

# Singleton

Philippe PRADOS

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



### ***Avant propos***

*Ce document explique les pièges du pattern singleton, quand et comment l'utiliser..*

Le pattern *Singleton* est souvent pratiqué dans le code. Une initialisation paresseuse est utilisée pour instancier l'instance au moment de la première invocation de la méthode `getInstance()`.

```
public class MaClass
{
    private MaClass()
    {
    }
    static private MaClass singleton_;
    public static MaClass getInstance()
    {
        if (singleton_==null)
        {
            singleton_=new MaClass();
        }
        return singleton_;
    }
    public void maMethode()
    {
        ...
    }
}
```

Si le singleton peut être accédé par plusieurs tâches, la méthode `getInstance()` est déclarée `synchronized`.

```
public class MaClass
{
    private MaClass()
    {
    }
    static private MaClass singleton_;
    public static synchronized MaClass getInstance()
    {
        if (singleton_==null)
        {
            singleton_=new MaClass();
        }
        return singleton_;
    }
    public void maMethode()
    {
        ...
    }
}
```

Pour éviter la dégradation des performances dues à l'attribut `synchronized`, la littérature préconise généralement d'utiliser le pattern double-check.

```
public class MaClass
{
    private MaClass()
    {
    }
    static private MaClass singleton_;
    public static MaClass getInstance()
    {
        if (singleton_==null)
        {
            singleton_=new MaClass();
        }
        return singleton_;
    }
}
```

```

{
    if (singleton_==null)
    {
        synchronized(MaClass.class)
        {
            if (singleton_==null)
                singleton_=new MaClass();
        }
    }
    return singleton_;
}
public void maMethode()
{
    ...
}
}

```

C'est approche est élégante, mais ne fonctionne qu'en mono-processeur.

En effet, l'ordre réel d'accès à la mémoire par deux processeurs peut être différent de ce que prévoit le programme. Des composants spécialisés s'occupent de réorganiser les lectures et les écritures en mémoires, afin de réduire le passage d'un état vers un autre. Les lectures sont regroupées ensemble, et les écritures également. Dans cette situation, rien ne garantit que la valorisation réelle en mémoire de l'attribut `singleton_`, sera effectué après la lecture de celui-ci à `null`. Du point de vu d'un processeur, l'ordre sera respecté, mais pas du point de vu d'un autre processeur.

Pour contrôler les frontières temporelles d'accès à la mémoire, il faut impérativement utiliser l'attribut `synchronized`, ce qui réduit les performances.

L'approche utilisée par l'instanciation paresseuse se justifie, si et seulement si, il existe plusieurs sous-classe de `MaClass`, dont une seule sera installée lors de la première invocation. Un paramétrage généralement indirect, permet de sélectionner la bonne sous classe à instancier.

```

public class MaClass
{
    protected MaClass()
    {
    }
    static private MaClass singleton_;
    public synchronized MaClass getSingleton()
    {
        if (singleton_==null)
        {
            switch(Paremeter.type)
            {
                case 0 :
                    singleton_=new SousClass1();
                    break;
                case 1 :
                    singleton_=new SousClass2();
                    break;
                ...
            }
        }
        return singleton_;
    }
    public void maMethode()
    {
        ...
    }
}

```

Dans les autres cas, il n'est pas nécessaire d'effectuer une initialisation paresseuse, comme nous allons le voir plus bas.

Les singletons proposent généralement un constructeur `private`, indiquant ainsi que la classe ne peut être surchargée. Un attribut `final` serait préférable car cela supprime l'utilisation du polymorphisme pour les méthodes du singleton.

```
final public class MaClass
{
    private MaClass()
    {
    }
    static private MaClass singleton_;
    public synchronized MaClass getSingleton()
    {
        if (singleton_==null)
        {
            singleton_=new MaClass();
        }
        return singleton_;
    }
    public void maMethode()
    {
        ...
    }
}
```

La classe ne pouvant être surchargée, elle ne correspond pas à la situation justifiant l'instanciation paresseuse.

Java charge les classes en mémoire lors de la première utilisation de celle-ci. Si un programme invoque la méthode `MaClass.getSingleton()`, la classe est chargée en mémoire. Il n'est alors pas nécessaire d'effectuer une instanciation paresseuse. L'initialisation de l'instance lors de la déclaration est suffisante.

```
final public class MaClass
{
    private MaClass()
    {
    }
    static private MaClass singleton_=new MaClass();
    public static MaClass getSingleton()
    {
        return singleton_;
    }
    public void maMethode()
    {
        ...
    }
}
```

Les classes ne pouvant être surchargées, il n'est pas nécessaire d'utiliser le polymorphisme. Dans ce cas, utiliser des méthodes statiques est plus efficace. En effet, pour invoquer un traitement avec un singleton `final`, il faut obtenir l'instance unique (invocation d'une méthode statique) pour invoquer une méthode sur cette instance.

```
MaClass.getSingleton().maMethode();
```

Il est plus rapide d'invoquer directement une méthode statique.

```
final public class MaClass
{
    public static void maMethode()
    {
        ...
    }
}
...
MaClass.maMethode();
```

Nous voyons que le pattern singleton est un faux ami. Les singletons se justifient, si et seulement si, il existe plusieurs sous-classes de `MaClass`, dont une seule sera installée lors de la première invocation. Cela permet de bénéficier de paramètres indirects pour sélectionner la bonne version du singleton.