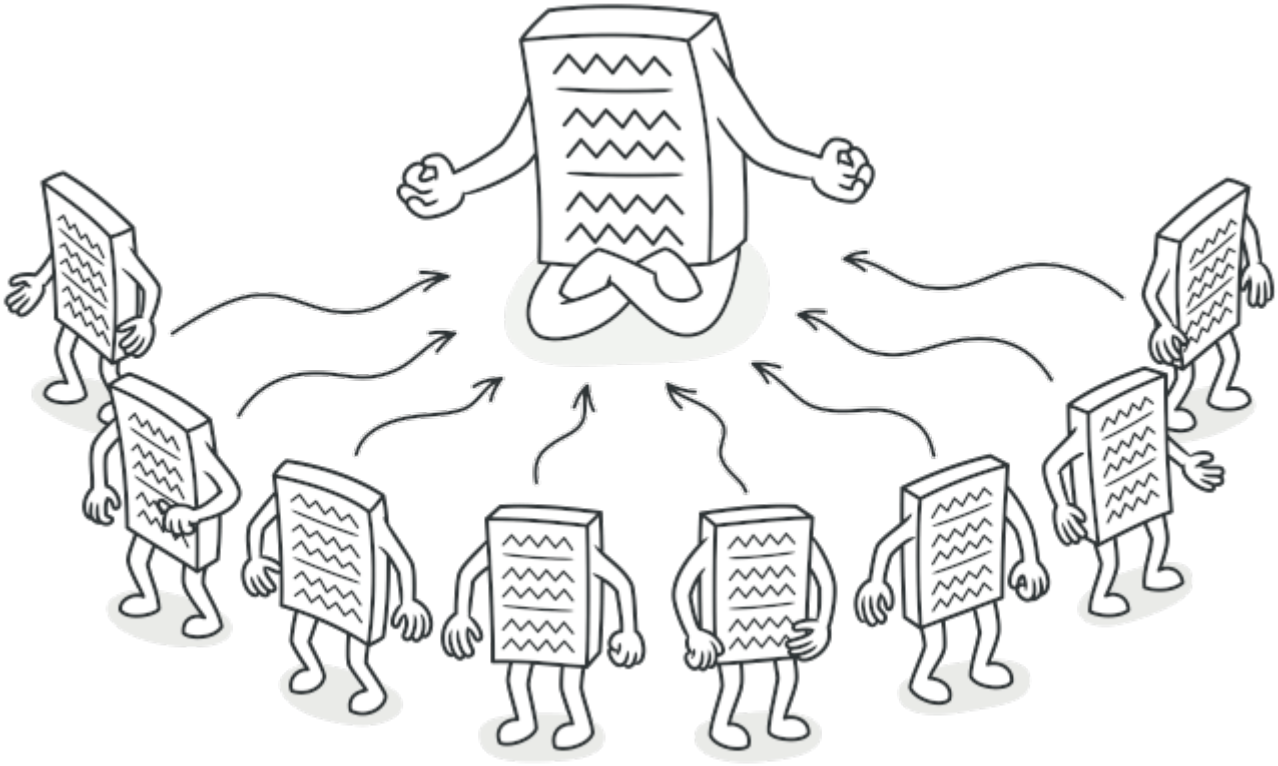
Java Singleton Pattern - Mẫu thiết kế Singleton trong Java

Singleton là một Design Pattern cho phép bạn đảm bảo rằng một lớp chỉ có một thể hiện, và cung cấp một truy xuất toàn cục cho thể hiện này.

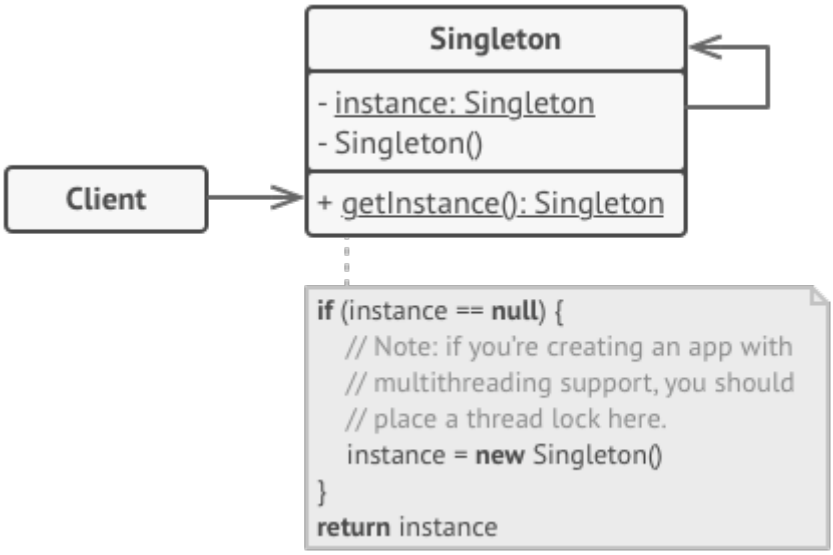


1. Tại sao cần có Singleton Pattern

Đôi khi, trong quá trình phân tích thiết kế một hệ thống, chúng ta mong muốn có những đối tượng cần tồn tại duy nhất và có thể truy xuất mọi lúc mọi nơi. Làm thế nào để hiện thực được một đối tượng như thế khi xây dựng mã nguồn? Chúng ta có thể nghĩ tới việc sử dụng một biến toàn cục (global variable: public static final). Tuy nhiên, việc sử dụng biến toàn cục nó phá vỡ quy tắc của **OOP** (encapsulation). Để giải bài toán trên, người ta hướng đến một giải pháp là sử dụng **Singleton pattern**.

Singleton là 1 trong 5 design pattern của nhóm **Creational Design Pattern**.

Singleton đảm bảo chỉ duy nhất **một thể hiện (instance)** được tạo ra và nó sẽ cung cấp cho bạn một method để có thể truy xuất được thể hiện duy nhất đó mọi lúc mọi nơi trong chương trình.



Sử dụng Singleton khi chúng ta muốn:

- Đảm bảo rằng chỉ có một instance của lớp.
- Việc quản lý việc truy cập tốt hơn vì chỉ có một thể hiện duy nhất.
- Có thể quản lý số lượng thể hiện của một lớp trong giới hạn chỉ định.

Một số trường hợp sử dụng của Singleton Pattern thường gặp:

- Vì class dùng Singleton chỉ tồn tại 1 Instance (thể hiện) nên nó thường được dùng cho các trường hợp giải quyết các bài toán cần truy cập vào các ứng dụng như: Database, Shared resource, Logger, Configuration, Caching, Thread pool…
- Một số design pattern khác cũng sử dụng Singleton để triển khai: Abstract Factory, Builder, Prototype, Facade,…
- Đã được sử dụng trong một số class của core java như: java.lang.Runtime, java.awt.Desktop.

2. Thực thi Singleton Pattern

Có rất nhiều cách để implement Singleton Pattern bằng các dựa trên nguyên tắc dưới đây cơ bản dưới đây:

- Khai báo **private constructor** để hạn chế truy cập từ class bên ngoài.
- Khai báo **private static final variable** đảm bảo biến chỉ được khởi tạo trong class.
- Khai báo một method **public static** để **return instance** được khởi tạo ở trên.

2.1. Eager initialization

Singleton Class được khởi tạo ngay khi được gọi đến. Đây là cách dễ nhất nhưng nó có một nhược điểm mặc dù instance đã được khởi tạo mà có thể sẽ không dùng tới.

```
public class SingleObject {

   //create an object of SingleObject
   private static SingleObject instance = new SingleObject();

   //make the constructor private so that this class cannot be
   //instantiated
   private SingleObject(){}

   //Get the only object available
   public static SingleObject getInstance(){
      return instance;
   }

   public void showMessage(){
      System.out.println("Hello World!");
   }
}
```

```
public class SingletonPatternDemo {

   public static void main(String[] args) {

      //illegal construct
      //Compile Time Error: The constructor SingleObject() is not visible
      //SingleObject object = new SingleObject();

      //Get the only object available
      SingleObject object = SingleObject.getInstance();

      //show the message
      object.showMessage();
   }
}
```

2.2. Static block initialization

Vì `static block` sẽ chỉ được gọi một lần, chúng ta có thể sử dụng `static block` để phát triển lớp singleton. Dưới đây ví dụ cho thấy cách tạo các lớp singleton bằng cách sử dụng `static block`:

- Để tạo lớp singleton, hãy tạo constructor thành private, để bạn không thể tạo đối tượng bên ngoài lớp.
- Tạo một biến private static có cùng loại class, để đối tượng được tạo sẽ được trỏ đến tham chiếu này. Bây giờ tạo `static block` và tạo đối tượng bên trong `static block`. Vì `static block` sẽ chỉ được gọi một lần, nên đối tượng sẽ chỉ được tạo một lần.

```
public class MyStaticSingleton {

    public static void main(String a[]){
        MySingleton ms = MySingleton.getInstance();
        ms.testSingleton();
    }
}

class MySingleton{

    private static MySingleton instance;

    static{
        instance = new MySingleton();
    }

    private MySingleton(){
        System.out.println("Creating MySingleton object...");
    }

    public static MySingleton getInstance(){
        return instance;
    }

    public void testSingleton(){
        System.out.println("Hey.... Instance got created...");
    }
}
```

Output:

```
Creating MySingleton object...
Hey.... Instance got created...
```

2.3. Lazy Initialization

Là một cách làm mang tính mở rộng hơn so với 2 cách làm trên và hoạt động tốt trong môi trường đơn luồng (single-thread).

```
public class LazyInitializedSingleton {

    private static LazyInitializedSingleton instance;

    private LazyInitializedSingleton() {
    }

    public static LazyInitializedSingleton getInstance() {
        if (instance == null) {
            instance = new LazyInitializedSingleton();
        }
        return instance;
    }
}
```

- Ưu điểm: Cách này đã khắc phục được nhược điểm của cách **Eager initialization**, chỉ khi nào **getInstance()** được gọi thì instance mới được khởi tạo.

- Nhược điểm:

- Tuy nhiên, cách này chỉ sử dụng tốt trong trường hợp đơn luồng (single-thread), trường hợp nếu có nhiều luồng (multi-thread) cùng chạy và cùng gọi hàm getInstance() tại cùng một thời điểm thì có thể có nhiều hơn 1 thể hiện của instance. Để khắc phục nhược điểm này chúng ta sử dụng **Thread Safe Singleton**.
- Một nhược điểm nữa của **Lazy Initialization** cần quan tâm là: đối với thao tác create instance quá chậm thì người dùng có phải chờ lâu cho lần sử dụng đầu tiên.

2.4. Thread Safe Singleton

Để đảm bảo khi khởi tạo Singletong trong Thread, chúng ta có thể sử dụng phương thức **synchronized** trong hàm **getInstance()** và như vậy hệ thống đảm bảo rằng tại cùng một thời điểm chỉ có thể có 1 luồng có thể truy cập vào hàm getInstance() và đảm bảo rằng chỉ có duy nhất 1 thể hiện của class.

```
// Java program to create Thread Safe
// Singleton class
public class GFG {
    // private instance, so that it can be
    // accessed by only by getInstance() method
    private static GFG instance;
```

```
    private GFG() {
        // private constructor
    }

    //synchronized method to control simultaneous access
    synchronized public static GFG getInstance() {
        if (instance == null) {
            // if instance is null, initialize
            instance = new GFG();
        }
        return instance;
    }
}
```

> Biến `volatile` trong Java có tác dụng thông báo sự thay đổi giá trị của biến tới các thread khác nhau nếu biến này đang được sử dụng trong nhiều thread.

Cách này có nhược điểm là một phương thức **synchronized** sẽ chạy rất chậm và tốn hiệu năng, bất kỳ Thread nào gọi đến đều phải chờ nếu có một Thread khác đang sử dụng. Có những tác vụ xử lý trước và sau khi tạo thể hiện không cần thiết phải block. Vì vậy chúng ta cần cải tiến nó đi 1 chút với **Lazy initialization with Double check locking**.

2.5. Lazy initialization with Double check locking

Để implement theo cách này, chúng ta sẽ kiểm tra sự tồn tại thể hiện của lớp, với sự hỗ trợ của đồng bộ hóa, hai lần trước khi khởi tạo. Phải khai báo **volatile** cho instance để tránh lớp làm việc không chính xác do quá trình tối ưu hóa của trình biên dịch.

```
// Java code to explain double check locking
public class GFG {
    // private instance, so that it can be
    // accessed by only by getInstance() method
    private static volatile GFG instance;

    private GFG() {
        // private constructor
    }

    public static GFG getInstance() {
```

```
        if (instance == null) {
            //synchronized block to remove overhead
            synchronized(GFG.class) {
                if (instance == null) {
                    // if instance is null, initialize
                    instance = new GFG();
                }
            }
        }
        return instance;
    }
}
```

2.6. Bill Pugh Singleton Implementation

Trước Java5, mô hình bộ nhớ có rất nhiều vấn đề và các phương thức trên gây ra lỗi trong các kịch bản nhất định trong môi trường đa luồng. Vì vậy, Bill Pugh đã đề xuất một khái niệm về các lớp inner static để sử dụng cho singleton.

```
// Java code for Bill Pugh Singleton Implementaion
public class GFG {

    private GFG() {
        // private constructor
    }

    // Inner class to provide instance of class
    private static class BillPughSingleton {
        private static final GFG INSTANCE = new GFG();
    }

    public static GFG getInstance() {
        return BillPughSingleton.INSTANCE;
    }
}
```

Khi lớp singleton được load, lớp bên trong không được load và do đó không tạo đối tượng khi load class. Inner class chỉ được tạo khi phương thức getInstance() được gọi. Vì vậy, nó có vẻ giống như eager initialization nhưng đó là lazy initialization.

Đây là cách tiếp cận được sử dụng rộng rãi nhất vì nó không sử dụng đồng bộ hóa.

2.7. Enum Singleton

Khi dùng **enum** thì các params chỉ được khởi tạo 1 lần duy nhất, đây cũng là cách giúp bạn tạo ra Singleton instance.

```
/**
 * Singleton implementation using enum initialization
 */
public enum EnumSingleton {

    INSTANCE;
}
```

Lưu ý:

- Enum có thể sử dụng như một Singleton, nhưng nó có nhược điểm là không thể extends từ một lớp được, nên khi sử dụng cần xem xét vấn đề này.
- Hàm **constructor** của **enum** là **lazy**, nghĩa là khi được sử dụng mới chạy hàm khởi tạo và nó chỉ chạy duy nhất một lần. Nếu muốn sử dụng như một eager singleton thì cần gọi thực thi trong một **static block** khi start chương trình.
- So sánh giữa 2 cách sử dụng **enum initialization** và **static block initialization method**, enum có một điểm rất mạnh khi giải quyết về vấn đề **Serialization/ Deserialization**.

3. Tổng kết

1. `Eager initialization` là cách dễ dàng nhất để thực thi Singleton nhưng nó có nhược điểm khi instance được khởi tạo mà có thể không được sử dụng.
2. Sử dụng `Static block` trong Eager initialization, chúng ta có thể cung cấp xử lý ngoại lệ và kiểm soát được instance.
3. Sử dụng `synchronized` chúng ta có thể tạo Singleton class trong môi trường đa luồng multi-threading nhưng nó có thể gây chậm chương trình, một cách tối ưu hơn là sử dụng cơ chế Double check locking.
4. `Bill Pugh` là phương pháp được sử dụng rộng rãi nhất trong thực thi Singleton do việc khai báo đơn giản và có nhiều ưu điểm.

Revision #3
Created 2 November 2019 13:53:01 by Laptrinh.vn
Updated 12 April 2020 14:52:08 by Laptrinh.vn