

Controlling Global Access to Your Java Objects with the Singleton Design Pattern

by [Barry A. Burd](#) and [Michael P. Redlich](#)

Introduction

This article, the seventh in a series about design patterns, introduces the Singleton design pattern, one of the 23 design patterns defined in the legendary 1995 book *Design Patterns – Elements of Reusable Object-Oriented Software*. The authors of this book, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, are known affectionately as the *Gang of Four* (GoF).

Design Patterns

The GoF book defines 23 design patterns. The patterns fall into three categories:

- A creational pattern abstracts the instantiation process.
- A structural pattern groups objects into larger structures.
- A behavioral pattern defines better communication among objects.

The Singleton design pattern fits into the creational category.

The Singleton Pattern

According to the GoF book, the Singleton design pattern “ensures a class has only one instance, and provides a global point of access to it.” You use the Singleton pattern to satisfy four simultaneous requirements:

- An application must have exactly one instance of a particular class.
- The sole instance must be accessible to clients from a well-known access point.
- The sole instance should be extensible by subclassing.
- Clients must be able to use and extend the instance without modifying their own code.

Motivation

An application uses a system resource such as a print spooler or file manager. The system has only one such resource so your application should have only one instance of a class representing the resource. You may decide to create a global variable. With a global variable, you may have only one instance at a time. But by calling the constructor several times, you can still create several instances, one after another.

The Singleton design pattern ensures that the application has only one global instance of an object. The pattern's code is quite simple, but behind this simplicity, the pattern has some potential pitfalls. This article describes the pattern, its pitfalls, and two ways to avoid the pitfalls.

UML Diagram

Figure 1 shows a UML diagram for the Singleton design pattern.

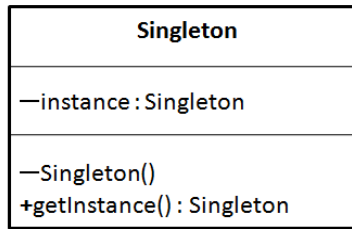


Figure 1: A UML diagram for the Singleton design pattern

This is the simplest UML diagram of all the 23 GoF design patterns. The diagram's essential concepts are as follows:

- The default constructor is **private** so you can't use the **new** keyword to instantiate the class.
- The **Singleton** class has a **getInstance ()** method. This method returns the value of the static variable, **instance**, of type **Singleton**.

Implementing the Singleton Design Pattern

Listings 1 and 2 illustrate a generic implementation of the Singleton design pattern.

```
// Warning: This code is flawed!

public class Singleton
{
    private static Singleton instance;

    private Singleton()
    {}

    public static Singleton getInstance()
    {
        if(instance == null)
        {
            System.out.println
                ("Creating a new Singleton instance...");
            instance = new Singleton();
        }

        System.out.println
            ("Returning instance " + instance + "...");
        return instance;
    }
}
```

Listing 1: The Singleton class

Notice that the default constructor is **private**. If you try to create an instance of **Singleton** using the **new** keyword, you get a compile time error.

```
public class SingletonTest
```

```
{
private static final int MAX = 10;

public static void main(String[] args)
{
    for(int i = 0;i < MAX; ++i)
        Singleton.getInstance();
}
}
```

Listing 2: The SingletonTest class – a client application

The client application in Listing 2 calls `Singleton.getInstance()` ten times. During the first call to `Singleton.getInstance()`, Listing 1 returns a newly created `Singleton` instance. During the remaining nine calls to `Singleton.getInstance()`, the variable `instance` isn't `null` so Listing 1 returns the existing (unique) instance. Simple, eh?

The output of the client code appears in Figure 2.

```
Creating a new Singleton instance...
Returning instance Singleton@19821f...
Returning instance Singleton@19821f...
Returning instance Singleton@19821f...
Returning instance Singleton@19821f...
Returning instance Singleton@19821f...
Returning instance Singleton@19821f...
Returning instance Singleton@19821f...
Returning instance Singleton@19821f...
Returning instance Singleton@19821f...
Returning instance Singleton@19821f...
```

Figure 2: The output of the singleton test application

Notice that the instance `Singleton@19821f` is exactly the same for all ten iterations of the `for` loop. The Singleton design pattern ensures that an application has only one global instance.

So What's the Potential Pitfall?

For a client application like the one in Listing 2, the Singleton pattern works well. But in a multithreaded application, the seemingly a simple `Singleton` pattern can become messy. Imagine a machine zippering together two threads, with each thread executing its own `getInstance()` method call. The interleaved sequence of steps are described in Figure 3.


Time	Thread 1	Thread 2
	Thread 1 tests the condition <code>instance == null</code> and learns that <code>instance</code> is indeed <code>null</code> . So Thread 1 enters the body of the <code>if</code> statement in Listing 1.	
		Thread 2 tests the condition <code>instance == null</code> and learns that <code>instance</code> is still <code>null</code> . So Thread 2 enters the body of the <code>if</code> statement in Listing 1.
		Thread 2 calls <code>instance = new Singleton()</code> and returns the instance.
	The variable <code>instance</code> is no longer <code>null</code> , but <i>Thread 1 is already inside the <code>if</code> statement</i> . So Thread 1 calls <code>instance = new Singleton()</code> and returns a <i>second</i> instance.	

Figure 3: Listings 1 and 2 accidentally create two instances of the Singleton class.

In Figure 3, an application creates two instances of the `Singleton` class because Thread 1 hesitates in the middle of its run. Of course, you may ask, "What's keeping Thread 1 from continuing its run." For many scenarios, the answer is "nothing". Much of the time, Thread 1 doesn't hesitate and the `Singleton` class lives up to its name. An application creates only one `Singleton` instance.

But this non-hesitating behavior isn't guaranteed. You can work with a flawed singleton program, and never catch the flaw in testing. To help catch the flaw, author David Geary suggests adding code that simulates random activity. (See the references at the end of this article.) Listing 3 illustrates the idea.

```
// Warning: This code is flawed!

import java.util.Random;

public class Singleton
{
    private static Singleton instance;

    private static Random random = new Random();
    private static final int MAX_SLEEP_TIME = 5000;

    private Singleton()
    {}

    public static Singleton getInstance()
    {
        if(instance == null)
        {
            simulateRandomActivity();
            System.out.println
                ("Creating a new Singleton instance...");
            instance = new Singleton();
        }

        System.out.println(Thread.currentThread().getName() +
            " returning instance " + instance + "...");
        return instance;
    }

    private static void simulateRandomActivity()
    {
        try
        {
            Thread.sleep(random.nextInt(MAX_SLEEP_TIME));
        }
        catch (InterruptedException exception)
        {
            System.out.println("Sleep interrupted: " +
                Thread.currentThread().getName() + ": " +
                exception.toString());
        }
    }
}

```

Listing 3: The Singleton class with random activity

In Listing 3, method `simulateRandomActivity()` brings Geary's idea to life. As in any singleton class, the `getInstance()` method checks to determine if the `instance` variable is `null`. But when the `instance` variable is `null`, the `getInstance()` method calls a special `simulateRandomActivity()` method. The `simulateRandomActivity()` method puts the first thread to sleep for a random number of milliseconds -- enough time for a second thread to rush in and make trouble (the kind of trouble you see in Figure 3).

To bring out the worst in Listing 3, you write a test that fires up a few threads. A test is shown in Listing 4.

```
public class SingletonTest
{
    public static void main(String[] args)

```

```

{
    Thread[] threads = new Thread[10];

    for (int i = 0; i < threads.length; i++)
        {
            threads[i] = new Thread()
                {
                    public void run()
                        {
                            Singleton.getInstance();
                        }
                };
        }

    for (int i = 0; i < threads.length; i++)
        {
            threads[i].start();
        }
}

```

Listing 4: A client program to run the code in Listing 3

Figure 4 shows the output from a run of the bad singleton code -- the code in Listings 3 and 4.

```

Creating a new Singleton instance...
Thread-5 returning instance Singleton@1b67f74...
Creating a new Singleton instance...
Thread-3 returning instance Singleton@69b332...
Creating a new Singleton instance...
Thread-2 returning instance Singleton@173a10f...
Creating a new Singleton instance...
Thread-1 returning instance Singleton@530daa...
Creating a new Singleton instance...
Thread-0 returning instance Singleton@a62fc3...
Creating a new Singleton instance...
Thread-7 returning instance Singleton@89ae9e...
Creating a new Singleton instance...
Thread-4 returning instance Singleton@1270b73...
Creating a new Singleton instance...
Thread-8 returning instance Singleton@60aeb0...
Creating a new Singleton instance...
Thread-9 returning instance Singleton@16caf43...
Creating a new Singleton instance...
Thread-6 returning instance Singleton@8813f2...

```

Figure 4: The result of coding the Singleton pattern incorrectly

Instead of creating only one instance, Listings 3 and 4 generate ten different instances of the `Singleton` class. That's bad. By the very nature of pseudo-randomness, a run of these listings doesn't always generate ten different instances. But the code indicates a vulnerability. Without some kind of protection against *laissez-faire* multithreading, the naïve singleton code is dangerous and incorrect.

Avoiding the Pitfall

The easiest way to dodge a multithreading bullet is to synchronize a method. In fact, when you add one word to the code in Listing 3, you avoid the pitfall.

```
public static synchronized Singleton getInstance()
```

Listing 5 contains the corrected code.

```
import java.util.Random;

public class Singleton
{
    private static Singleton instance;

    private static Random random = new Random();
    private static final int MAX_SLEEP_TIME = 5000;

    private Singleton()
    {}

    public static synchronized Singleton getInstance()
    {
        if(instance == null)
        {
            simulateRandomActivity();
            System.out.println
                ("Creating a new Singleton instance...");
            instance = new Singleton();
        }

        System.out.println(Thread.currentThread().getName() +
            " returning instance " + instance + "...");
        return instance;
    }

    private static void simulateRandomActivity()
    {
        try
        {
            Thread.sleep(random.nextInt(MAX_SLEEP_TIME));
        }
        catch (InterruptedException exception)
        {
            System.out.println("Sleep interrupted: " +
                Thread.currentThread().getName() + ": " +
                exception.toString());
        }
    }
}
```

Listing 5: A copy of Listing 3 with the `synchronized` keyword added

By adding the word **synchronized**, you insist that the Java Virtual Machine execute only one copy of the `getInstance()` method at a time. If Thread 1 starts executing the `getInstance()` method then, even if Thread 2 calls `simulateRandomActivity()`, Thread 2 must wait. That is, Thread 2 can't begin executing `getInstance()` until Thread 1 has returned from its call to `getInstance()`. The nasty scenario illustrated in Figure 3 can't happen. The output of a run of Listings 4 and 5 is shown in Figure 5.

```
Creating a new Singleton instance...
```

```
Thread-0 returning instance Singleton@69b332...
Thread-9 returning instance Singleton@69b332...
Thread-7 returning instance Singleton@69b332...
Thread-8 returning instance Singleton@69b332...
Thread-5 returning instance Singleton@69b332...
Thread-6 returning instance Singleton@69b332...
Thread-3 returning instance Singleton@69b332...
Thread-1 returning instance Singleton@69b332...
Thread-4 returning instance Singleton@69b332...
Thread-2 returning instance Singleton@69b332...
```

Figure 5: The output after adding the `synchronized` keyword to the `getInstance ()` method

In Figure 5, the threads return from calls to `getInstance ()` in no particular order. But each thread returns the same instance. The Singleton pattern's prime directive (that there be only one instance of the `Singleton` class) has been satisfied.

Isn't Synchronization Expensive?

In Listing 5 only one thread at a time can execute `getInstance ()` method's code. But letting one thread hog an entire method's code is wasteful. Figure 6 shows what may happen after a `Singleton` instance has already been created.

Time	Thread 101	Thread 102
	Thread 101 starts executing the <code>getInstance()</code> method.	
		Thread 102 tries to start executing the <code>getInstance()</code> method, but Thread 102 must wait for Thread 101 to return from the method.
	Thread 101 tests the condition <code>instance == null</code> and learns that <code>instance</code> is not <code>null</code> . So Thread 101 skips the creation of a new instance, and proceeds to the method's last few statements.	
	At last, Thread 101 returns from the call to the <code>getInstance()</code> method.	
		Thread 102 starts executing the <code>getInstance()</code> method.
		Thread 102 tests the condition <code>instance == null</code> and learns that <code>instance</code> is not <code>null</code> ...
		And so on.

Figure 6: A possible scenario with method synchronization

In Figure 6, Thread 102 waits needlessly to discover that `instance` isn't `null`. This waiting isn't necessary. As a rule, you should let threads execute whatever statements are safe for them to execute. That is, you should narrow the amount of synchronized code as much as possible.

But of course you must be careful. If you synchronize too little code, you get the awful effect illustrated in Figure 3. So how do you find an optimal amount of synchronized code?

Double-Checked Locking

To control the overhead due to synchronization, you can use *double-checked locking*. The word "double" refers to the fact that you check twice to determine if the variable `instance` is `null`. Here's how it works:

If `instance` is `null`:

- Synchronize
- Check the "`instance` is `null`" condition again
- Create the unique instance of `instance`

The code is shown in Listing 6.

```

import java.util.Random;

public class Singleton
{
    private static volatile Singleton instance;

    private static Random random = new Random();
    private static final int MAX_SLEEP_TIME = 5000;

    private Singleton()
    {}

    public static Singleton getInstance()
    {
        if(instance == null)
        {
            synchronized(Singleton.class)
            {
                if(instance == null)
                {
                    simulateRandomActivity();
                    System.out.println
                        ("Creating a new Singleton instance...");
                    instance = new Singleton();
                }
            }
        }
        System.out.println(Thread.currentThread().getName() +
            " returning instance " + instance + "...");
        return instance;
    }

    private static void simulateRandomActivity()
    {
        try
        {
            Thread.sleep(random.nextInt(MAX_SLEEP_TIME));
        }
        catch(InterruptedException exception)
        {
            System.out.println("Sleep interrupted: " +
                Thread.currentThread().getName() + ": " +
                exception.toString());
        }
    }
}

```

Listing 6: The Singleton class with double-checked locking

At first glance, Listing 6 looks strange. "Are you sure that `instance == null`?" How reliable is that second check? It reminds you of the familiar dialog box. "Are you sure you want to delete this file?" Your sarcastic reply is "I wouldn't have pressed Delete if I didn't want to delete the file." Sometimes you click "Yes" and then think "Oops! I clicked 'Yes' out of habit. I didn't really want to delete the file." So what good is double-checking?

Well, double-checked locking isn't like having a dialog box double-check your file deletion. In fact, double-checked locking works quite nicely. You can grasp the main idea by examining Figures 7 and 8. Figure 7 illustrates the creation of the first `Singleton` instance. Figure 8 shows what can happen *after* a `Singleton` instance has been created.


Time	Thread 1	Thread 2
	Thread 1 tests the condition <code>instance == null</code> and learns that <code>instance</code> is indeed <code>null</code> . So Thread 1 enters the outer <code>if</code> statement of Listing 6.	
		Thread 2 tests the condition <code>instance == null</code> and learns that <code>instance</code> is still <code>null</code> . So Thread 2 enters the outer <code>if</code> statement of Listing 6.
	Thread 1 enters the synchronized block of code.	
		Thread 2 tries to enter the synchronized block of code. but Thread 2 must wait for Thread 1 to finished executing the synchronized block.
	Thread 1 tests the condition <code>instance == null</code> and learns that <code>instance</code> is still <code>null</code> . So Thread 1 creates an instance, and exits the synchronized block.	
		Thread 2 enters the synchronized block of code.
		Thread 2 tests the inner <code>if</code> statement's <code>instance == null</code> condition. Thread 2 learns that <code>instance</code> is not <code>null</code> ... And so on.

Figure 7: A possible scenario with double-checked locking

Figure 7 shows that double-checked locking gives you only one instance of the `Singleton` class.

Time	Thread 101	Thread 102
	Thread 101 starts executing the <code>getInstance()</code> method.	
		Thread 102 starts executing the <code>getInstance()</code> method.
	Thread 101 tests the condition <code>instance == null</code> and learns that <code>instance</code> is not <code>null</code> . So Thread 101 skips the creation of a new instance, and proceeds to the method's last few statements.	
		Thread 102 tests the condition <code>instance == null</code> and learns that <code>instance</code> is not <code>null</code> . So Thread 102 skips the creation of a new instance, and proceeds to the method's last few statements.
	Thread 101 returns from the call to the <code>getInstance()</code> method.	
		Thread 102 returns from the call to the <code>getInstance()</code> method.

Figure 8: A scenario later in the double-checked locking run

Figure 8 shows that double-checked locking works more efficiently than the method synchronization in Figure 6.

Over the past few years, many people have written articles criticizing the effectiveness of double-checked locking. The criticism helped motivate JSR-133 which was implemented in Java 5. A change in the memory model that included safer use of variables being declared `volatile` made it possible to use double-checked locking more effectively.

The keyword `volatile` warns the Java Virtual Machine that a particular variable may be used simultaneously by more than one thread. Java 5 strengthens the way in which a volatile variable acquires and releases locks. The local copy of the variable within a given thread is ensured to be correct because the JVM reads the master copy of the variable (not a cached copy) from memory each time the variable's value is used. A volatile variable does not participate in certain potentially risky compiler optimizations.

Uses of the Singleton in the Java API

The Singleton design pattern is used quite extensively in the Java API. Just look for any method named `getInstance()` in the JDK documentation.

Resources

Design Patterns – Elements of Reusable Object-Oriented Software

<http://www.awprofessional.com/bookstore/product.asp?isbn=0201633612&rl=1>

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides

ISBN 0-201-63361-2

Head First Design Patterns

<http://www.oreilly.com/catalog/hfdesignpat/>

Eric & Elisabeth Freeman

ISBN 0-596-00712-4

Object-Oriented Software Construction

<http://vig.prenhall.com/catalog/academic/product/0,1144,0136291554.html,00.html>

Bertrand Meyer

ISBN 0-13-629155-4

Data & Object Factory

<http://www.dofactory.com/Patterns/Patterns.aspx>

Simply Singleton: Navigate the Deceptively Simple Singleton Pattern

David Geary

JavaWorld, April 25, 2003

<http://www.javaworld.com/javaworld/jw-04-2003/jw-0425-designpatterns.html>

Double-Checked Locking in Java

Michael Gilfix

Michael Gilfix Online, March 16, 2007

<http://www.michaelgilfix.com/techblog/2007/03/16/double-checked-locking>

About the Authors

[Barry Burd](#) is a professor in the [Department of Mathematics and Computer Science](#) at [Drew University](#) in Madison, New Jersey. When he's not lecturing at Drew University, Dr. Burd leads training courses for professional programmers in business and industry. He has lectured at conferences in America, Europe, Australia and Asia. He is the author of several articles and books, including [Java For Dummies](#) and [Ruby on Rails For Dummies](#), both published by [Wiley](#).

[Michael P. Redlich](#) is a Senior Research Technician (formerly a Systems Analyst) at [ExxonMobil Research & Engineering, Co.](#) in Clinton, New Jersey with extensive experience in developing custom web and scientific laboratory applications. Mike also has experience as a Technical Support Engineer for [Ai-Logix, Inc.](#) where he developed computer telephony applications. As a member of the [Amateur Computer Group of New Jersey \(ACGNJ\)](#), he dedicates much of his free time facilitating the monthly [ACGNJ Java Users Group](#) and serving on the ACGNJ Board of Directors. Mike is the current ACGNJ President and has previously served as Secretary. He has a Bachelor of Science in Computer Science from [Rutgers University](#). Mike's computing experience includes computer security, relational database design and development, object-oriented design and analysis, C/C++, Java, Visual Basic, FORTRAN, Pascal, MATLAB, HTML, XML, ASP, VBScript, and JavaScript in both the PC and UNIX environments.

The authors thank Jeanne Boyarsky and Mike Krier for their technical assistance in writing this article.