THE SINGLETON PATTERN

DOUBLE-CHECKED LOCKING

In software engineering, **double-checked locking** (also known as "**double**-checked **locking** optimization") is a software design pattern used to reduce the overhead of acquiring a **lock** by first testing the **locking** criterion (the **"lock** hint") without actually acquiring the **lock**.

Double-checked locking - Wikipedia, the free encyclopedia
https://en.wikipedia.org/wiki/**Double-checked_locking**

More about Double-checked locking

CHECK THE SAME CONDITION TWICE

```
//Check 1 of the double-checked lock
if (singleton == null) {
    synchronized (Singleton.class) {
        // Check 2 of the double-checked lock
        if (singleton == null) {
            singleton = new Singleton();
        }
    }
}
```

WITH A LOCK IN-BETWEEN

SYNCHRONIZED CAN LEAD TO QUITE A PERFORMANCE HIT - YOU CAN GET AROUND THIS IN 2 WAYS:

Declaring a **volatile Java** variable means: The value of this variable will never be cached thread-locally: all reads and writes will go straight to "main memory"; Access to the variable acts as though it is enclosed in a synchronized block, synchronized on itself.

1. EAGERLY INSTANTIATE THE SINGLETON (NO NEED TO SYNCHRONIZE THE GETTER AFTER THAT)

```
public class Singleton {

    // We maintain just one private static instance
    // This is the singleton object

    // Instantiate this member variable eagerly, to
    // make sure that this happens exactly once, and
    // the getter then need not be synchronized
    private volatile static Singleton singleton = new Singleton();

    private Singleton() {
        // this private constructor is the key:
        // nobody can instantiate outside this class
    }

    public static Singleton getInstance() {
        return singleton;
    }

}
```

2. DOUBLE-CHECKED LOCKING * MARK THE MEMBER VARIABLE AS "VOLATILE"

```
public class Singleton {

    // We maintain just one private static instance
    // This is the singleton object

    // Mark the member variable as volatile, so each
    // access of this variable is a fresh read from
    // memory
    private volatile static Singleton singleton;

    private Singleton() {
        // this private constructor is the key:
        // nobody can instantiate outside this class
    }

    // use a standard double-checked locking test
    // so that the synchronization penalty is only
    // incurred the first time, when the Singleton is
    // null. On all subsequent calls, this penalty is
    // avoided. The use of the volatile keyword then
    // prevents any thread from reading a stale version
    // of the Singleton
    public static Singleton getInstance() {
        //Check 1 of the double-checked lock
        if (singleton == null) {
            synchronized (Singleton.class) {
                // Check 2 of the double-checked lock
                if (singleton == null) {
                    singleton = new Singleton();
                }
            }
        }
```

JAVA-SPECIFIC

SOME FINE PRINT

TECHNICALLY IT IS POSSIBLE TO END UP WITH 2 SINGLETON OBJECTS - DESPITE ALL THESE PRECAUTIONS - IF YOUR CODE USES MULTIPLE CLASS LOADERS

(A CLASS LOADER IS A PART OF THE JAVA VIRTUAL MACHINE - TECHNICALLY NAMESPACES ARE UNIQUE PER CLASS LOADER. USUALLY THERE IS JUST 1 CLASS LOADER PER PROGRAM)

IT IS POSSIBLE TO GET AROUND THIS CLASS LOADER SUBTLETY USING ENUMS, WHICH ARE GUARANTEED TO BE INSTANTIATED EXACTLY ONCE

A SINGLETON COULD ALSO BE REPLICATED VIA A CLASS WITH ALL VARIABLES AND METHODS STATIC..

..BUT SUBTLE BUGS CAN RESULT FROM THE ORDER OF INITIALIZATION OF VARIABLES.