classe **A** invoque la méthode `setAttribut()`. Celle-ci est abstraite ou n'est pas déclarée `final`. La sous-classe **B**, héritant de **A**, peut redéfinir cette méthode. Dans l'exemple indiqué, la nouvelle version de la méthode `setAttribut()` a besoin qu'un cache soit initialisé. Le cache est initialisé dans le constructeur de **B**. A première vue, le code est correct. Mais, java, contrairement à C++, ouvre le polymorphisme dès la création de l'instance. Lors de la création d'une instance **B**, le constructeur de **A** est appelé en premier. Celui-ci invoque la méthode `setAttribut()` de la classe **B**, avant que le cache soit initialisé. Ensuite, le constructeur de **B** est invoqué, permettant l'initialisation du cache. Cela ne correspond pas à ce qu'imaginait le développeur.

Si le concepteur de la classe **A** est différent du concepteur de la classe **B**, l'erreur n'est pas visible tant que personne redéfinit une méthode invoquée par un constructeur supérieur ayant un besoin d'initialisation de l'instance. Mais, plus tard dans l'évolution du projet, cette erreur peut apparaître, remettant en cause la conception de la classe **A**.

Solution : Il ne faut jamais invoquer de méthodes non finales dans un constructeur. L'ajout de l'attribut `final` aux méthodes invoquées, garanti qu'une sous-classe ne se fera pas piégées.

```
public class A
{
    public A() // Constructeur
    {
        setAttribut("MaValeur");
    }
    public final void setAttribut(String attr)
    {
        ...
    }
}
```

Il est même recommandé de déclarer toutes les méthodes finales tant qu'il n'est pas démontré que la surcharge est nécessaire.

### 2. COMPARAISON DE CHAÎNE ET VALEUR NULL

Pour comparer la valeur d'une chaîne, on utilise souvent `obj.equals("valeur")`. Que se passe-t'il si `obj` est à `null` ? Il y a alors une exception `NullPointerException` de levée. Si ce cas se présente, les développeurs ajoutent généralement un test initial pour éviter cette situation limite.

```
((obj !=null) && (obj.equals("valeur")))
```

Solution : Les développeurs ignorent généralement qu'une chaîne de caractère en java est un objet comme un autre. Il est possible d'y appliquer des méthodes. Cela permet une écriture plus efficace est moins complexe. A la place de tester que `obj` possède la même valeur qu'une chaîne, il est plus rapide de vérifier qu'une chaîne possède la même valeur qu'`obj`.

```
"valeur".equals(obj)
```

Ainsi, si `obj` est à `null`, la méthode retournera `false`, sans test supplémentaire. Attention, il y un seul cas où est il nécessaire de maintenir l'écriture avec deux tests, c'est lorsque la chaîne à comparer est la chaîne vide. En effet, `"".equals(obj)` donne `false` si `obj` est à `null`. Ce n'est pas toujours ce qui est souhaité par le développeur. `null` peut être assimilé à la chaîne vide.

### 3. COMPARAISON DE CHAINE A L'AIDE DE ==

Java proposent deux types de comparaisons : vérifier l'identité des instances, ou vérifier la valeur des instances. Ces vérifications, très proches sémantiquement, utilisent des syntaxes différentes, parfois confondues par les développeurs.

La comparaison de référence à l'aide de `==` permet de vérifier si les deux pointeurs pointent sur le même objet, pas si les deux objets ont les mêmes valeurs. Pour comparer la valeur de deux objets il faut utiliser la méthode `equals()`. Attention, la version par défaut de cette méthode transforme la comparaison de valeurs par la comparaison d'identité.

La classe `String` redéfinit cette méthode pour pouvoir comparer deux chaînes par valeurs.

Parfois, dans le code des applications, il y a des `str == ""`. Cela ne vérifie pas si `str` est vide, mais si `str` pointe sur l'instance générée par le compilateur pour porter la valeur `"`.

Par chance, ce code peut fonctionner. En effet, si `str` a été initialisé dans la même classe, il est possible que l'identité soit équivalente à la comparaison de valeurs.

Java utilise un pool de constante par classe. Donc, si la chaîne `"` est présente à plusieurs endroit de la classe, il n'y aura qu'une instance dans le pool. Si la variable `str` est initialisée par une constante dans la même classe, le pool jouera son rôle et utilisera une et une seule instance générée par le compilateur. Par contre, si `str` est initialisée par une instruction `substring()` ou dans une autre classe, l'utilisation de l'opérateur de comparaison ne sera pas pertinent.

Solution : utilisez `str.equals("")`. Mieux, utilisez `"".equals(str)`

### 4. SURCHARGE DE EQUALS()

Parfois, il est nécessaire de surcharger la méthode `equals()`. Cela permet de comparer des instances par leurs valeurs ou de les utiliser comme clef dans un dictionnaire. Cette méthode est intimement liée avec la méthode `hashCode()`.

Solution : Dans tous les cas où la méthode `equals()` est redéfinie, il faut également redéfinir la méthode `hashCode()` afin de bénéficier des optimisations des dictionnaires. Cette méthode doit exploiter les valeurs de `hashCode()` de tous les attributs utilisés par la méthode `equals()`.

### 5. CONTENEUR STATIQUE NON MULTI-TACHES

Java est un langage multi-tâches. Lors des premières versions du JDK, les API proposées respectaient cette contrainte. Les différentes classes de conteneurs (`Vector`, `Hashtable`, ...) étaient toutes compatibles avec les multiples tâches. Ce choix avait un impact sur les performances, car tous les conteneurs utilisés, devaient synchroniser les traitements. Une évolution a été proposée pour offrir des conteneurs plus rapides, mais non compatibles avec les multiples tâches.

On rencontre parfois dans les applications des conteneurs statiques dont les classes utilisées ne sont pas multi-tâches. Cela entraîne des erreurs très difficiles à identifier car elles sont aléatoires, ayant des effets de bords variables, et concernent des classes du JDK qu'il est difficile de remettre en cause.

```
class A
{
    private static HashMap cache=new HashMap();
    ...
}
```

Solution : changez le type de conteneur pour un conteneur du JDK 1.0 ou utilisez la méthode `synchronizedMap(Map m)` de la classe `Collections`.

```
class A
{
    private static HashMap cache=Collections.synchronizedMap(new HashMap());
    ...
}
```

### 6. CATCH THROWABLE

L'exception `Throwable` est la mère de toutes les exceptions. Capturer une exception de cette catégorie est rarissime, car tous y passe : les erreurs de format des fichiers `.class`, la saturation de la mémoire de l'application, l'utilisation malheureuse de pointeur `null`, les débordements dans l'utilisation des tableaux, les récursivités infinies et les erreurs applicatives. Si une fuite mémoire entraîne l'invocation de l'exception `OutOfMemoryError` l'application va capturer le problème et continuer son traitement. La cause réelle du problème sera masquée, entraînant des erreurs en cascades qu'il sera difficile d'analyser.

Solution : Capturez uniquement et si nécessaire les exceptions pouvant être émises. Pour cela, il faut indiquer une à une, dans chaque méthode, les exceptions pouvant remonter. Le compilateur se chargera de vous signaler tous les chemins de l'application où il est nécessaire de capturer ou de propager les exceptions.

Ne capturez jamais une exception sans y apporter un traitement pertinent.

## 7. PROPAGATION DES EXCEPTIONS

Les développeurs ne voulant pas se soucier de la gestion des exceptions indiquent parfois qu'une méthode peut émettre `Throwable` ou `Exception`. Cela ne permet pas d'identifier précisément les cas pouvant réellement arriver. L'application est alors obligée de capturer ces exceptions, avec tous les risques évoqués précédemment.

Solution : Au même titre qu'il faut importer spécifiquement les classes utilisées dans les imports (et éviter les imports avec étoile), il faut indiquer précisément les exceptions pouvant être émises par une classe. Supprimez les `throws Exception` des méthodes. Regardez le comportement de la compilation. S'il n'y a pas d'erreur, laissez la méthode sans exceptions, même si la super classe le déclare autrement. S'il y a des erreurs de compilation, ajoutez la clause `throws` à la méthode plutôt que d'encadrer le code de `try/catch`.

## 8. SHORT

Java utilise le `short` pour réduire l'espace mémoire occupé par les instances, et non pour optimiser le calcul. Au contraire. La JVM ne possède pas d'arithmétique en short. Le P-code doit convertir les `shorts` en `int`, faire le calcul et convertir le résultat en `short`.

Solution : ne pas utiliser `short` en java, sauf si l'impact mémoire est très important.

## 9. SERIALISATION

Les instances placées dans la session doivent être toutes sérialisables car les serveurs J2EE ne garde en mémoire qu'une dizaine de contexte. S'il y en a plus, ils sérialisent les plus vieux sur disque et les remontent lorsque c'est nécessaire. Utiliser un objet non sérialisable dans la session entraîne que l'instance disparaît lors du rechargement de la session, pouvant entraîner des erreurs difficiles à reproduire en environnement de test.

Solution : Implémentez `Serializable` pour toutes les instances présentes de la session HTTP.

## 10. STATIC FINAL

Il est d'usage de déclarer des constantes à l'aide de l'attribut final est d'un nom en majuscule.

```
class A
{
    private final String TOTO="ABCDEF";
    ...
}
```

Cette rédaction est correct à l'exécution mais consomme une place mémoire inutile à l'exécution. En effet, l'attribut n'étant pas `static`, une instance de `String` est présente dans chaque instance de la classe `A`. Le compilateur génère en fait le code suivant :

```
class A
{
    final String TOTO;
    A()
    {
        TOTO="ABCDEF";
        ...
    }
}
```

L'affectation est présente dans tous les constructeurs de la classe. Il y a parfois des cas plus graves :

```
class A
{
    private final String[] TOTO={"ABC", "DEF", "GHI"};
    ...
}
```

Dans cas, un tableau de trois élément et créé et initialisé pour chaque instance de `A` !

Solution : Il faut vérifier et déclarer `private static final` pour toutes les constantes.

## 11. AFFECTATION

Les classes java utilisent souvent le même nom pour les attributs et les paramètres. Pour lever l'ambiguïté lors de la compilation, le préfixe `this` est utilisé lors de l'exploitation des attributs. Cet usage entraîne parfois des erreurs difficiles à identifier. Par exemple, un constructeur reçoit plusieurs paramètres pour initialiser tous les attributs.

```
class A
{
```

```

private Object a;
private Object b;
private Object c;
A(Object a, Object b)
{
    this.a=a;
    this.b=b;
    this.c=c;
}
}

```

Contrairement à ce que laisse supposer une lecture rapide du code, l'attribut `c` n'est pas initialisé. En effet, il n'existe pas de paramètre `c` dans le constructeur. Le compilateur comprend alors

```

class A
{
    private Object a;
    private Object b;
    private Object c;
    A(Object a, Object b)
    {
        this.a=a;
        this.b=b;
        c=c;
    }
}

```

L'attribut est initialisé avec lui-même. Cette affectation ne sert à rien et laisse `c` avec la valeur par défaut.

Solution : Il est préférable d'utiliser des noms différents entre les attributs et les paramètres. Une approche consiste à ajouter un préfixe ou un suffixe aux attributs.

```

class A
{
    private Object a_;
    private Object b_;
    private Object c_;
    A(Object a, Object b)
    {
        a_=a;
        b_=b;
        c_=c; // Erreur de compilation !
    }
}

```

## 12. CONVERSIONS DE FORMAT

Java propose des méthodes statiques permettant de convertir des types primitifs en chaîne et réciproquement. Il n'est pas nécessaire de construire une instance pour invoquer une méthode statique. Les développeurs débutant construisent parfois des instances inutiles pour invoquer une méthode statique.

```

String s=new String().valueOf(var);
int i=new Integer(str).intValue();

```

La création des instances `String` et `Integer` est inutile.

Solution :

```

String s=String.valueOf(var);
int i=Integer.parseInt(str);

```

## 13. CLASSE UTILITAIRE

Des classes ne proposent que des méthodes statiques. Il s'agit de classe utilitaire. Il n'est pas cohérent de construire une instance d'une de ces classes. Parfois, les développeurs débutants, ne sachant pas comment invoquer les méthodes statiques, construisent une instance inutile.

```

new Utilitaire().f(3);

```

Solution : Pour éviter ces utilisations malheureuses, il est préférable d'ajouter un constructeur privé à toutes les classes utilitaires.

```

final class Utilitaire
{
    private Utilitaire(){}
    public static void f(int a)
    {
        ...
    }
}

```

Ainsi, une erreur de compilation indiquera la bonne démarche de développement. Déclarez également la classe en `final` puisqu'elle ne contient que des méthodes statiques

## 14. CREATION DE STRINGS EN CASCADE.

Les instances `Strings` sont immuables. Elles ne peuvent évoluer après leurs constructions. Il n'est donc pas nécessaire de construire une nouvelle instance à partir d'une autre. On trouve parfois un code comme ceci : `var=new String(autreString)`. La création d'une nouvelle instance pour la donner à la variable `var` est inutile. On peut même s'interroger de la pertinence du constructeur `String()` acceptant une instance `String` en paramètre.

Pour les experts, il est vrai que parfois, cela peut être nécessaire. En effet, lors de l'invocation de la méthode `substring()`, une nouvelle instance est créée qui fait directement référence au tableau de chaîne de caractère de la chaîne originale.

```
String sub="AAA...chaîne très longues...AAA".substring(10,15);
```

La variable `sub` pointe sur la très longue chaîne et indique le début à la fin à prendre en compte. En mémoire, la très longue chaîne est présente tant que `sub` vit encore. Imaginons que la longue chaîne soit un flux XML complet, dont un petit extrait nous intéresse. Si le développeur désire ne garder en mémoire que l'extrait, il est pertinent de dupliquer l'instance `string`.

```
String sub=new String("AAA...chaîne très longues...AAA".substring(10,15));
```

Ainsi, la très longue chaîne est supprimée par le ramasse-miette. Seul l'extrait est en mémoire.

Ce cas aux limites étant très rare, toutes invocations du constructeur de `String` avec une autre `String` est à proscrire.

Solution : Utilisez l'affectation. `var=autreString`

## 15. INITIALISATION DE STRING.

De même, les développeurs construisent parfois une instance `String` à partir d'une constante chaîne de caractère.

```
var=new String("abc").
```

Cette écriture est une invocation subtile du constructeur de copie de `String`. La chaîne `"abc"` est déjà une instance `String`. Le seul intérêt de cette écriture est de garantir que l'identité entre `var` et la constante est différente.

```
var=new String("abc").
var != "abc"
```

Solution : Utilisez l'affectation.

```
var="abc"
```

## 16. CONCATENATION DE STRING.

Les méthodes de la classe `StringBuffer` retournent `this`. Il est donc possible d'enchaîner les invocations. C'est ce que fait le compilateur. On trouve souvent un code de construction de chaîne de caractère dont l'instance retournée par les méthodes `append()` n'est pas utilisée.

```
buf.append("<xml>") ;
buf.append(var) ;
buf.append("</xml>");
```

Cela entraîne que le compilateur doit générer un code pour effacer de la pile l'objet retourné par le premier `append()`, puis rechercher la valeur de la variable `buf`, la placer sur la pile d'exécution, et enfin invoquer la deuxième méthode `append()`.

Il est plus efficace de cascader les invocations.

```
buf.append("<xml>")
    .append(var)
    .append("</xml>") ;
```

Ainsi, l'instance présente dans la pile après l'invocation du premier `append()` est directement utilisée pour invoquer le deuxième. Lors de l'addition de chaînes de caractères par le compilateur, le code généré est très similaire.

```
"<xml>"+var+"</xml>"
```

est généré en

```
new StringBuffer()
    .append("<xml>")
    .append(var)
    .append("</xml>")
    .toString();
```

La méthode `toString()` est invoquée à la fin de la concaténation.

## 17. LIBERATION DES RESSOURCES

Les ressources fichiers, réseaux ou connexion à la base de données sont limitées. Si l'application ne ferme pas correctement les fichiers dans tous les cas, il y a une fuite de ressource, entraînant le plantage à terme de l'application. Il est indispensable de fermer les flux dans un bloc `finally`. Ainsi, quoi qu'il arrive, la ressource est libérée et peut servir à d'autres applications du système d'information.

Solution : Le modèle à suivre pour toutes utilisation des flux est le suivant :

```
void f(String name) throws IOException
{
    InputStream in=null;
    try
    {
        in=new FileInputStream(name);
        // ...
    }
    finally
    {
        if (in!=null)
            in.close();
    }
}
```

## 18. UTILISATION DE TYPE GENERIQUE

Java propose différents conteneurs répondant à des interfaces. L'intérêt de cette approche est de pouvoir sélectionner plus tard l'implantation exacte de l'interface afin de bénéficier de différentes optimisations.

```
class A
{
    Hashtable data_=new Hashtable()
    public void setData(Hashtable data)
    {
        data_=data;
    }

    public Hashtable getData()
    {
        return data_;
    }
}
```

Solution : Pour tirer bénéfice de cette conception, il est important de n'utiliser que les types les plus génériques pour tous les conteneurs.

```
class A
{
    Map data_=new Hashtable()
    public void setData(Map data)
    {
        data_=data;
    }

    public Map getData()
    {
        return data_;
    }
}
```

Ainsi, il est possible d'invoquer la méthode `setData()` avec une instance `TreeMap` si le contexte s'y prête ou de modifier l'initialisation de la variable `data_` pour utiliser un autre dictionnaire plus pertinent. Cela n'a alors pas d'impact sur les classes manipulant `A`.

Philippe Prados

3/2004