

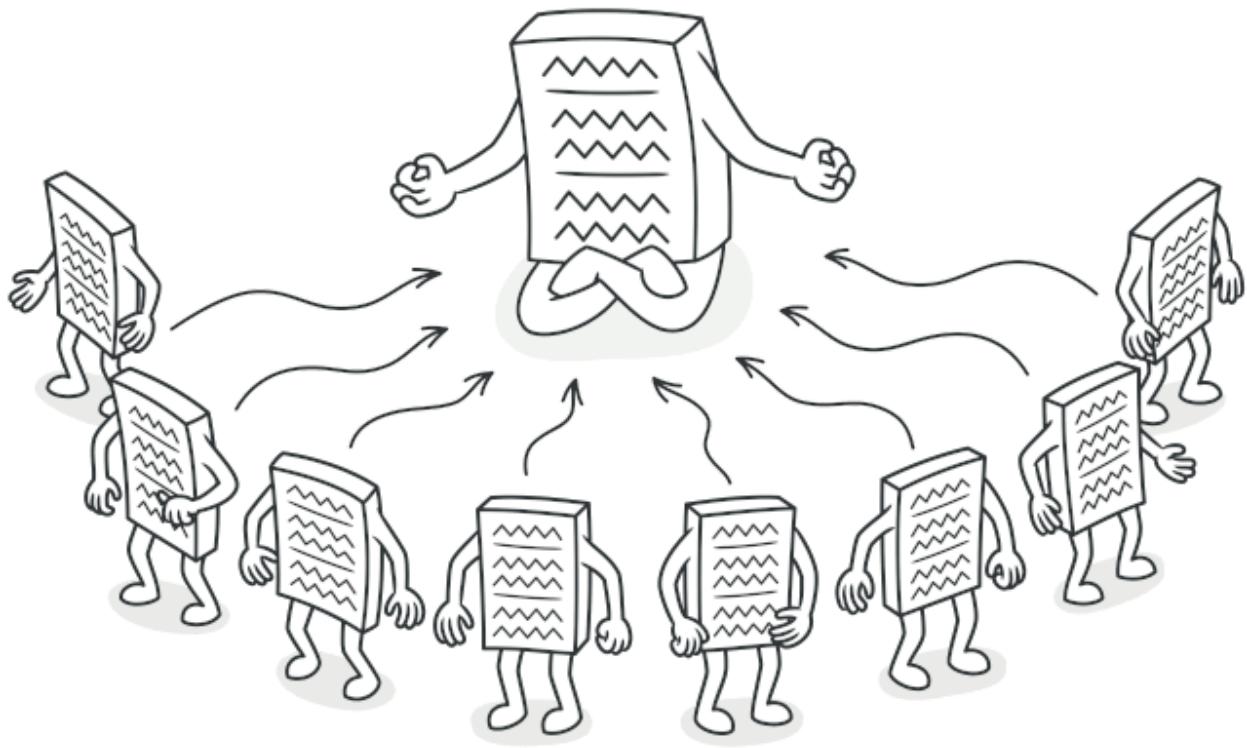


Singleton



Intention :

Singleton est un patron de conception de création qui garantit que l'instance d'une classe n'existe qu'en un seul exemplaire, tout en fournissant un point d'accès global à cette instance.



🤔 Problème :

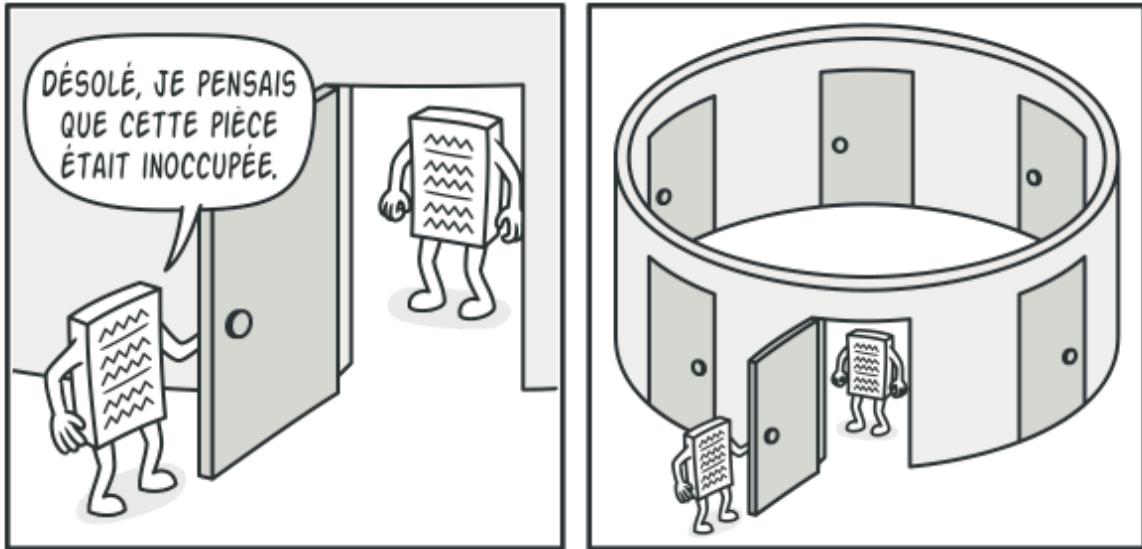
Le singleton règle deux problèmes à la fois, mais ne respecte pas le *principe de responsabilité unique*.

1. **Il garantit l'unicité d'une instance pour une classe.** Pour quelle raison voudrait-on maîtriser le nombre d'instances d'une classe ? En général, cette situation se présente lorsque l'on veut contrôler l'accès à une ressource partagée — une base de données ou un fichier par exemple.

Son fonctionnement est le suivant : vous créez un objet, mais après un certain temps, vous décidez d'en créer un autre. Plutôt que de vous retrouver avec un objet flambant neuf, vous récupérez celui qui existe déjà.

Vous noterez qu'il est impossible d'implémenter ce comportement avec un constructeur normal, puisqu'un constructeur **doit** théoriquement toujours

retourner un nouvel objet.



Les clients ne se rendent pas forcément compte qu'ils travaillent toujours avec le même objet

2. Il fournit un point d'accès global à cette instance. Vous rappelez-vous ces variables globales que vous (bon, d'accord : moi) avez utilisées pour stocker des objets essentiels ? Elles sont très pratiques mais peu fiables, puisque n'importe quelle partie du code peut potentiellement écraser leur contenu et faire planter l'application.

Le singleton vous permet d'accéder à l'objet n'importe où dans le programme, telle une variable globale. Cependant, il protège son instance et l'empêche d'être modifiée.

Un autre aspect majeur vient se glisser dans l'équation : le code qui résout le problème numéro 1 ne doit pas se retrouver éparpillé dans tout le programme. En effet, on préfèrera tout mettre dans une même classe, surtout si le reste du code repose dessus.

Aujourd'hui, le singleton est devenu très populaire et le terme *singleton* est même parfois employé pour une entité ne résolvant qu'un seul des problèmes listés.

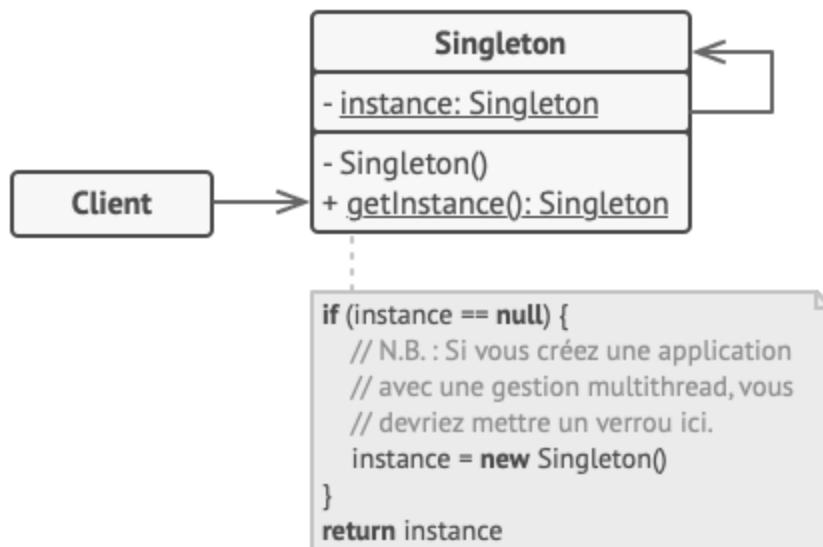
😊 Solution :

Toute mise en place d'un singleton est constituée des deux étapes suivantes :

- Rendre le constructeur par défaut privé afin d'empêcher les autres objets d'utiliser l'opérateur `new` avec la classe du singleton.
- Mettre en place une méthode de création statique qui se comporte comme un constructeur. En coulisse, cette méthode appelle le constructeur privé pour créer un objet et le sauvegarde dans un attribut statique. Tous les appels ultérieurs à cette méthode retournent l'objet en cache.

Si votre code a accès à la classe du singleton, alors il peut appeler sa méthode statique. À chaque appel de cette méthode, c'est toujours le même objet qui est retourné.

Structure



1. La classe **Singleton** déclare la méthode statique `getInstance` qui retourne la même instance de sa propre classe.

Le code client ne doit pas avoir de visibilité sur le constructeur du singleton. Seule la méthode `getInstance` doit permettre l'accès à l'objet du singleton.

Pseudocode :

Dans cet exemple, la classe de la connexion à la base de données est le **Singleton**. Cette classe n'a pas de constructeur public, vous ne pouvez y accéder que grâce à la méthode `getInstance`. Cette méthode met en cache le premier objet créé puis retourne ce même objet lors des appels ultérieurs.

```
// La classe baseDeDonnées définit la méthode `getInstance` qui
// permet aux clients d'accéder à la même instance de la
// connexion à la base de données dans tout le programme.
class Database is
    // L'attribut qui stocke l'instance du singleton doit être
    // 'static'.
    private static field instance: Database

    // Le constructeur du singleton doit toujours être privé
    // afin d'empêcher les appels à l'opérateur `new`.
    private constructor Database() is
        // Code d'initialisation (la connexion au serveur de la
        // base de données par exemple).
        // ...

    // La méthode statique qui contrôle l'accès à l'instance du
    // singleton.
    public static method getInstance() is
        if (Database.instance == null) then
            acquireThreadLock() and then
                // Ce thread attend la levée du verrou (lock) le
                // temps de s'assurer que l'instance n'a pas
                // déjà été initialisée dans un autre thread.
```

```

        if (Database.instance == null) then
            Database.instance = new Database()
        return Database.instance

        // Pour finir, tout singleton doit définir de la logique
        // métier qui peut être exécutée dans sa propre instance.
    public method query(sql) is
        // Par exemple, toutes les requêtes sur la base de
        // données d'une application passent par cette méthode.
        // Par conséquent, vous pouvez définir le code des
        // limitations ou de la mise en cache ici.
        // ...

    class Application is
        method main() is
            Database foo = Database.getInstance()
            foo.query("SELECT ...")
            // ...
            Database bar = Database.getInstance()
            bar.query("SELECT ...")
            // La variable `bar` contiendra le même objet que la
            // variable `foo`.

```

Applicabilité :

 **Utilisez le singleton lorsque l'une de vos classes ne doit fournir qu'une seule instance à tous ses clients. Par exemple, une base de données partagée entre toutes les parties d'un programme.**

 La méthode spéciale de création devient le seul moyen de fabriquer des objets pour la classe, car le singleton désactive les autres. Cette méthode crée un objet ou retourne l'objet existant s'il a déjà été créé.

Utilisez le singleton lorsque vous voulez un contrôle absolu sur vos variables globales.

⚡ Contrairement aux variables globales, le singleton garantit l'unicité de l'instance de la classe. Seule la classe singleton peut remplacer l'instance mise en cache.

Vous pouvez moduler le nombre d'instances du singleton comme vous le voulez. Vous devez juste apporter une modification dans le corps de la méthode `getInstance`.

Comment implémenter :

1. Ajoutez un attribut statique à la classe pour stocker l'instance du singleton.
2. Déclarez une méthode de création publique et statique pour récupérer l'instance du singleton.
3. Implémentez une « instanciation paresseuse » (lazy initialization) à l'intérieur de la méthode statique. Elle devrait créer un nouvel objet lors du premier appel et le stocker dans l'attribut statique. La méthode doit retourner cette instance lors de tous les appels suivants.
4. Rendez privé le constructeur de la classe. La méthode statique de la classe doit être la seule à pouvoir appeler le constructeur.
5. Parcourez le code client et remplacez les appels directs au constructeur du singleton par des appels à la méthode statique.

Avantages et Inconvénients

- ✓ Vous garantissez l'unicité de l'instance de la classe.
- ✓ Vous obtenez un point d'accès global à cette instance.
- ✓ L'objet du singleton est uniquement initialisé la première fois qu'il est appelé

- ✗ Ne respecte pas le *principe de responsabilité unique*. Ce patron résout deux problèmes à la fois.
- ✗ Le singleton peut masquer une mauvaise conception ; il se peut, par exemple, que les composants aient trop de visibilité les uns envers les autres
- ✗ Il doit bénéficier d'un traitement spécial pour fonctionner dans un environnement multithread afin que le singleton ne se retrouve pas en plusieurs exemplaires
- ✗ Les tests unitaires du code client peuvent se révéler difficiles, car de nombreux frameworks reposent sur l'héritage lorsqu'ils créent des objets fictifs. Étant donné que le constructeur de la classe du singleton est privé et que redéfinir la méthode statique est impossible dans la majorité des langages, vous allez devoir être créatif pour reproduire un singleton fictif. Ou ne pas faire de tests. Ou ne pas utiliser de singleton

Relations avec d'autres patterns :

- Une classe **Façade** peut souvent être transformée en **Singleton**, car un seul objet façade est en général suffisant.

- Le **Poids mouche** ressemble au **Singleton** si vous parvenez à compiler tous les états partagés des objets en un seul objet poids mouche. Mais ces patrons de conception ont deux différences fondamentales :
 1. Il ne devrait y avoir qu'une seule instance de singleton, mais une classe *poids mouche* peut avoir plusieurs instances avec différents états intrinsèques.
 2. L'objet *singleton* peut être modifiable alors que les objets poids mouche ne sont pas modifiables.
- Les **Fabriques abstraites**, **Monteurs** et **Prototypes** peuvent tous être implémentés comme des **Singlenton**.