Singleton

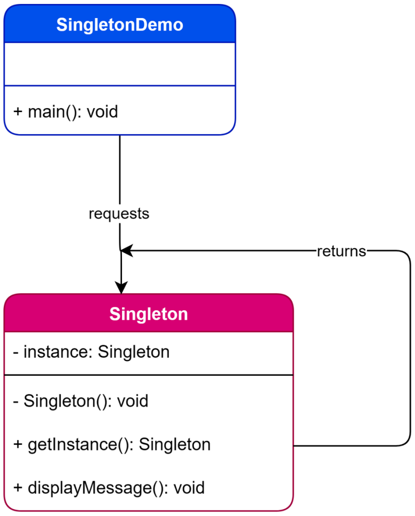
Resources

- Video → 27. All Creational Design Patterns | Prototype, Singleton, Factory, AbstractFactory, Builder Pattern
- Video → 28. BUG in Double-Checked Locking of Singleton Pattern & its Fix | Low Level System Design Question

Definition

The Singleton design pattern is used when we have to create only ONE instance of a class. This is useful when exactly one object is needed to perform and coordinate many actions across the system. A Singleton class ensures that there is only one instance of itself, regardless of the number of clients attempting to access it.

Class Diagram



Implementation of Singleton Pattern

There are 4 ways to implement the Singleton Pattern, each with its advantages and disadvantages.

1. Eager Initialization

- One of the simplest implementations of the Singleton Pattern.
- The instance is created when the class is loaded.
- Make the constructor `private` → prevents instantiation by other classes.
- Mark the singleton instance field as:
  - `static` variable → this ensures there's only one class instance shared across all instances of the class.
  - `final` variable → prevents the singleton instance from being reassigned after initialization.

Implementation

```java
// 1. Eager Initialization - Singleton
public class DBConnectionEager {

    // The single instance, created immediately
    private static final DBConnectionEager instance = new DBConnectionEager();

    // The private constructor prevents instantiation
    private DBConnectionEager() {
    }

    // Method to return the unique instance of this class
    public static DBConnectionEager getInstance() {
        return instance;
    }

    // Method to display a message
    public void displayMessage() {
        System.out.println("Eager Initialization - Singleton - " + this);
    }
}
// Test Singleton Implementation
public class SingletonDemo {
    public static void main(String[] args) {
        System.out.println("====== Testing Eager Initialization ======");
        // SingletonObject obj = new SingletonObject(); --> Compilation error
        // Get the unique instance of SingletonObject
        DBConnectionEager eager1 = DBConnectionEager.getInstance();
        DBConnectionEager eager2 = DBConnectionEager.getInstance();
        // Display the message
        eager1.displayMessage();
        eager2.displayMessage();
        // Check if the instances are the same
        System.out.println("Same instance? " + (eager1 == eager2)); //true
      }
}
```

2. Lazy Initialization

- The instance is created only once when the client requests it.
- Make the constructor `private` → prevents instantiation by other classes.
- It is not thread-safe. When multiple threads access the `getInstance` method for the first time, there is a chance of multiple singleton instances being created.
- Mark the singleton instance field as:
  - `static` variable → this ensures there's only one class instance shared across all instances of the class.

Implementation

```java
// 2. Lazy Initialization Singleton
public class DBConnectionLazy {

    private static DBConnectionLazy instance = null;

    // The private constructor prevents instantiation
    private DBConnectionLazy() {
    }

    // Singleton Object is created only when it is required
    // This method returns the unique instance of this class
    // Drawback: This implementation is not thread-safe.
    public static DBConnectionLazy getInstance() {
        if (instance == null) {
            instance = new DBConnectionLazy();
        }
        return instance;
    }

    // Method to display a message
    public void displayMessage() {
        System.out.println("Lazy Initialization - Singleton - " + this);
    }
}
// Test Singleton Implementation
public class SingletonDemo {
    public static void main(String[] args) {
      System.out.println("====== Testing Lazy Initialization ======");
        // Get the unique instance of SingletonObject
        DBConnectionLazy lazy1 = DBConnectionLazy.getInstance();
        DBConnectionLazy lazy2 = DBConnectionLazy.getInstance();
        // Display the message
        lazy1.displayMessage();
        lazy2.displayMessage();
        // Check if the instances are the same
        System.out.println("Same instance? " + (lazy1 == lazy2)); // true
      }
}
```

3. Thread-Safe/Synchronized Intialization

- This is similar to Lazy Initialization
- It is made thread-safe by using the `synchronized` keyword in the `getInstance` method, ensuring only one thread can execute this method at a time.
- This will ensure only one singleton instance is created when multiple threads are invoking the same `getInstance` method.
- But this process of singleton instantiation can be expensive because when a thread enters the synchronized method, it acquires a **lock** on the **class object**, does the job, and releases the lock for other threads to acquire it and execute the method.
    - Imagine 100 threads invoking the synchronized `getInstance` method simultaneously; 99 will have to wait until the first thread finishes its execution.

Implementation

```java
// 3. Thread Safe Singleton
public class DBConnectionThreadSafe {

    // Singleton Object is created only when it is required
    private static DBConnectionThreadSafe instance = null;

    // Private Constructor to avoid client applications from using the constructor
    private DBConnectionThreadSafe() {
    }

    // Thread Safe Method to return the unique instance of this class
    public static synchronized DBConnectionThreadSafe getInstance() {
        if (instance == null) {
            instance = new DBConnectionThreadSafe();
        }
        return instance;
    }

    // Method to display a message
    public void displayMessage() {
        System.out.println("Thread Safe Singleton - " + this);
    }
}
// Test Singleton Implementation
public class SingletonDemo {
    public static void main(String[] args) {
       System.out.println("====== Testing Thread Safe ======");
        // Get the unique instance of SingletonObject
        DBConnectionThreadSafe threadSafe1 = DBConnectionThreadSafe.getInstance();
        DBConnectionThreadSafe threadSafe2 = DBConnectionThreadSafe.getInstance();
        // Display the message
        threadSafe1.displayMessage();
        threadSafe2.displayMessage();
        // Check if the instances are the same
        System.out.println("Same instance? " + (threadSafe1 == threadSafe2)); //true
    }
}
```

4. Double Locking

- It is a more optimised version of thread-safe singleton object instantiation.
- This method reduces performance overhead from synchronization(as seen previously) by only synchronizing the block of code when the singleton object is initially created(the first time). Upon instantiation, the other/future threads do not have to enter the synchronized block. Thus, making it faster.
- Widely used in the industry.

Implementation

```
// 4. Double Locking Singleton
public class DBConnectionDoubleLocking {

    // Double Locking Singleton instance variable
    private static DBConnectionDoubleLocking instance = null;

    // Private constructor
    private DBConnectionDoubleLocking() {
    }

    // Thread Safe Method to return the unique instance of this class
    public static DBConnectionDoubleLocking getInstance() {
        if (instance == null) { // first check
            synchronized (DBConnectionDoubleLocking.class) {
                if (instance == null) { // second check
                    instance = new DBConnectionDoubleLocking();
                }
            }
        }
```

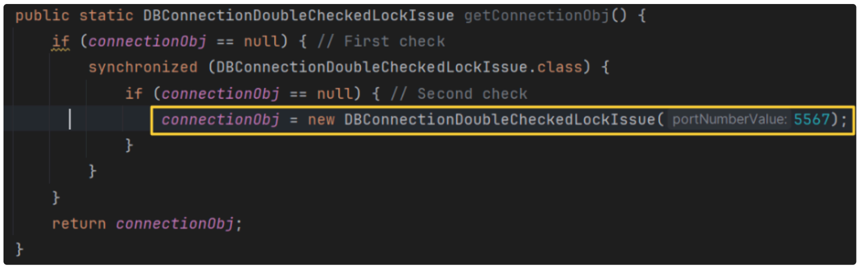
```java
        return instance;
    }

    // Method to display a message
    public void displayMessage() {
        System.out.println("Double Locking Singleton - " + this);
    }
}
// Test Singleton Implementation
public class SingletonDemo {
    public static void main(String[] args) {
      System.out.println("====== Testing Double Locking ======");
        // Get the unique instance of SingletonObject
        DBConnectionDoubleLocking doubleLocking1 = DBConnectionDoubleLocking.getInstance();
        DBConnectionDoubleLocking doubleLocking2 = DBConnectionDoubleLocking.getInstance();
        // Display the message
        doubleLocking1.displayMessage();
        doubleLocking2.displayMessage();
        // Check if the instances are the same
        System.out.println("Same instance? " + (doubleLocking1 == doubleLocking2)); // true

    }
}
```

Issue with Double-Checked Locking

Video explanation → 28. BUG in Double-Checked Locking of Singleton Pattern & its Fix | Low Level System Design Question

Consider the following example with a member variable `portNumber` :

```java
// Singleton - Double Checked Locking - Issue Demo
public class DBConnectionDoubleCheckedLockIssue {

    // Double Locking Singleton instance variable
    private static DBConnectionDoubleCheckedLockIssue connectionObj = null;
    int portNumber;

    // Private constructor
    private DBConnectionDoubleCheckedLockIssue(int portNumberValue) {
        portNumber = portNumberValue;
    }

    // Thread Safe Method to return the unique instance of this class
    public static DBConnectionDoubleCheckedLockIssue getConnectionObj() {
        if (connectionObj == null) { // First check
            synchronized (DBConnectionDoubleCheckedLockIssue.class) {
                if (connectionObj == null) { // Second check
                    connectionObj = new DBConnectionDoubleCheckedLockIssue(5567);
                }
            }
        }
        return connectionObj;
    }

    // Method to display a message
    public void displayMessage() {
       System.out.println("Singleton - Double Checked Locking - Issue - " + this);
    }
}
```



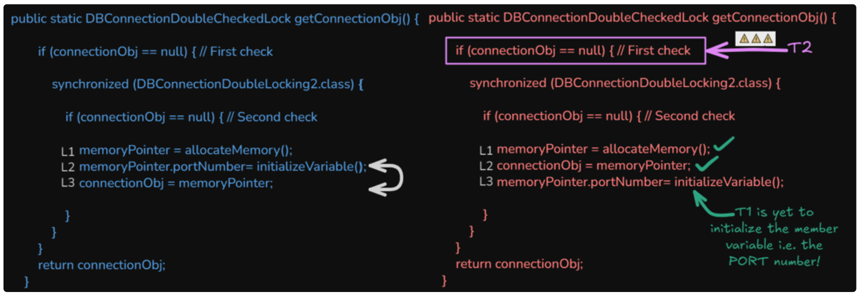
```java
public static DBConnectionDoubleCheckedLockIssue getConnectionObj() {
    if (connectionObj == null) { // First check
        synchronized (DBConnectionDoubleCheckedLockIssue.class) {
            if (connectionObj == null) { // Second check
                connectionObj = new DBConnectionDoubleCheckedLockIssue( portNumberValue: 5567);
            }
        }
    }
    return connectionObj;
}
```

The above highlighted step(creation of object) involves the following steps:

1. Allocating memory
2. Initializing the object with all member variables
3. Assigning the reference

Issue 1: Instruction Reordering

The JVM can reorder the object construction steps to achieve better performance. The JVM might **assign the reference before fully initializing the object**, causing other threads to see a **partially constructed object.**



```java
public static DBConnectionDoubleCheckedLock getConnectionObj() {

    if (connectionObj == null) { // First check

        synchronized (DBConnectionDoubleLocking2.class) {

          if (connectionObj == null) { // Second check

            L1 memoryPointer = allocateMemory();
            L2 memoryPointer.portNumber= initializeVariable();
            L3 connectionObj = memoryPointer;

          }
        }
    }
    return connectionObj;
}
```

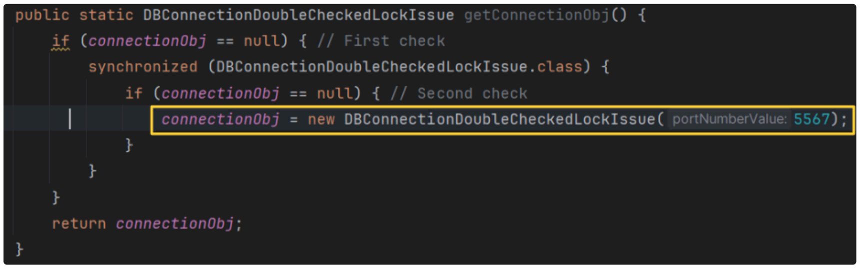
```java
public static DBConnectionDoubleCheckedLock getConnectionObj() {

    if (connectionObj == null) { // First check

        synchronized (DBConnectionDoubleLocking2.class) {

          if (connectionObj == null) { // Second check

            L1 memoryPointer = allocateMemory();
            L2 connectionObj = memoryPointer;
            L3 memoryPointer.portNumber= initializeVariable();

          }
        }
    }
    return connectionObj;
}
```

⚠⚠⚠
T2
T1 is yet to initialize the member variable i.e. the PORT number!

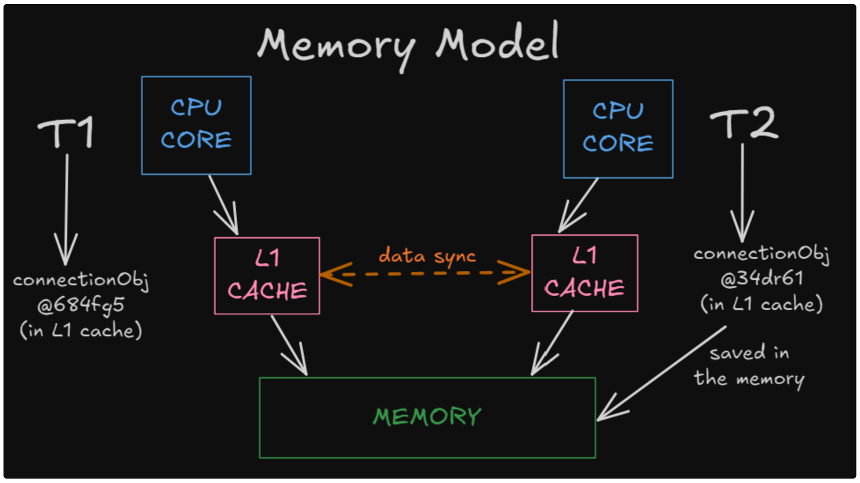
> ⚠️ While T1 is executing the reordered instructions, when T1 is executing L3, let’s say T2 performs the first check, it gets the `connectionObject` i.e. `NOT NULL` and the `portNumber` holds the default value(after execution of L2 by T1) and proceeds with performing operations using the partially constructed `connectionObject` , which can result in an error as T1 is yet to initialize the `portNumber` (member variable).

Issue 2: L1 Caching

Without proper synchronization, changes made by one thread may not be visible to other threads because of CPU caching and memory models.



```java
public static DBConnectionDoubleCheckedLockIssue getConnectionObj() {
    if (connectionObj == null) { // First check
        synchronized (DBConnectionDoubleCheckedLockIssue.class) {
            if (connectionObj == null) { // Second check
                connectionObj = new DBConnectionDoubleCheckedLockIssue( portNumberValue: 5567);
            }
        }
    }
    return connectionObj;
}
```



⚠ In a multicore CPU, each CPU core has its own cache; sometimes, cores might not synchronize their caches, leading to inconsistent views of memory. When T1 creates a fully constructed `connectionObject` and saves it in the L1 cache, and if another thread T2 requests a `connectionObject` before the Caches have been synchronised and the changes have been saved in memory, T2 will proceed to create another singleton `connectionObject` instance and perform operations with it. This will lead to multiple singleton instances.

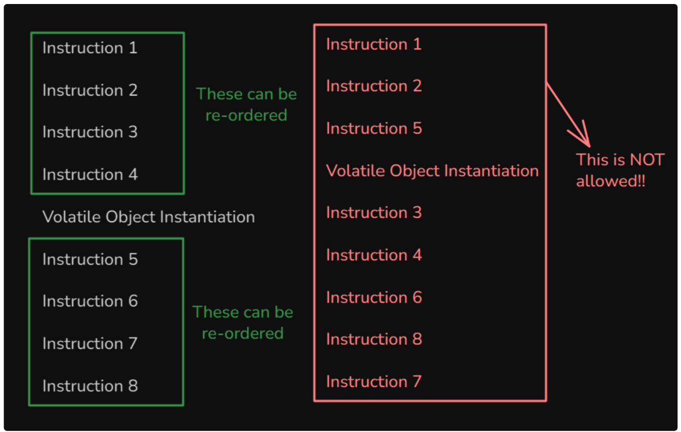
The Correct Solution: Using the `volatile` keyword

Understanding the `volatile` keyword

When a variable is declared as volatile, it instructs the JVM and the compiler to handle that variable in a specific way. This modifier is primarily used in multithreaded programming to ensure memory visibility, consistency of shared variables across different threads, and instruction ordering.

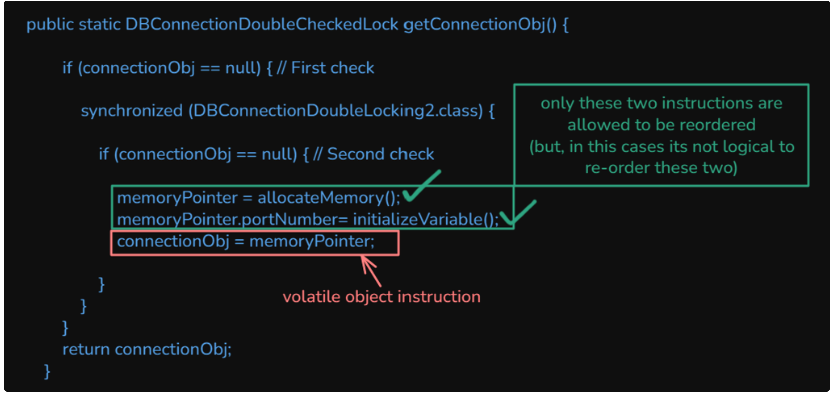
How do `volatile` keyword properties solve the issues with double-checked locking?

- **Memory Visibility Guarantee:** All reads and writes of a `volatile` variable are always performed directly to and from the main memory, and all write operations are immediately visible to all threads. Without `volatile`, threads might cache variables locally and not see updates from other threads. **[Solves Issue 2]**
- **Instruction Re-ordering:** The `volatile` keyword provides guarantees to prevent specific types of instruction reordering, as illustrated below. This prevention is accomplished by establishing "happens-before" guarantees, which effectively create memory barriers or fences. These barriers stop the compiler and CPU from reordering instructions around `volatile` reads or writes, even if such reordering would enhance performance in a single-threaded context.
  [Solves Issue 1]



Instruction 1
Instruction 2
Instruction 3
Instruction 4
Volatile Object Instantiation
Instruction 5
Instruction 6
Instruction 7
Instruction 8

Instruction 1
Instruction 2
Instruction 5
Volatile Object Instantiation
Instruction 3
Instruction 4
Instruction 6
Instruction 8
Instruction 7



```
public static DBConnectionDoubleCheckedLock getConnectionObj() {
    if (connectionObj == null) { // First check
        synchronized (DBConnectionDoubleLocking2.class) {
            if (connectionObj == null) { // Second check
                memoryPointer = allocateMemory();
                memoryPointer.portNumber= initializeVariable();
                connectionObj = memoryPointer;
            }
        }
    }
    return connectionObj;
}
```

only these two instructions are allowed to be reordered
(but, in this cases its not logical to re-order these two)
volatile object instruction

Implementation: Using the volatile keyword

```java
// Singleton - Double Checked Locking - Fix Demo
public class DBConnectionDoubleCheckedLockFix {

    // volatile keyword is used to ensure that the value of the variable
    // is fetched from the memory every time.
    private static volatile DBConnectionDoubleCheckedLockFix connectionObj = null;

    int portNumber;

    private DBConnectionDoubleCheckedLockFix(int portNumberValue) {
        portNumber = portNumberValue;
    }

    // Thread Safe Method to return the unique instance of this class
    public static DBConnectionDoubleCheckedLockFix getConnectionObj(int portNumberValue) {
        if(connectionObj == null) {
            synchronized(DBConnectionDoubleCheckedLockFix.class) {
                if(connectionObj == null) {
                    connectionObj = new DBConnectionDoubleCheckedLockFix(portNumberValue);
                }
            }
        }
        return connectionObj;
    }

    // Method to display a message
    public void displayMessage() {
        System.out.println("Singleton - Double Checked Locking - Fix - " + this);
    }
}
```