Tutoriel pour apprendre à coder le patron de conception Singleton en Java

Par François-Xavier Robin

Date de publication : 5 novembre 2018

L'objectif de ce billet est de présenter une implémentation simple en Java du pattern Singleton.

Je ne traiterai donc pas de l'utilité des recommandations d'usage liées à celui-ci, mais bien de la mise en œuvre (codage pour les intimes) en Java Standard Edition (JSE).

Pour réagir à cet article, un espace de dialogue vous est proposé sur le forum **Commentez**.

I - Le constat

Il est vraiment frappant en consultant les sites spécialisés en développement Java de constater à quel point le **Design Pattern Singleton** est dans le top 3 des patterns les plus abordés.

De la même façon, il est encore plus frappant de voir à quel point toutes ces ressources en ligne, toutes ces explications sur l'unicité en mémoire, le *double-check-locking* prennent une part importante. Même sur des sites pourtant reconnus comme DZONE, on y trouve des articles erronés, car il n'y a **rien de plus simple que d'écrire un bon singleton Thread-Safe et Lazy en Java : mais je le garde pour la fin de ce billet…**

II - Les origines du Design Pattern Singleton

Au préalable, revenons aux origines du Singleton, alors que Java n'existait pas encore, et à ses principales caractéristiques :

- Un singleton, c'est un objet construit conformément à sa classe et dont on a la garantie qu'il n'existe qu'**une et une seule instance** en mémoire à un instant donné.
- En cas d'accès concurrent lors de l'instanciation d'un singleton, il faut veiller à ce que cet aspect soit pris en compte par un **mécanisme de verrous.**
- En général, on souhaite que le singleton ne s'initialise pas entièrement, mais seulement à son premier appel, afin d'économiser de la mémoire. On appelle cela le mécanisme « *lazy* ».

Ainsi le GoF, propose son pattern singleton. Et certains l'appliquent alors en C++.

III - Et la plateforme Java alors ?

Java arrive alors sur le marché, ressemblant tellement à C++ sur sa syntaxe que le singleton du GoF, façon C++, est tout simplement imité sans prendre en compte les spécificités de la plateforme Java :

- une classe n'est chargée que lors de son premier appel ;
- le chargement d'une classe est *thread-safe*, c'est un mécanisme garanti par la hiérarchie de *ClassLoader*s de la JVM.

Ce qui permet d'envisager déjà que :

- le singleton version Java, sera forcément Lazy ;
- l'instanciation statique du singleton version Java, sera forcément ThreadSafe ;
- toute autre tentative de ne pas se reposer sur ces caractéristiques apportera un code plus lourd, inutile et potentiellement buggé. (Less Code, Less Bug !) ;
- un singleton version Java SE avec une seule hiérarchie de ClassLoader sera seul en mémoire JVM.

IV - Et « un lazy thread-safe, un… »

Le voilà, notre beau Singleton Lazy Thread-Safe :

```
1. public class LazySingleton
2. {
3.     private static final LazySingleton instance = new LazySingleton();
4.
5.     private LazySingleton()
6.     {
7.         System.out.println("Construction du Singleton au premier appel");
8.     }
9.
10.     public static final LazySingleton getInstance()
```

```
11.     {
12.         return instance;
13.     }
14.
15.     @Override
16.     public String toString()
17.     {
18.        return String.format("Je suis le LazySingleton : %s", super.toString());
19.     }
20. }
```

Voici un programme qui en obtient une instance :

```
1. public class MainProg
2. {
3.     public static void main(String[] args)
4.     {
5.         System.out.println("Démarrage du programme");
6.         System.out.println("Mon singleton n'est toujours pas chargé ...");
7.         System.out.println("Bon allez, je me décide à l'appeler ...");
8.         LazySingleton singleton = LazySingleton.getInstance();
9.         System.out.println("Et maintenant je l'affiche ...");
10.         System.out.println(singleton);
11.     }
12. }
```

et voici le résultat de son exécution qui prouve bien son chargement « *lazy* » :

```
1. Démarrage du programme
2. Mon singleton n'est toujours pas chargé ...
3. Bon allez, je me décide à l'appeler ...
4. Construction du Singleton au premier appel
5. Et maintenant je l'affiche ...
6. Je suis le LazySingleton : demo.LazySingleton@7852e922
```

V - Et depuis Java 5, ça donne quoi ?

Enfin, depuis Java 5, c'est-à-dire fin 2004, une éternité, un singleton peut s'implémenter au moyen d'une « enum ». Petite limitation dans ce cas : on ne peut pas en hériter, mais en a-t-on souvent besoin ?

Version enum Java 5 :

```
1. public enum LazySingletonEnum
2. {
3.     INSTANCE;
4.
5.     private LazySingletonEnum()
6.     {
7.         System.out.println("Construction du LazySingletonEnum");
8.     }
9.
10.     public static LazySingletonEnum getInstance()
11.     {
12.         return INSTANCE;
13.     }
14.
15.     public String getMessage()
16.     {
17.         return String.format("Je suis le LazySingleton : %s", super.toString());
18.     }
19. }
```

et son usage :

```
System.out.println("Démarrage du programme");
System.out.println("Mon singleton n'est toujours pas chargé ...");
System.out.println("Bon allez, je me décide à l'appeler ...");
LazySingletonEnum singleton = LazySingletonEnum.getInstance();
System.out.println("Et maintenant je l'appelle  ...");
System.out.println(singleton.getMessage());
System.out.println("On peut aussi l'appeler directement : ");
System.out.println(LazySingletonEnum.INSTANCE.getMessage());
```

qui donne le résultat probant suivant :

```
Démarrage du programme
Mon singleton n'est toujours pas chargé ...
Bon allez, je me décide à l'appeler ...
Construction du LazySingletonEnum
Et maintenant je l'appelle  ...
Je suis le LazySingleton : INSTANCE
On peut aussi l'appeler directement :
Je suis le LazySingleton : INSTANCE
```

VI - Et la « serialization » entre en jeu…

Je n'ai pas non plus abordé un autre problème : souvent un Singleton a besoin d'être Serializable, mais de fait, la désérialisation d'un singleton permet de créer plusieurs instances. Ceci « casse » le principe du Singleton qui doit être unique.

Pour résoudre ce problème, il suffit de définir readResolve() dans le singleton lui-même.

```
public class Singleton implements Serializable
{
    private static Singleton singleton = new Singleton();

    private Singleton()
    {
       // protection
    }

     public static Singleton getInstance( )
     {
       return singleton;
     }

     public Object readResolve()
     {
        return Singleton.getInstance( );
     }
}
```

> À noter que cela n'est pas nécessaire dans le cas d'un singleton codé au moyen d'une enum. Attention aussi à vos objets que vous sérialisez peut-être et qui posséderaient une référence vers le singleton. Il vaut mieux dans ce cas placer le mot clé transient devant sa référence.

VII - Petit détour du côté des Servlets

Ce petit paragraphe sort du cadre de ce billet, puisque je ne souhaitais évoquer que le cas de Java SE et non pas Java EE.

Mais pour illustrer et pour ceux qui connaissent, voici ce que sont chacune des servlets déclarées dans une WebApp Java : un singleton ! Eh oui, *chaque Servlet est un singleton* ! Lazy de surcroît ! Ce qui valait d'ailleurs au premier

déclencheur (un navigateur via URL par exemple) d'avoir un temps d'attente un peu plus long que les clients suivants qui avaient alors la servlet chargée en mémoire et prête à répondre à la requête au moyen d'un Thread.

Idem pour les pages JSP, qui en plus passaient par une phase de compilation éventuelle, préalablement transformées en Servlet, toujours LAZY, car chargées seulement au premier appel.

Pour éviter ces effets d'attente et charger chaque servlet (singleton) dès le déploiement de l'application, il faut utiliser le fameux :

```
1. <load-on-startup>1</load-on-startup>
```

Cette configuration de la servlet dans le web.xml permet ainsi de la passer en mode « EAGER » (inverse de LAZY).

VIII - Un singleton ça offre quoi ?

Et maintenant que nous avons notre beau singleton, unique en mémoire, il faudrait quand même qu'il nous serve à quelque chose.

En général, on y conserve de l'information, partagée par l'ensemble des utilisateurs du système et/ou par l'ensemble des *threads*, il est accessible donc par n'importe quel code qu'il soit static ou d'instance au sein de méthodes.

Il faut donc faire très attention, tous les chargements, modifications, suppressions d'informations doivent être *thread-safe* ! Cela dépasse un peu l'objectif de ce billet, mais il va vous falloir gérer la synchronisation avec des verrous, des synchronized, ou mieux, n'utiliser que des classes thread-safe et celles de la *concurrency API,* par exemple :

- AtomicInteger ;
- Lock et ReentrantLock ;
- Collections.synchronizedList() ou Collections.synchronizedMap() ;
- etc.

IX - Et depuis Java 8 alors ?

Une question devrait vous tarauder :

> Mais jusqu'ici pourquoi avions-nous besoin d'un Singleton en lieu et place de simples champs static ?

Il s'agit d'une question de zone de mémoire de la JVM. Sans rentrer dans trop de détails, il faut simplement savoir que jusqu'à Java 7 inclus, les classes et les types primitifs static ainsi que les références static à des instances étaient stockés dans la zone nommée *Permanent Generation Space*.

Cette zone était limitée au démarrage de JVM et bien que paramétrable, elle ne pouvait pas s'étendre dynamiquement. Ainsi, il fallait prévoir au mieux : ni trop, ni trop peu. De plus et il s'agit du point clé, il fallait optimiser cet espace en y mettant le moins possible d'éléments static. D'où la nécessité d'un singleton avec, comme seule partie static, la référence vers son instance.

Cela a conduit bon nombre de sites fonctionnant sous Java EE à observer le fameux OutOfMemory : PermGen space. On triturait alors quelques paramètres de JVM (PermSize, MaxPermSize), mais au fil des redéploiements d'applications (surtout en DEV), le *Permanent Generation Space* se saturait et il fallait tout vider en relançant le serveur d'applications et donc en redémarrant la JVM : PAS BIEN.

En Java 8, bim, paf, badaboum, adieu le *PermGen Space*, bienvenue au **Meta Space**.

Le **Meta Space** est une zone mémoire qui appartient à la zone **HEAP**. C'est dans cette zone que l'ensemble des chargements se réalise depuis Java 8. Elle est *garbage collectée* ce qui signifie qu'elle est nettoyée quand les classes ne sont plus utilisées depuis un certain temps et les champs statiques sont libérés eux aussi par la même occasion. Enfin, la taille de cette zone est dynamique : finies les limitations. En résumé, un champ statique n'est plus coûteux « comme avant ».

Pourquoi alors s'enquiquiner avec un Singleton depuis Java 8 puisque maintenant les champs statiques ne posent plus de problèmes ?

Voici un « vieux » singleton Java 7 et son adaptation Java 8 qui offrent les mêmes fonctionnalités, sans impact mémoire.

Version Java 7 et - :

```
1. public class VisitCounter
2. {
3.     // implémenté sous forme de singleton //
4.
5.     private static VisitCounter singleton = new VisitCounter();
6.
7.     private AtomicInteger visitCounter = new AtomicInteger();
8.
9.     private VisitCounter()
10.     {
11.        // protection
12.     }
13.
14.     public static VisitCounter getInstance()
15.     {
16.       return singleton;
17.     }
18.
19.     public int getCounter()
20.     {
21.         return visitCounter.get();
22.     }
23.
24.     public int increment()
25.     {
26.         return visitCounter.incrementAndGet();
27.     }
28. }
```

Version Java 8 et + :

```
1. public final class VisitCounter
2. {
3.     private static AtomicInteger visitCounter = new AtomicInteger();
4.
5.     public static int getCounter()
6.     {
7.         return visitCounter.get();
8.     }
9.
10.     public static int increment()
11.     {
12.         return visitCounter.incrementAndGet();
13.     }
14. }
```

Il faut quand même préciser les inconvénients :

- les informations ne sont pas sérialisables ;
- les méthodes ne peuvent pas être redéfinies.

Mais justement, c'est assez intéressant ! Bon nombre de vieux singletons offrant des services devraient se voir refactoriser de la sorte ! Simplicité, efficacité. Mais seulement en Java 8 et + !

X - Grosse paresse ! (double lazyness)

On vient donc de voir que, par défaut, un singleton était « lazy » en Java. Pourquoi aller donc chercher encore plus loin la paresse avec la fameuse technique du **Holder interne** ?

Il peut arriver que l'on veuille un peu « discuter » avec le singleton avant le réel usage de celui-ci et donc économiser la RAM jusqu'au dernier moment.

Reprenons l'exemple précédent. Quelle que soit la version de Java, la solution repose sur la définition d'une classe interne. Cette classe interne statique sera chargée par le ClassLoader uniquement lors du premier appel à l'une des méthodes du singleton :

```
1. public class VisitCounter
2. {
3.     // implémenté sous forme de singleton //
4.
5.     private static class Holder
6.     {
7.         private static final VisitCounter singleton = new VisitCounter();
8.     }
9.
10.     private AtomicInteger visitCounter = new AtomicInteger();
11.
12.     private VisitCounter()
13.     {
14.        // protection
15.     }
16.
17.     public static Singleton getInstance()
18.     {
19.       return Holder.singleton;
20.     }
21.
22.     public int getCounter()
23.     {
24.         return visitCounter.get();
25.     }
26.
27.     public int increment()
28.     {
29.         return visitCounter.incrementAndGet();
30.     }
31. }
```

Grâce encore une fois à la nouvelle gestion de la mémoire en Java 8, c'est même encore plus simple, puisque le **Holder** contient les champs « utiles » :

```
1. public final class VisitCounter
2. {
3.     private static class Holder
4.     {
5.         private static final AtomicInteger visitCounter = new AtomicInteger();
6.     }
7.
8.     public static int getCounter()
9.     {
10.         return Holder.visitCounter.get();
11.     }
12.
13.     public static int increment()
14.     {
15.         return Holder.visitCounter.incrementAndGet();
```

```
16.     }
17. }
```

Ainsi, un VisitCounter.class ou une introspection de premier niveau de cette classe n'ira pas charger le **Holder** et donc n'ira pas initialiser les champs nécessaires au fonctionnement. D'ailleurs, on peut mixer « lazy » classique avec le **holder** pour les données les plus coûteuses.

J'appelle cela de la très grosse paresse…

XI - Il faut conclure

Je viens d'écrire ce que je m'étais pourtant interdit de faire : un énième billet sur le Singleton en Java venant s'ajouter à la quantité déjà astronomique de ceux qui existent sur le Net.

Ce qu'il faut retenir : vous n'aurez JAMAIS la garantie d'avoir une instance unique d'une classe en Java. Par introspection, par AOP, vous aurez toujours un moyen de casser l'unicité mémoire qu'on attend pourtant d'un singleton. Il suffit juste de faire un peu attention.

En guise de réelle conclusion, utilisez :

- @ApplicationScoped de CDI, que vous pouvez utiliser même en Java SE si vous prenez « Weld » dans vos dépendances ;
- @Scope("singleton") si vous utilisez Spring ;
- @Singleton de la spec EJB en environnement Java EE et vous serez définitivement tranquille.

Mais, de grâce, arrêtez de faire du *double-check locking* ! À part des ennuis, vous n'aurez rien à gagner !

XII - Remerciements

Cet article a été publié avec l'aimable autorisation de **François-Xavier Robin**.

Nous tenons à remercier **f-leb** pour sa relecture orthographique attentive de cet article et **Mickael Baron** pour la mise au gabarit.