Superman Problem

This lesson is about correctly implementing a singleton pattern in Java

Problem

You are designing a library of superheroes for a video game that your fellow developers will consume. Your library should always create a single instance of any of the superheroes and return the same instance to all the requesting consumers.

Say, you start with the class `Superman`. Your task is to make sure that other developers using your class can never instantiate multiple copies of superman. After all, there is only one superman!

Solution

You probably guessed we are going to use the **singleton** pattern to solve this problem. The singleton pattern sounds very naive and simple but when it comes to implementing it correctly in Java, it's no cakewalk.

First let us understand what the pattern is. **A singleton pattern allows only a single object/instance of a class to ever exist during an application run.**

There are two requirements to make a class adhere to the singleton pattern:

- Declaring the constructor of a class `private`. When you declare the `Superman` class's constructor `private` then the constructor isn't visible outside the class or in its subclasses. Only the instance and static methods of the `Superman` class are able to access the constructor and create instances of the `Superman` class.
- The second trick is to create a public static method usually named `getInstance()` to return the only instance. We create a private static object of the class `Superman` and return it via the `getInstance()` method. We can control when to instantiate the lone static private instance. Here's what the code looks like:

```
public class SupermanNaiveButCorrect {

    // We are initializing the object inline
    private static SupermanNaiveButCorrect superman = new SupermanNai
veButCorrect();

    // We have marked the constructor private
    private SupermanNaiveButCorrect() {
    }

    public static SupermanNaiveButCorrect getInstance() {
        return superman;
    }

    // Object method
    public void fly() {
        System.out.println("I am Superman & I can fly !");
    }
}
```

Here's what your interviewer will tell you when you write this code:

- What if the no one likes Superman and instead creates Batman in the game. You just created Superman and he kept waiting without ever being called upon to save the world. It's a waste of Superman's time and also the memory and other resources he'll consume.
- What if the creation of Superman is a very resource-intensive effort after all he's coming from planet Krypton. We would really like to only create Superman once we need him. Or in programming-speak, we want to ***lazily initialize*** Superman

The next version is what most candidates would write and is incorrect.

```
public class SupermanWithFlaws {

    private static SupermanWithFlaws superman;

    private SupermanWithFlaws() {

    }

    // This will fail with multiple threads
    public static SupermanWithFlaws getInstance() {
        if (superman == null) {
            // A thread can be context switched at this point and
            // superman will evaluate to null for any other threads
            // testing the if condition. Now multiple threads will
            // fall into this if clause till the superman object is
            // assigned a value. All these threads will intialize the
            // superman object when it should have been initialized
            // only one.
            superman = new SupermanWithFlaws();
        }
        return superman;
    }

    // Object method
    public void fly() {
        System.out.println("I am Superman & I can fly !");
    }
}
```

As any reader of this course should realize by now (if I have done a good job of teaching) that the `getInstance()` method would fail miserably in a multi-threaded scenario. A thread can context switch out just before it initializes the Superman, causing later threads to also fall into the if clause and end up creating multiple superman objects.

The naive way to fix this issue is to use our good friend `synchronized` and either add `synchronized` to the signature of the `getInstance()` method or add a `synchronized` block within the method body. Thee mutual exclusion ensures that only one thread gets to initialize the object.

```
public class SupermanCorrectButSlow {

    private static SupermanCorrectButSlow superman;

    private SupermanCorrectButSlow() {

    }

    public static SupermanCorrectButSlow getInstance() {
        synchronized(Superman.class) {
            if (superman == null) {
                superman = new SupermanWithFlaws();
            }
         }
        return superman;
    }

    // Object method
    public void fly() {
        System.out.println("I am Superman & I can fly !");
    }
}
```

The con of the above solution is that every invocation of the `getInstance()` method causes the invoking thread to synchronize, which is prohibitively more expensive in terms of performance than non-synchronized snippets of code. Can we synchronize only when initializing the singleton instance and not at other times? The answer is yes and leads us to an implementation known as **double checked locking**. The idea is that we do two checks for `superman == null` in a nested fashion. The first check is without synchronization and the second with. Once a singleton instance has been initialized, all future invocations of the `getInstance()` method don't pass the first null check and return the instance without getting involved in synchronization. Effectively, threads only synchronize when the singleton instance has not yet been initialized.

```
public class SupermanSlightlyBetter {

    private static SupermanSlightlyBetter superman;

    private SupermanSlightlyBetter() {

    }

    public static SupermanSlightlyBetter getInstance() {

        // Check if object is uninitialized
        if (superman == null) {

            // Now synchronize on the class object, so that only
            // 1 thread gets a chance to initialize the superman
            // object. Note that multiple threads can actually find
            // the superman object to be null and fall into the
            // first if clause
            synchronized (SupermanSlightlyBetter.class) {

                // Must check once more if the superman object is sti
ll

                // null. It is possible that another thread might hav
e

                // initialized it already as multiple threads could h
ave

                // made past the first if check.
                if (superman == null) {
                    superman = new SupermanSlightlyBetter();
                }
            }

        }

        return superman;
    }
}
```

The above solution seems almost correct. In fact, it'll appear correct unless you understand how the intricacies of Java's memory model and compiler optimizations can affect thread behaviors. The memory model defines what state a thread may see when it reads a memory location modified by other threads. The above solution needs one last missing piece but before we add that consider the below scenario:

1. Thread A comes along and gets to the second if check and allocates memory for the superman object but doesn't complete construction of the object and gets switched out. The Java memory model doesn't ensure that the constructor completes before the reference to the new object is assigned to an instance. It is possible that the variable `superman` is non-null but the object it points to, is still being initialized in the constructor by another thread.

2. Thread B wants to use the superman object and since the memory is already allocated for the object it fails the first if check and returns a semi-constructed superman object. Attempt to use a partially created object results in a crash or undefined behavior.

To fix the above issue, we mark our `superman` static object as `volatile`. The *happens-before* semantics of `volatile` guarantee that the faulty scenario of threads A and B never happens.

By now you'll probably appreciate how hard it is to get the singleton pattern right in a multithreaded scenario. Also, note that the discussed solution works with Java 1.5 or above. `volatile`'s behavior in Java 1.4 and earlier is different and the **double checked locking** pattern is broken when run on those versions of Java.

Last but not the least, double-checked locking (DCL) is an antipattern and its utility has dwindled over time as the JVM startup and uncontended synchronization speeds have improved.

The next lesson explains alternate Singleton implementations in Java.

Complete Code

The complete code appears below:

```
public class Superman {

    private static volatile Superman superman;

    private Superman() {

    }

    public static Superman getInstance() {

        if (superman == null) {
            synchronized (Superman.class) {

                if (superman == null) {
                    superman = new Superman();
                }
            }
        }

        return superman;
    }

    public void fly() {
        System.out.println("I am Superman & I can fly !");
    }
}
```