(/)

Double-Checked Locking with Singleton

Last modified: November 18, 2018

by Donato Rimenti (https://www.baeldung.com/author/donato-rimenti/)

Java (https://www.baeldung.com/category/java/) +

1. Introduction

In this tutorial, we'll talk about the double-checked locking design pattern. This pattern reduces the number of lock acquisitions by simply checking the locking condition beforehand. As a result of this, there's usually a performance boost.

Let's take a deeper look at how it works.

2. Implementation

To begin with, let's consider a simple singleton with draconian synchronization:

```java
public class DraconianSingleton {
    private static DraconianSingleton instance;
    public static synchronized DraconianSingleton getInstance() {
        if (instance == null) {
            instance = new DraconianSingleton();
        }
        return instance;
    }

    // private constructor and other methods ...
}
```

Despite this class being thread-safe, we can see that there's a clear performance drawback: each time we want to get the instance of our singleton, we need to acquire a potentially unnecessary lock.

To fix that, **we could instead start by verifying if we need to create the object in the first place and only in that case we would acquire the lock.**

Going further, we want to perform the same check again as soon as we enter the synchronized block, in order to keep the operation atomic:

```java
public class DclSingleton {
    private static volatile DclSingleton instance;
    public static DclSingleton getInstance() {
        if (instance == null) {
            synchronized (DclSingleton .class) {
                if (instance == null) {
                    instance = new DclSingleton();
                }
            }
        }
        return instance;
    }

    // private constructor and other methods...
}
```

One thing to keep in mind with this pattern is that **the field needs to be *volatile*** to prevent cache incoherence issues. In fact, the Java memory model allows the publication of partially initialized objects and this may lead in turn to subtle bugs.

3. Alternatives

Even though the double-checked locking can potentially speed things up, it has at least two issues:

- since it requires the *volatile* keyword to work properly, it's not compatible with Java 1.4 and lower versions
- it's quite verbose and it makes the code difficult to read

For these reasons, let's look into some other options without these flaws. All of the following methods delegate the synchronization task to the JVM.

3.1. Early Initialization

The easiest way to achieve thread safety is to inline the object creation or to use an equivalent static block. This takes advantage of the fact that static fields and blocks are initialized one after another (Java Language Specification 12.4.2 (https://docs.oracle.com/javase/specs/jls/se7/html/jls-12.html#jls-12.4.2)):

```java
public class EarlyInitSingleton {
    private static final EarlyInitSingleton INSTANCE = new EarlyInitSingleton();
    public static EarlyInitSingleton getInstance() {
        return INSTANCE;
    }

     // private constructor and other methods...
}
```

3.2. Initialization on Demand

Additionally, since we know from the Java Language Specification reference in the previous paragraph that a class initialization occurs the first time we use one of its methods or fields, we can use a nested static class to implement lazy initialization:

```
public class InitOnDemandSingleton {
    private static class InstanceHolder {
        private static final InitOnDemandSingleton INSTANCE = new InitOnDemandSingleton();
    }
    public static InitOnDemandSingleton getInstance() {
        return InstanceHolder.INSTANCE;
    }

     // private constructor and other methods...
}
```

In this case, the *InstanceHolder* class will assign the field the first time we access it by invoking *getInstance.*

3.3. Enum Singleton

The last solution comes from the *Effective Java* book (Item 3) by Joshua Block and uses an *enum* instead of a *class*. At the time of writing, this is considered to be the most concise and safe way to write a singleton:

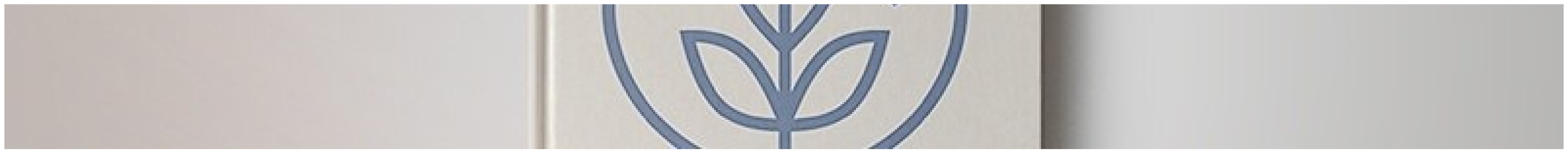
```
public enum EnumSingleton {
    INSTANCE;

    // other methods...
}
```

4. Conclusion

To sum up, this quick article went through the double-checked locking pattern, its limits and some alternatives.

In practice, the excessive verbosity and lack of backward compatibility make this pattern error-prone and thus we should avoid it. Instead, we should use an alternative that lets the JVM do the synchronizing.

As always, the code of all examples is available on GitHub (https://github.com/eugenp/tutorials/tree/master/patterns/design-patterns).



CATEGORIES

SPRING (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SPRING/)