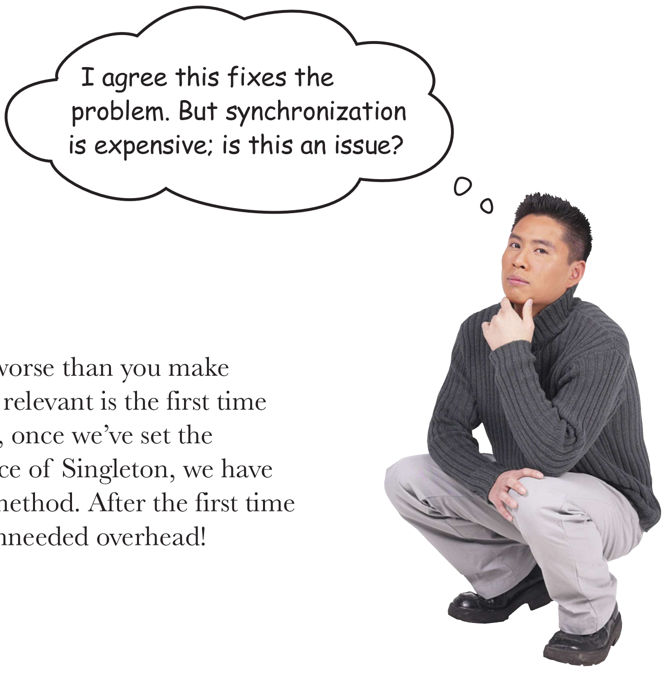
Dealing with multithreading

Our multithreading woes are almost trivially fixed by making getInstance() a synchronized method:

```
public class Singleton {
    private static Singleton uniqueInstance;

    // other useful instance variables here

    private Singleton() {}

    public static synchronized Singleton getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
        }
        return uniqueInstance;
    }

    // other useful methods here
}
```

By adding the synchronized keyword to getInstance(), we force every thread to wait its turn before it can enter the method. That is, no two threads may enter the method at the same time.


I agree this fixes the problem. But synchronization is expensive; is this an issue?

Good point, and it's actually a little worse than you make out: the only time synchronization is relevant is the first time through this method. In other words, once we've set the uniqueInstance variable to an instance of Singleton, we have no further need to synchronize this method. After the first time through, synchronization is totally unneeded overhead!

Can we improve multithreading?

For most Java applications, we obviously need to ensure that the Singleton works in the presence of multiple threads. But it's expensive to synchronize the getInstance() method, so what do we do?

Well, we have a few options...

1. Do nothing if the performance of getInstance() isn't critical to your application.

That's right; if calling the getInstance() method isn't causing substantial overhead for your application, forget about it. Synchronizing getInstance() is straightforward and effective. Just keep in mind that synchronizing a method can decrease performance by a factor of 100, so if a high-traffic part of your code begins using getInstance(), you may have to reconsider.

2. Move to an eagerly created instance rather than a lazily created one.

If your application always creates and uses an instance of the Singleton, or the overhead of creation and runtime aspects of the Singleton isn't onerous, you may want to create your Singleton eagerly, like this:

```
public class Singleton {
    private static Singleton uniqueInstance = new Singleton();

    private Singleton() {}

    public static Singleton getInstance() {
        return uniqueInstance;
    }
}
```

Go ahead and create an instance of Singleton in a static initializer. This code is guaranteed to be thread safe!

We've already got an instance, so just return it.

Using this approach, we rely on the JVM to create the unique instance of the Singleton when the class is loaded. The JVM guarantees that the instance will be created before any thread accesses the static uniqueInstance variable.

3. Use "double-checked locking" to reduce the use of synchronization in getInstance().

With double-checked locking, we first check to see if an instance is created, and if not, THEN we synchronize. This way, we only synchronize the first time through, just what we want.

Let's check out the code:

```
public class Singleton {
    private volatile* static Singleton uniqueInstance;

    private Singleton() {}

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

Check for an instance and if there isn't one, enter a synchronized block.

Note we only synchronize the first time through!

Once in the block, check again and if it's still null, create an instance.

*The volatile keyword ensures that multiple threads handle the uniqueInstance variable correctly when it is being initialized to the Singleton instance.

If performance is an issue in your use of the getInstance() method, then this method of implementing the Singleton can drastically reduce the overhead.

Watch it!

Double-checked locking doesn't work in Java 1.4 or earlier!

If for some reason you're using an old version of Java, unfortunately, in Java version 1.4 and earlier, many JVMs contain implementations of the volatile keyword that allow improper synchronization for double-checked locking. If you must use a JVM earlier than Java 5, consider other methods of implementing your Singleton.