Singleton Design Pattern

What is Singleton design pattern?

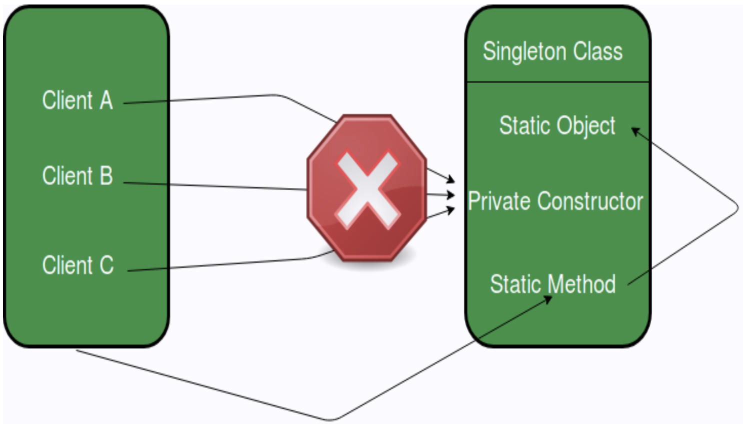
The ***Singleton Design Pattern*** ensures that a class has ***only one instance*** throughout the application. If an attempt is made to create another object, it returns the ***same existing instance*** instead of creating a new one.

Let's implement the ***Singleton Design Pattern*** in ***Java***. As we know, Java uses ***two*** primary ***memory*** areas:

1. The ***stack***, which stores ***primitive datatypes*** and ***references***.
2. The **heap**, which stores ***actual objects*** (**non-primitive data**).

When we create an object using a constructor, the ***object*** itself is allocated in the ***heap***, while its ***reference*** is stored on the ***stack***.

To implement the **Singleton pattern**, we need to design a class in such a way that it ***creates only one instance*** of itself in the ***heap***. Any subsequent request for an object should return the ***same existing instance*** instead of creating a new one.



It is two types:

1. Eager initialization:

```
// sample code
public class Singleton {
  private static final Singleton obj = new Singleton ();

  // Constructor
  private Singleton () {
    System.out.println("<>--Creating-an-Object--<>");
  }

  // getter
  public static Singleton getObj () {
    return obj;
  }
}
```

------------------------Client-Code------------------------

```
public class Main {
  public static void main(String[] args) {
    Singleton obj = Singleton.getObj();
    Singleton obj1 = Singleton.getObj();

    System.out.println(obj == obj1);
  }
}
```

-------------------Client-Code-Output-------------------

```
<>--Creating-an-Object--<>
true
```

✅ ***Advantages***:

- ***Simplicity***: Very easy to implement.
- ***Thread-safe***: Since the instance is created at class loading time, no need for synchronization.
- ***Performance***: No locking or conditional checks on every access.

❌ ***Disadvantages***:

- ***Memory overhead***: Instance is created even if it's never used.
- ***Lacks flexibility***: Cannot handle exceptions during object creation.
- ***Not suitable for heavy objects***: If the object is large or expensive to create, it wastes resources if unused.

@Sibasundarj8

2. Lazy initialization:

```
// Sample Code
public class Singleton {
    // Changes made by one thread to obj are immediately
    // visible to other threads.
    private static volatile Singleton obj;

    // Constructor
    private Singleton () {
        System.out.println("<>--Creating-an-Object--<>");
    }

    // getter
    public static Singleton getObj () {
        // 🔐 Lock is a very expensive operation
        // so we check nullability before locking
            if (obj == null) {
                synchronized (Singleton.class){
                    // double check if multiple thread
                    // enter simultaneously
                    if (obj == null){
                        obj = new Singleton ();
                    }
                }
            }
        return obj;
    }
}
```

------------------------Client-Code-----------------------

```
public class Main {
    public static void main(String[] args) {
        Singleton obj = Singleton.getObj();
        Singleton obj1 = Singleton.getObj();

        System.out.println(obj == obj1);
    }
}
```

-------------------Client-Code-Output------------------

```
<>--Creating-an-Object--<>
true
```

✅ ***Advantages***:

- ***Resource-efficient***: Instance is created only when needed.
- ***Suitable for heavy objects***: Better for objects that are rarely used or expensive to create.

❌ ***Disadvantages***:

- ***More complex***: Implementation becomes more complicated with thread-safety mechanisms.
- ***Slightly slower***: First-time access may be slower due to checks or synchronization.

Singleton Design:

1. Create a private constructor.
2. Create a private static instance and getter which create once and returns same instance every time.

Practical use cases:

1. 🪵 Logger system
2. 🗂️ Configuration Manager
3. 🧵 Thread Pool (ExecutorService)
4. 🫘 Spring Beans

@Sibasundarj8