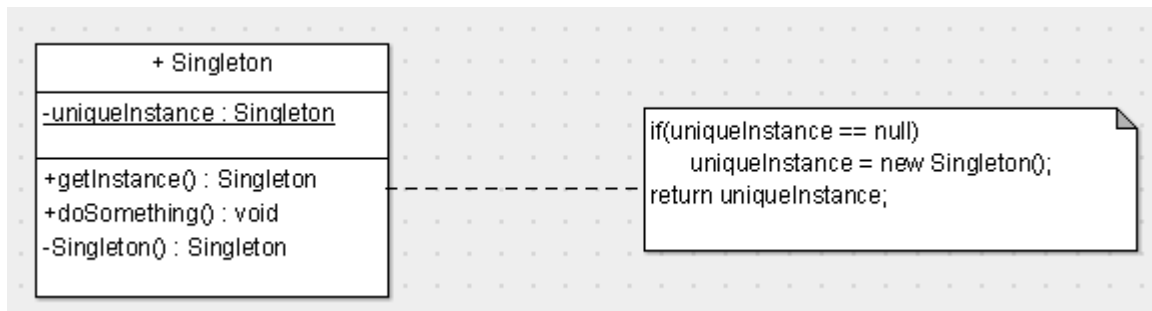


# Singleton Design Pattern

Sometimes it's appropriate to have exactly one instance of a class: window managers, print spoolers, and filesystems are prototypical examples. Typically, those types of objects—known as singletons—are accessed by disparate objects throughout a software system, and therefore require a global point of access.



A Java beginner will know about singleton design pattern. At least he will think that he knows singleton pattern. The definition is even easier than Newton's third law. Then what is special about the singleton pattern? Is it so simple and straightforward? Do you believe that you know 100% about singleton design pattern? If you believe so and you are a beginner read through the end, there are surprises for you.

There are only two points in the definition of a singleton design pattern,

- there should be only one instance allowed for a class and
- we should allow global point of access to that single instance.
- Maybe allow multiple instances in the future without affecting a singleton class's clients

GOF says, “Ensure a class has only one instance, and provide a global point of access to it. [GoF, p127]”.

The key is not the problem and definition. In singleton pattern, trickier part is implementation and management of that single instance. Two points looks very simple, is it so difficult to implement it. Yes it is very difficult to ensure “single instance” rule.

Implementation is very specific to the language you are using. So the security of the single instance is specific to the language used.

### Example 1. The classic singleton

```
public class ClassicSingleton {  
    private static ClassicSingleton instance = null;  
    protected ClassicSingleton() {  
        // Exists only to defeat instantiation.  
    }  
    public static ClassicSingleton getInstance() {  
        if(instance == null) {  
            instance = new ClassicSingleton();  
        }  
        return instance;  
    }  
}
```

The singleton implemented in Example 1 is easy to understand. The `ClassicSingleton` class maintains a static reference to the lone singleton instance and returns that reference from the static `getInstance()` method.

There are several interesting points concerning the `ClassicSingleton` class.

First, `ClassicSingleton` employs a technique known as *lazy instantiation* to create the singleton; as a result, the singleton instance is not created until the `getInstance()` method is called for the first time. This technique ensures that singleton instances are created only when needed.

Second, notice that `ClassicSingleton` implements a protected constructor so clients cannot instantiate `ClassicSingleton` instances; however, client code in the same package can call the constructor directly.

This dilemma has two solutions: You can make the `ClassicSingleton` constructor private so that only `ClassicSingleton()` methods call it; however, that means `ClassicSingleton` cannot be subclassed. Sometimes, that is a desirable solution; if so, it's a good idea to declare your singleton class `final`, which makes that intention explicit and allows the compiler to apply performance optimizations.

The other solution is to put your singleton class in an explicit package, so classes in other packages (including the default package) cannot instantiate singleton instances.

Thirdly, if `ClassicSingleton` implements the `java.io.Serializable` interface, the class's instances can be serialized and deserialized. However, if you serialize a singleton object and subsequently deserialize that object more than once, you will have multiple singleton instances.

Finally, and perhaps most important, Example 1's `ClassicSingleton` class is not thread-safe. If two threads—we'll call them Thread 1 and Thread 2—call `ClassicSingleton.getInstance()` at the same time, two `ClassicSingleton` instances can be created if Thread 1 is preempted just after it enters the `if` block and control is subsequently given to Thread 2.

As you can see from the preceding discussion, although the Singleton pattern is one of the simplest design patterns, implementing it in Java is anything but simple.

## Synchronization

Making singleton class thread-safe is easy—just synchronize the `getInstance()` method like this:

```
public synchronized static Singleton getInstance() {
    if (singleton == null) {
        simulateRandomActivity();
        singleton = new Singleton();
    }
    logger.info("created singleton: " + singleton);
    return singleton;
}
```

However, you may realize that the `getInstance()` method only needs to be synchronized the first time it is called. Because synchronization is very expensive performance-wise (synchronized methods can run up to 100 times slower than unsynchronized methods), perhaps we can introduce a performance enhancement that only synchronizes the singleton assignment in `getInstance()`.

```
public static Singleton getInstance() {
    if (singleton == null) {
        synchronized (Singleton.class) {
            singleton = new Singleton();
        }
    }
    return singleton;
}
```

Instead of synchronizing the entire method, the preceding code fragment only synchronizes the critical code. However, the preceding code fragment is not thread-safe. Consider the following scenario: Thread 1 enters the synchronized block, and, before it can assign the `singleton` member variable, the thread is preempted. Subsequently,

another thread can enter the `if` block. The second thread will wait for the first thread to finish, but we will still wind up with two distinct singleton instances. Is there a way to fix this problem?

## Double-checked locking

Double-checked locking is a technique that, at first glance, appears to make lazy instantiation thread-safe. That technique is illustrated in the following code fragment:

```
public static Singleton getInstance() {  
    if (singleton == null) {  
        synchronized (Singleton.class) {  
            if (singleton == null) {  
                singleton = new Singleton();  
            }  
        }  
    }  
    return singleton;  
}
```

What happens if two threads simultaneously access `getInstance()`? Imagine Thread 1 enters the synchronized block and is preempted. Subsequently, a second thread enters the `if` block. When Thread 1 exits the synchronized block, Thread 2 makes a second check to see if the `singleton` instance is still `null`. Since Thread 1 set the `singleton` member variable, Thread 2's second check will fail, and a second singleton will not be created. Or so it seems.

Unfortunately, double-checked locking is not guaranteed to work because the compiler is free to assign a value to the `singleton` member variable before the singleton's constructor is called.

Multithreading was considered, other issues can arise from classloaders and serialization. For full discussion see this article:

<http://www.javaworld.com/article/2073352/core-java/simply-singleton.html>