Lab Report

On

Singleton Design Pattern

(Thread Safe Methods)

Course Code: CSE - 418

Course Title: Software Engineering & Design Pattern Lab

Submitted to:

Plabon Talukder

Lecturer

Department of Computer Science & Engineering

Metropolitan University, Bangladesh

Submitted by:

Mahmud

ID: 222-115-191

Batch: 57th (E)

Department of Computer Science & Engineering

Metropolitan University, Bangladesh

Date of Submission: Mar 16, 2025

Thread Safety in Singleton

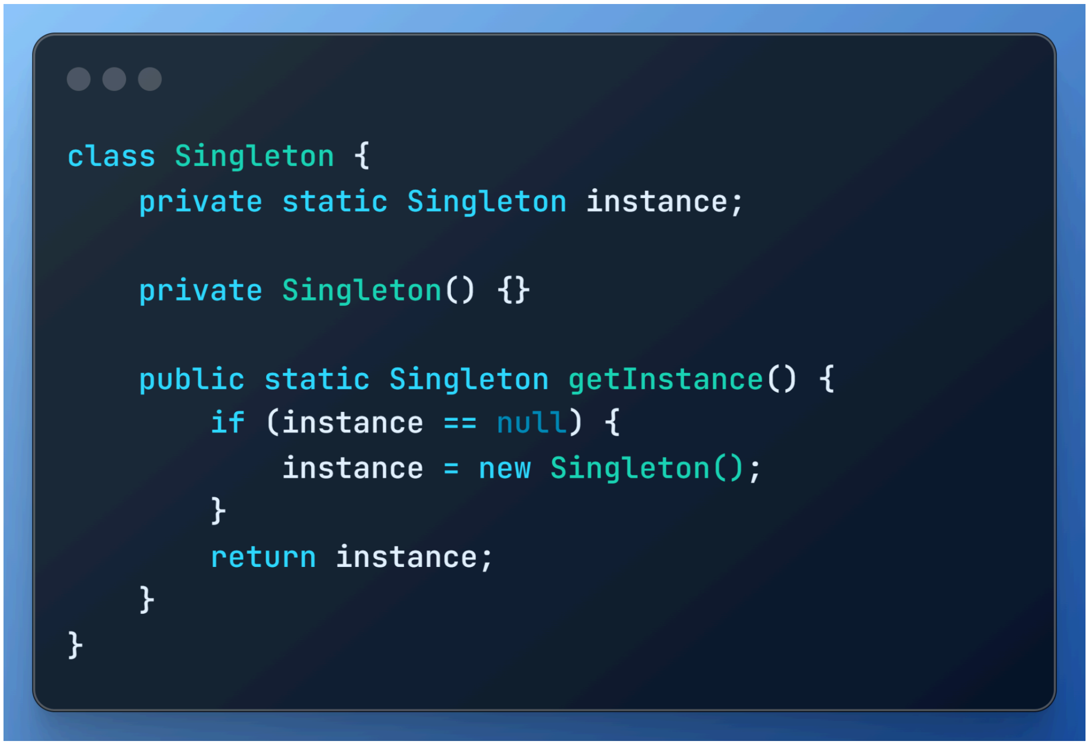
Thread safety in Singleton refers to ensuring that multiple threads do not create multiple instances when accessing the Singleton class concurrently. Without thread safety, multiple threads might create multiple instances of the Singleton, violating its purpose.

Implementing Thread-Safe Singleton in Java

There are several ways to make a Singleton thread-safe:

1. Lazy Initialization

Lazy Initialization also known as Basic Singleton or Non-Thread Safe Method. This is the simplest but not thread-safe because multiple threads can create separate instances.



```
class Singleton {
    private static Singleton instance;

    private Singleton() {}

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

Pros:

- Simple and easy to implement
- Works well in single-threaded applications

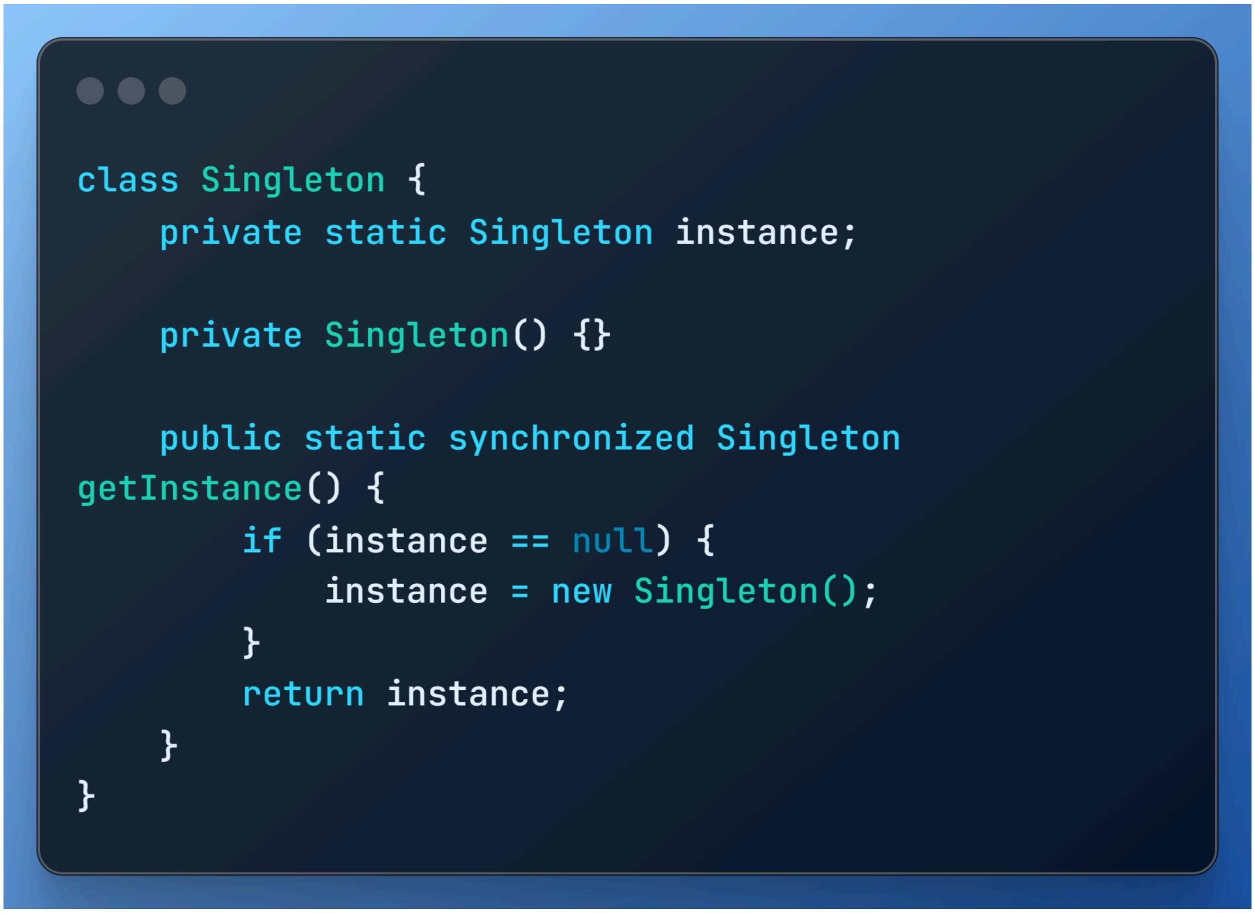
Cons:

- **Not thread-safe** → Multiple threads may create separate instances
- Cannot be used in concurrent applications

Conclusion: Suitable only for non-multithreaded environments.

2. Synchronized Method

Adding `synchronized` ensures only one thread at a time can access `getInstance()`, but it slows down performance. It is thread safe, but slow.



```
class Singleton {
    private static Singleton instance;

    private Singleton() {}

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

Pros:

- Ensures **thread safety**
- Straightforward implementation

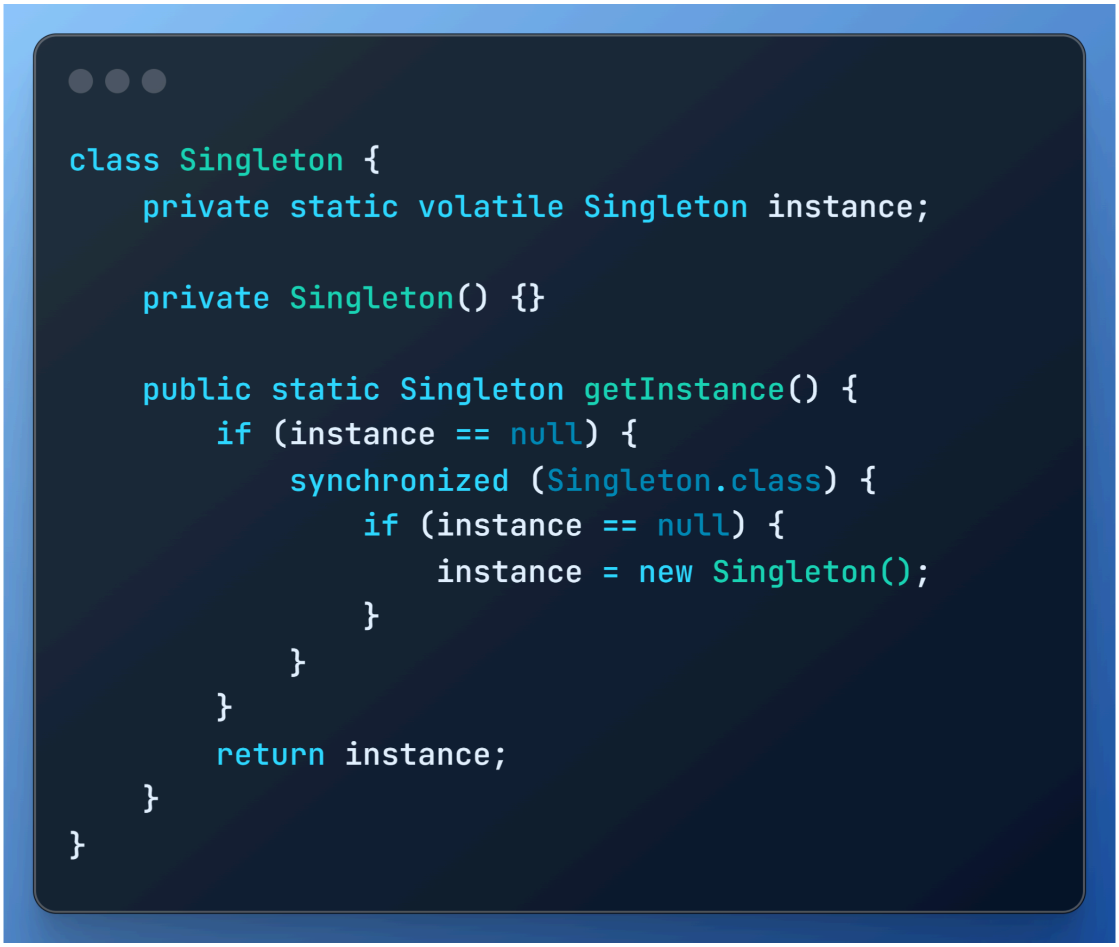
Cons:

- **Performance overhead** → Synchronization slows down every call to `getInstance()`

Conclusion: Safe but inefficient when performance is a concern.

3. Double-Checked Locking

This reduces synchronization overhead while ensuring thread safety. It is a balanced approach.



```java
class Singleton {
    private static volatile Singleton instance;

    private Singleton() {}

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

Key Points:

- `volatile` ensures that multiple threads handle `instance` correctly.
- The **first check** avoids unnecessary synchronization once the instance is created.
- The **second check** ensures that no two threads create separate instances.

Pros:

- **Thread-safe** without excessive synchronization overhead
- **Performance-efficient** for concurrent applications

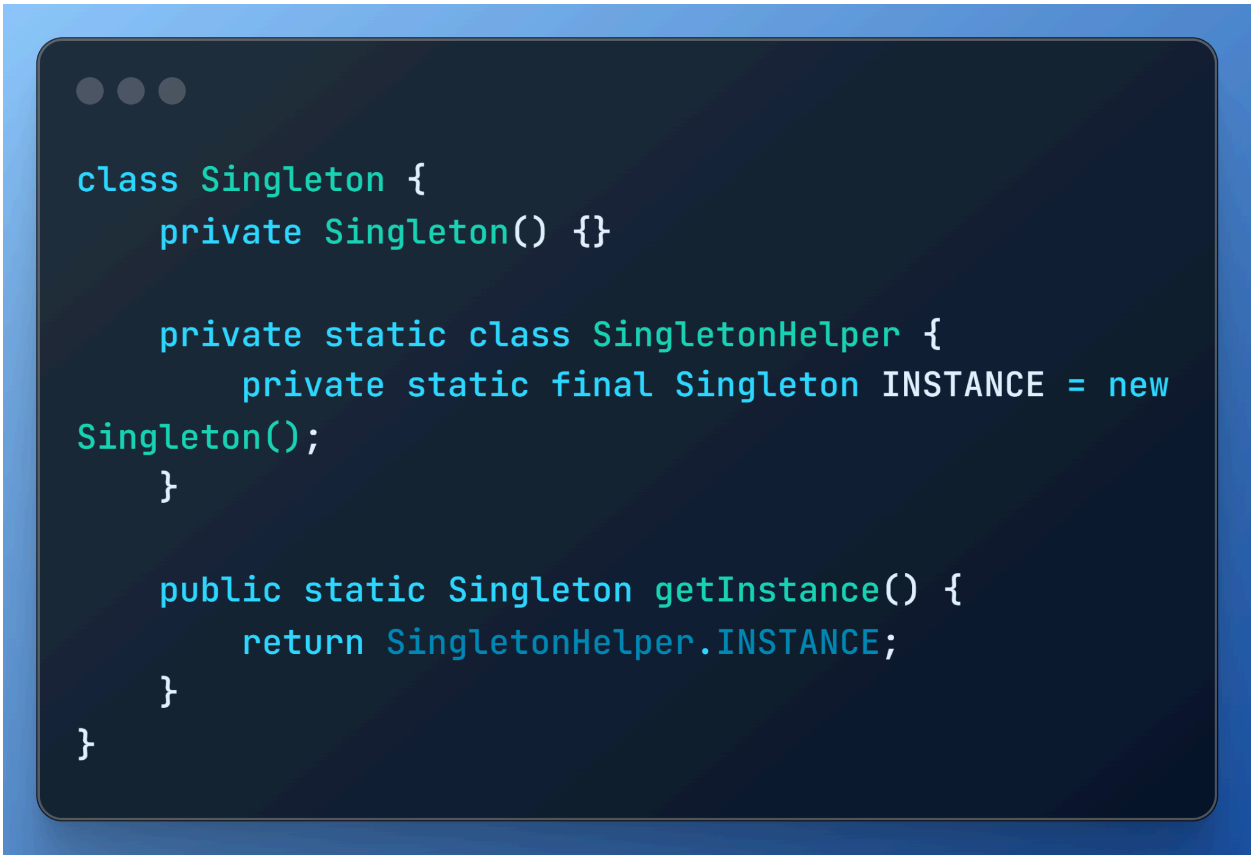
Cons:

- Requires `volatile`, which was unreliable in Java versions before 1.5

Conclusion: A solid choice for multithreading while maintaining good performance.

4. Bill Pugh Singleton

Uses an inner static helper class, which is thread-safe and lazy-loaded. It is the most efficient approach.



```java
class Singleton {
    private Singleton() {}

    private static class SingletonHelper {
        private static final Singleton INSTANCE = new
Singleton();
    }

    public static Singleton getInstance() {
        return SingletonHelper.INSTANCE;
    }
}
```

Keys Points:

- The **inner static class** ensures that the instance is created only when **`getInstance()`** is called.
- This approach is **lazy-loaded and thread-safe** without synchronization overhead.

Pros:

- **Thread-safe** without explicit synchronization
- **Lazy initialization** without performance cost
- **Fastest implementation** due to JVM class loading

Cons:

- Slightly **less intuitive** than other approaches

Conclusion: The most efficient and recommended method for production use.

Comparison Table

Method	Thread-Safe?	Performance	Complexity	Best Use Case
Lazy Initialization	No	Fast	Simple	Single-threaded applications
Synchronized Method	Yes	Slow	Simple	Low-concurrency applications
Double-Checked Locking	Yes	Fast	Moderate	High-performance applications
Bill Pugh Singleton	Yes	Fatest	Slightly Complex	Best for production

Which to Choose

- If you want a **simple** approach and performance is not an issue, go with the **Synchronized Method**.
- If you need a balance between safety and speed, **Double-Checked Locking** works well.
- If you're looking for the most efficient and production-ready approach, **Bill Pugh Singleton** is the best choice.