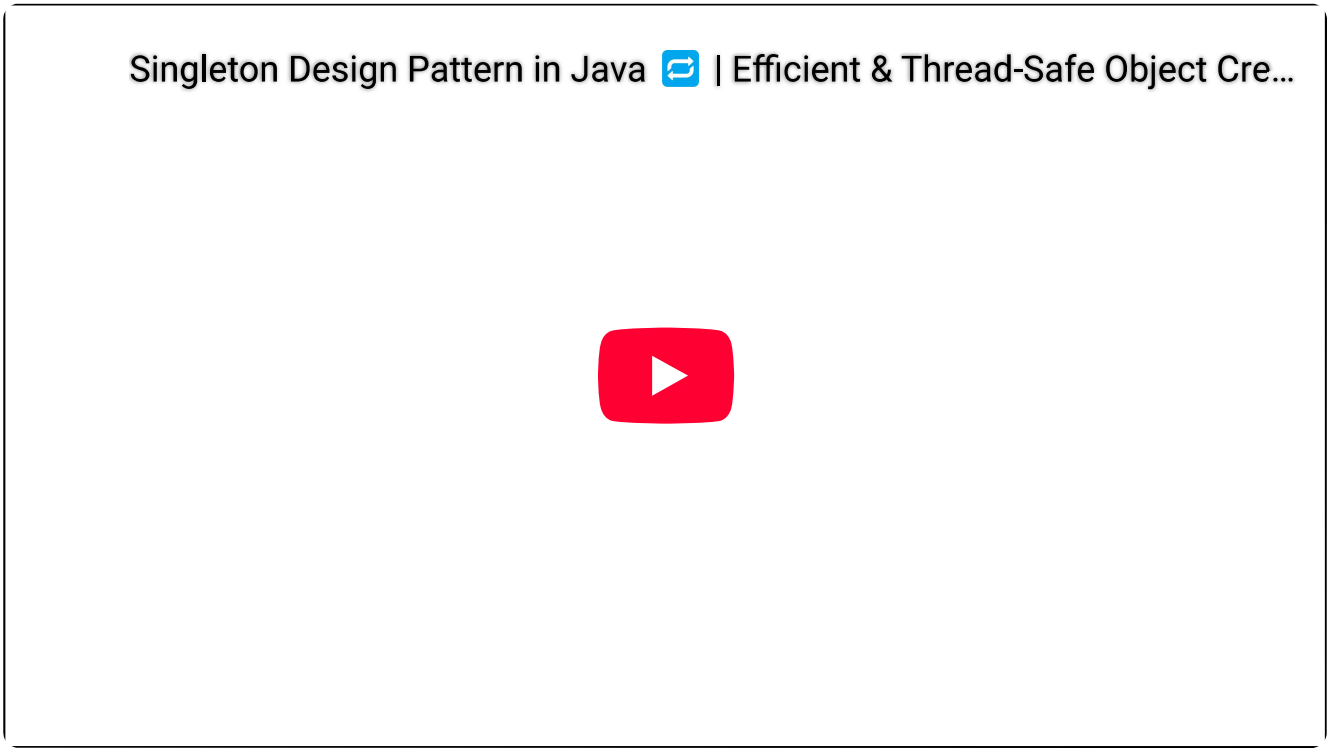
Search articles...

Singleton Design Pattern



Topic Tags:

LLD System Design

Singleton Design Pattern: Ensuring Only One Instance 🔒

The Problem We Need to Solve 🧐

Let's imagine you're building a logging system for a large application. The goal is to have one and only one instance of the logger throughout the entire application. This means:

• No matter how many classes or threads use the logger, they all refer to the same object. 🔁
• This ensures that there are no multiple loggers created, which would waste resources. 💸

The log refers to the messages that the Logger object writes, helping track and monitor system events, user actions, and errors in a consistent manner across your entire application. 📊

For example, a Logger might log:
• A successful login: "User 'john_doe' logged in successfully" ✅
• A failed login: `"ERROR: Invalid login attempt for user 'john_doe'" ❌
• An exception: "ERROR: NullPointerException at line 42 in UserService.java" ⚠️

Now, you might think, "Why not just create a new logger every time I need it?" 🤷 Well, here's the issue: creating multiple instances of the logger could cause issues with memory usage, or even worse, inconsistent logging if multiple loggers are trying to write to the log at the same time. 😱

By having a single instance of the Logger, you ensure that all parts of your application write these log entries to the same location (e.g., a file, database, or console), making it easier to monitor and debug the system. 🔍

This is where the Singleton Design Pattern comes in! 🎉 It allows you to create only one instance of a class and ensures that all parts of your application use that same instance. One logger instance ensures that all logs go to the same place and are written in the same format, making your logs more useful and easier to manage. 😊

Solving the Problem with the Traditional Approach (Not the Best Way) 🔧
So, you start by creating a simple Logger class. The idea is that the Logger will handle writing messages to the console or a log file.

Here's the Logger class:

Java

```java
public class Logger {
  public void log(String message) {
    System.out.println("Log: " + message);
  }
}
```

It looks simple enough. Now, you need to use this logger in your application to keep track of important events. So, you go ahead and create a new Logger instance every time you need it.

For example, in the Application class, you create a new instance of the Logger and use it to log a message:

Java

```java
public class Application {
    public void run() {
        Logger logger = new Logger();  // New instance created every time
        logger.log("Application started.");
    }
}
```

The Problem: Multiple Instances of the Logger

This approach seems to work fine at first. However, let's stop and think for a second: What's happening here every time we call the run() method?

Each time the run() method is executed, a new Logger instance is created. This means that the application is constantly creating new instances of the logger, even though all these loggers are supposed to do the same job: log messages.

Now, imagine that you have several classes in your application that need to log messages. For example, you might also have a UserService class that handles user actions like logging in:

Java

```java
public class UserService {
    public void login(String username) {
        Logger logger = new Logger();  // Another new instance created
        logger.log("User " + username + " logged in.");
    }
}
```

In the UserService class, you're creating a new logger instance every time a user logs in. So, now you have two loggers running in your application — one in Application and one in UserService.

Interviewer's Question: Can We Improve This? 🤔

An interviewer might ask:
• What if you want to make sure only one instance of Logger exists across the entire application? 🔑
• How can we avoid creating multiple instances of Logger? 🚫
• Is this the most efficient way to handle the logging system? ⚡

As we can see, this code is creating a new Logger object every time, which is inefficient. We need to ensure that only one instance of the logger exists, no matter how many times we reference it. 🛠️

The Problem with the Traditional Approach: Messy and Inefficient 😓

The issue with this approach is that every part of the application (like Application, UserService, etc.) creates a new instance of the Logger class when they need to log something. This creates several problems:

1. **Multiple Instances of Logger:**
   ○ If different parts of the system are creating multiple instances of the Logger, it leads to inefficient resource usage. If you are logging to a file, for example, each logger might try to access and write to the file at the same time, leading to potential conflicts or overhead. 📝⚠️

2. **Inconsistent Logging:**
   ○ With multiple loggers, you might end up with log messages spread across different log files or inconsistent output in the same log file, as each instance of the Logger might manage its own logging output. This makes debugging and monitoring harder. 🔍📉

3. **Difficulty Managing State:**
   ○ If the logger has state-related data (e.g., which log file it writes to, configuration settings, etc.), creating multiple instances means that each logger could have a different state. This would cause inconsistency in how logs are managed and stored. ⚖️💼

If we wanted to make sure there was only one logger, we would have to check for the existence of an existing instance of the Logger each time we want to use it, and that would make the code ugly and complex. You'd end up with a lot of extra code to track and manage the instance. 🤦‍♂️🛠️

For example:

Java

```java
public class Logger {
  private static Logger logger = null;
  private Logger() {} // Private constructor to prevent external instantia
  public static Logger getLogger() {
    if (logger == null) {
      logger = new Logger(); // New instance only if one doesn't exist
    }
    return logger;
  }
```

```
10    public void log(String message) {
11      System.out.println("Log: " + message);
12    }
13  }
14  public class Application {
15    public void run() {
16      Logger logger = Logger.getLogger(); // Always fetch the same instance
17      logger.log("Application started.");
18    }
19  }
```

Enter Our Savior: The Singleton Design Pattern 😎

Now, we introduce our savior: the Singleton Design Pattern. The Singleton pattern ensures that a class has only one instance and provides a global point of access to that instance. 🌍

The Singleton Design Pattern is called "Singleton" because it ensures that a class has only one instance throughout the entire system, and it provides a global point of access to that instance.
The word "single" in "Singleton" refers to the fact that the class will have only one instance, no matter how many times it's accessed or instantiated. The pattern guarantees that there is only one object of that class at any given time. 🔑

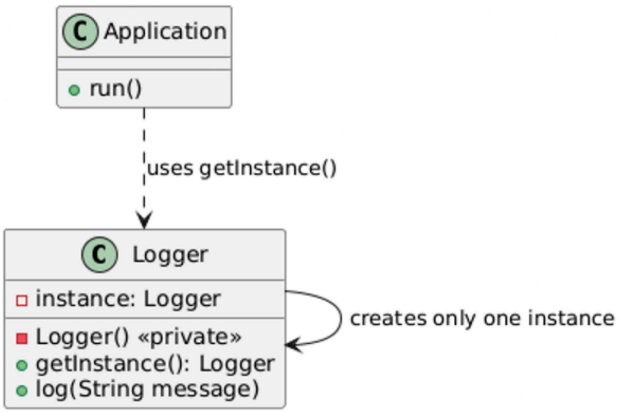
To put it simply, just like how a singleton (a person) is unique and exists only once in a specific context, the Singleton pattern ensures that only one object of a certain class is created, and it's used across the whole application. 🎯

In short:
• "Single" = Only one instance. 1️⃣
• "Ton" = Ensures that instance is accessible globally. 🌐
It's called Singleton because it focuses on creating a single, unique instance that is shared across the entire application, making it efficient and manageable. 😊

Here's how the Singleton works:
1. We make the constructor private, so no one can directly instantiate the class. 🚫
2. We create a static instance of the class inside the class itself. 📦
We provide a public static method (getInstance()) to return the single instance of the class. 🔄

Java

```java
public class Logger {
  // 1. Private static variable to hold the single instance
  private static Logger instance;
  // 2. Private constructor to prevent instantiation
  private Logger() {}
  // 3. Public method to provide access to the instance
  public static Logger getInstance() {
    if (instance == null) {
      instance = new Logger(); // Create a new instance only if it doesn't
    }
    return instance; // Return the existing instance
  }
  public void log(String message) {
    System.out.println("Log: " + message);
  }
}
public class Application {
  public void run() {
    // 4. Fetch the single instance of the Logger
    Logger logger = Logger.getInstance();
    logger.log("Application started.");
  }
}
```



Solving the Follow-up Questions Using Singleton 😄

Now that we've applied the Singleton pattern, let's see how we can address the interviewer's follow-up questions:

• What if we want only one Logger instance? 🔑
○ With the Singleton pattern, there will always be only one instance of the Logger class, no matter how many times you call getInstance(). 🌟

• How can we avoid creating multiple instances of Logger? 🚫
○ The Singleton ensures that only one instance is created, and subsequent calls to getInstance() return the same instance. 🔁

• Is this the most efficient way to handle logging? ⚡
○ Yes, this is a very efficient way because it ensures we're not repeatedly creating new instances, which would waste memory and resources. 🧠

Real-life Use Cases of Singleton Pattern 🌍

The Singleton Design Pattern is used in various real-life situations where we want to ensure only one instance of an object. Here are some examples:

• Logging Systems:
As we've already seen, logging systems often use Singleton to ensure that there's only one logger instance, ensuring consistent logging throughout an application. 📜

• Database Connections:
We often need a single database connection throughout the application to avoid multiple connections that could lead to inefficiency or resource exhaustion. 💻

• Configuration Settings:
Imagine having configuration settings for your application that need to be consistent across the app. Using a Singleton pattern ensures that only one instance of the settings object exists. ⚙️

• Thread Pooling:
A thread pool manager can also use a Singleton to ensure that the pool is managed efficiently with only one thread pool instance. 🎷

Usage of Singleton in Multithreading 🎷

Let's take a step back and imagine you're working on an application that has multiple parts, each running on different threads (like a multi-tasking kitchen with different chefs preparing different dishes at the same time). 👨‍🍳👩‍🍳 Now, let's say one of those parts needs to access a Logger to write some logs. You've already applied the Singleton Design Pattern to ensure that only one instance of the Logger class exists, which is great! 👍

But here's where the multithreading magic happens: Since multiple parts of the application might be running at the same time, multiple threads might try to access and create the Singleton instance of the Logger simultaneously. 😬
What happens then? 🤯

Problem in Multithreading: The Chaos of Multiple Instances 🤯

Imagine this:
• Thread A checks if the Logger instance is null (it is, because no instance has been created yet). 🔍
• Thread B does the same thing at the same time, not knowing that Thread A is also trying to create the Logger instance. 🚨

Both threads decide to create a new instance of the Logger, and suddenly you have two instances of the Logger, which totally breaks the Singleton pattern! 😱

This problem is especially common in multithreaded environments where multiple parts of the program are running simultaneously, trying to access shared resources. 🧩

Why is this a problem?

• Multiple instances: Now you have more than one Logger when you only wanted one, leading to inefficiency and possible issues with logging output (e.g., logs could be written to different places, causing confusion). 🔄 📉
• Race conditions: This also introduces race conditions, where the threads are competing to create the instance, leading to unpredictable behavior. ⚠️

Solution: Making Singleton Thread-Safe 🛡️

Now, we need to fix this problem so that no matter how many threads try to access the Logger at the same time, only one instance gets created. 🔑

The solution is to make the Singleton instance creation process thread-safe. Here's how we can do it:
1. Using Synchronized Blocks 🛠️
We can use synchronization to ensure that only one thread can create the Logger instance at a time. In Java, the synchronized keyword is used to control access to critical sections of code, making sure that only one thread can execute a block of code at any given time. 🕐

Here's how we can apply synchronization:

Java

```
public class Logger {
  private static volatile Logger
      instance; // volatile keyword ensures visibility across threads
  private Logger() {} // Private constructor to prevent instantiation
  public static Logger getInstance() {
    if (instance == null) { // First check (no synchronization needed here
      synchronized (
          Logger.class) { // Synchronize only when creating the instance
        if (instance == null) { // Second check (inside synchronized block
          instance = new Logger(); // Create the instance if it's still nu
        }
      }
    }
    return instance; // Return the single instance
  }
  public void log(String message) {
    System.out.println("Log: " + message);
  }
}
```

What's different here?

• The volatile keyword ensures that when one thread updates the instance, it's visible to all other threads. This prevents any threads from getting an outdated version of the Logger object.
• We only use the synchronized block once—when the instance is null and needs to be created.

After that, any thread can access the already-created Logger instance without needing synchronization.

How It Works:

1. First Check: The getInstance() method first checks if the instance is already created (i.e., not null). If it is, no synchronization is needed, and the method immediately returns the existing instance.
2. Second Check (Inside Synchronized Block): If the instance is still null, we enter the synchronized block, ensuring that only one thread can create the instance.
3. Efficient Access: Once the instance is created, other threads can access it without needing to wait.

This makes the Singleton thread-safe without the performance cost of synchronizing on every call to getInstance().

Summary: How We Solved the Problem 🎉

By using Double-Checked Locking, we made sure that:
• Only one instance of the Logger is created, even in a multithreaded environment.
• Threads don't block each other unnecessarily after the instance is created, which keeps the application efficient.
• The use of volatile ensures that changes to the instance are visible across all threads.

This approach gives us a safe and efficient way to implement the Singleton pattern in a multithreaded environment, ensuring that the Logger instance remains consistent and is only created once, even when multiple threads try to access it at the same time.

Conclusion 🎉

The Singleton Design Pattern is a powerful way to ensure that a class has only one instance throughout the application. By using a private constructor and a static method to access the instance, the Singleton pattern simplifies resource management, particularly for things like logging, database connections, and configuration management.

In a multithreaded environment, we can make the Singleton thread-safe by using Double-Checked Locking and the volatile keyword, ensuring that only one instance is created even when multiple threads are involved.
The Singleton Pattern is widely used in many real-world applications because of its ability to provide consistent access to a single resource, helping to reduce memory usage and increase efficiency.