
Le patron "Singleton"

Singleton :

Spécification (1 / 12)

- ▶ Objectif: Créer un type d'objet pour lequel on crée seulement une seule instance
- ▶ C'est le patron ayant le diagramme de classes le plus simple
- ▶ Il y a plusieurs objets dont on a besoin d'une seule instance: pool d'impression, boîte de dialogue, objet qui manipule les préférences, objet de logging, objet agissant comme pilote de carte graphique/imprimante...
- ▶ La création de plus d'une instance de ces objets est une source de problème, telle que la sur-utilisation des ressources, des comportements incorrectes de programme, des résultats inconsistants, etc.

Singleton :

Créer un singleton (2 / 12)

- ▶ Comment créer un seul objet?
 - ▶ New MonObjet()
- ▶ Et si un autre objet veut créer un MonObjet? Est-ce qu'il peut appeler new sur MonObjet une autre fois?
 - ▶ Oui
- ▶ Pour toute classe, est ce qu'on peut l'instancier plus qu'une fois?
 - ▶ Oui (il faut que la classe soit publique)
- ▶ Que signifie ce code ?
 - ▶ C'est une classe qui ne peut pas être instanciée, car elle possède un constructeur privé

```
public class MonObjet{  
    private MonObjet() {}  
}
```

- ▶ Qui peut utiliser ce constructeur?
 - ▶ Le code de MonObjet est le seul code qui peut l'appeler (dans une méthode)

Singleton :

Créer un singleton (3 / 12)

- ▶ Comment je peux appeler cette méthode (pour créer une instance) si je n'ai pas d'instance?

- ▶ static
- ▶ Que signifie ce code.

```
public class MonObjet{  
    public static MonObjet getInstance() {  
    }  
}
```

- ▶ C'est une méthode statique qui peut être appelée à partir du nom de la classe : **MonObjet.getInstance()**
- ▶ Si on met les choses ensemble, est ce qu'on peut instancier MonObjet?

```
public class MonObjet{  
    private MonObjet(){ }  
    public static MonObjet getInstance() {  
        return new MonObjet();  
    }  
}
```

- ▶ Comment faire pour créer une seule instance?

Singleton :

Implémentation du patron (4/12)

Nous avons une variable statique pour stocker notre instance

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

Le constructeur est déclaré privé. Seulement la classe Singleton qui peut instancier cette classe

Cette méthode nous offre une manière pour instancier la classe Singleton

Si uniqueInstance n'est pas à nul, ça veut dire qu'elle a été créée précédemment

```
public static void main(String args[]) {  
    Singleton s= Singleton.getInstance();  
}  
}
```

Singleton :

L'usine de chocolat (5/12)



```
public class ChocolateBoiler{
    private boolean empty;
    private boolean boiled;
    public ChocolateBoiler() {
        empty=true; boiled=false;
    }
    public void fill(){
        if (empty){
            System.out.println("remplir la casserole avec du lait/chocolat");
            empty=false; boiled=false;
        }
    }
    public void boil(){
        if (!empty && !boiled){
            System.out.println("faire bouillir");
            boiled=true;
        }
    }
    public void drain(){
        if (!empty && boiled){
            System.out.println("vider la casserole");
            empty=true;
        }
    }
}
```

Le code démarre lorsque la casserole est vide

Pour remplir la casserole, elle doit être vide. Lorsqu'elle est pleine, on met empty à false.

Pour mixer le contenu de la casserole, elle doit être pleine et non déjà mixée. Lorsqu'elle est pleine, on met boiled à true.

Pour vider la casserole, elle doit être pleine et déjà mixée. une fois vidée, on met empty à true.

Singleton :

L'usine de chocolat (6/12)



- Améliorer le code de l'usine de chocolat en le transformant en Singleton

```
public class ChocolateBoiler {  
    private boolean empty;  
    private boolean boiled;  
    private static ChocolateBoiler uniqueInstance;  
  
    private ChocolateBoiler() {  
        empty=true; boiled=false;  
    }  
  
    public static ChocolateBoiler getInstance() {  
        if (uniqueInstance == null) {  
            uniqueInstance = new ChocolateBoiler(); }  
        return uniqueInstance;  
    }  
    //reste du code...  
}
```

Singleton :

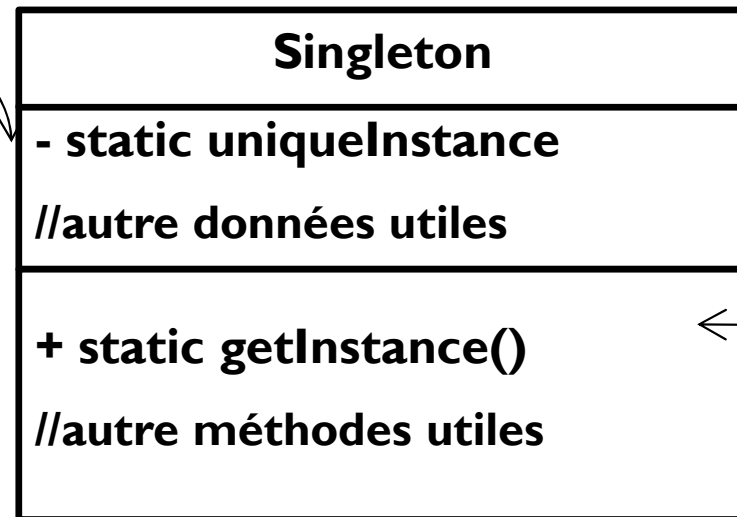
Le patron Singleton(7 / 12)

- ▶ Définition: **Singleton**
 - ▶ Le **patron Singleton** assure une seule instance pour une classe, et offre un point d'accès global à cette classe.

Singleton:

Le diagramme de classes du patron (8/12)

La variable de classe `uniqueInstance` tient la seule instance du Singleton



La méthode `getInstance()` est statique. C'est une méthode de classe qu'on peut y accéder partout dans le code avec `Singleton.getInstance()`. Il s'agit d'une instantiation facile de cette classe

Singleton :

Problème des threads (9/12)



- Supposant que nous avons deux threads qui vont exécuter la méthode `getInstance()`. Est-ce qu'il y a un cas où on crée 2 instances?

Thread-1	Thread-2	Valeur de <code>uniqueInstance</code>
<code>public static ChocolateBoiler getInstance()</code>		null
	<code>public static ChocolateBoiler getInstance()</code>	null
<code>if (uniqueInstance == null)</code>		null
	<code>if (uniqueInstance == null)</code>	null
<code>uniqueInstance = new ChocolateBoiler();</code>		Object1
<code>return uniqueInstance;</code>		Object1
	<code>uniqueInstance = new ChocolateBoiler();</code>	Object2
	<code>return uniqueInstance;</code>	Object2


Singleton :

Gestion du multi-threading (10/12)

- ▶ Solution 1: synchroniser l'accès à la méthode getInstance()

```
public class Singleton {  
    private static Singleton uniqueInstance;  
  
    private Singleton() {}  
  
    public static synchronized Singleton getInstance() {  
        if (uniqueInstance == null)  
            {uniqueInstance = new Singleton();}  
  
        return uniqueInstance;  
    }  
    //autres méthodes utiles  
}
```

Un seul thread peut accéder,
à la fois, à cette méthode



- ▶ Inconvénient: **synchronized** réduit la performance d'un facteur de 100
 - ▶ Si la méthode getInstance() n'est pas critique pour notre application, on peut se contenter de cette solution

Singleton :

Gestion du multi-threading (11/12)

- Solution 2: réduire l'utilisation de la synchronisation dans getInstance()

Le mot clé volatile assure que les threads gèrent la variable uniqueInstance correctement au moment de son initialisation

```
public class Singleton {  
    private volatile static Singleton uniqueInstance;
```

```
    private Singleton() {}
```

```
    public static Singleton getInstance() {  
        if (uniqueInstance == null) {  
            synchronized(Singleton.class) {  
                if (uniqueInstance == null) {  
                    uniqueInstance = new Singleton();  
                }  
            }  
        }  
    }
```

On synchronise seulement la première fois

```
        return uniqueInstance;
```

```
    }
```

```
    //autres méthodes utiles
```

```
}
```

***volatile**: inclus à java depuis jdk5

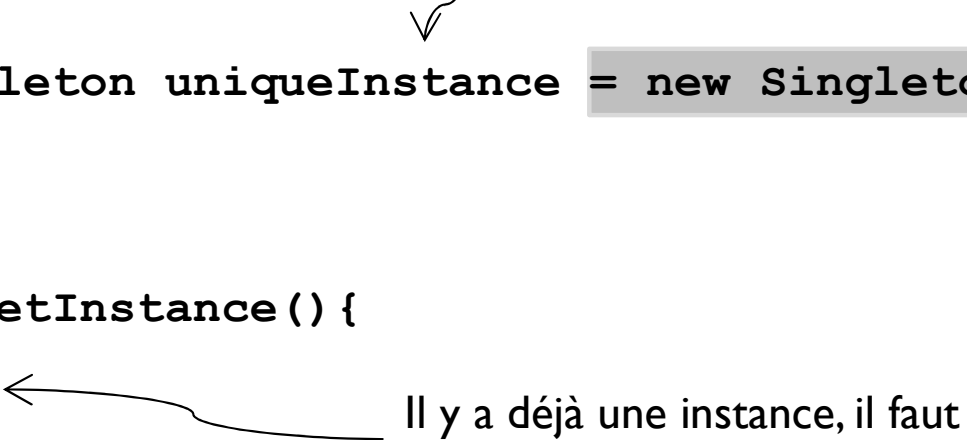
Singleton :

Gestion du multi-threading (12/12)

- Solution 3: création au moment de la définition de la variable de classe

Initialisation par le JVM avant accès des threads

```
public class Singleton {  
    private final static Singleton uniqueInstance = new Singleton();  
  
    private Singleton() {}  
  
    public static Singleton getInstance() {  
        return uniqueInstance;  
    }  
    //autres méthodes utiles  
}
```



- La JVM crée une instance de Singleton lors du chargement de la classe. La JVM garantit que l'instance va être créée avant que les threads accèdent la variable statique **uniqueInstance**.

Récapitulatif (1 / 2)

- ▶ Bases de l'OO: Abstraction, Encapsulation, Polymorphisme & Héritage
- ▶ Principes de l'OO
 - ▶ Encapsuler ce qui varie
 - ▶ Favoriser la composition sur l'héritage
 - ▶ Programmer avec des interfaces et non des implémentations
 - ▶ Opter pour une conception faiblement couplée
 - ▶ Les classes doivent être ouvertes pour les extensions et fermées pour les modifications
 - ▶ Dépendre des abstractions. Ne jamais dépendre de classes concrètes
- ▶ Patron de l'OO
 - ▶ Strategy: définit une famille d'algorithmes interchangeables
 - ▶ Observer: définit une dépendance 1-à-plusieurs entre objets.
 - ▶ decorator: attache des responsabilités additionnelles à un objet dynamiquement.
 - ▶ Abstract Factory: offre une interface de création de familles d'objets
 - ▶ Factory Method: définit une interface de création des objets
 - ▶ **Singleton**: assure à une classe une seule instance et lui offre un point d'accès global

Récapitulatif (2 / 2)

- ▶ Le patron singleton assure la création d'au plus une instance d'une classe de notre application
- ▶ Le patron offre aussi un seul point d'accès global à cette instance
- ▶ L'implémentation Java du patron utilise un constructeur privé une méthode statique combinée avec une variable statique
- ▶ Le développeur examine la performance et les contraintes des ressources et choisit soigneusement une implémentation pour une application multi-thread