When is a Singleton not a Singleton?
by Joshua Fox; Reprinted from JavaWorld

Published January 2001

The Singleton is a useful Design Pattern for allowing only one instance of your class, but common mistakes can inadvertently allow more than one instance to be created. In this article, I'll show you how that can happen and how to avoid it.

The Singleton's purpose is to control object creation, limiting the number to one but allowing the flexibility to create more objects if the situation changes. Since there is only one Singleton instance, any instance fields of a Singleton will occur only once per class, just like static fields.

Singletons often control access to resources such as database connections or sockets. For example, if you have a license for only one connection for your database or your JDBC driver has trouble with multithreading, the Singleton makes sure that only one connection is made or that only one thread can access the connection at a time. If you add database connections or use a JDBC driver that allows multithreading, the Singleton can be easily adjusted to allow more connections.

Moreover, Singletons can be stateful; in this case, their role is to serve as a unique repository of state. If you are implementing a counter that needs to give out sequential and unique numbers (such as the machine that gives out numbers in the deli), the counter needs to be globally unique. The Singleton can hold the number and synchronize access; if later you want to hold counters in a database for persistence, you can change the private implementation of the Singleton without changing the interface.

On the other hand, Singletons can also be stateless, providing utility functions that need no more information than their parameters. In that case, there is no need to instantiate multiple objects that have no reason for their existence, and so a Singleton is appropriate.

The Singleton should not be seen as way to implement global variables in the Java programming language; rather, along the lines of the factory design patterns, the Singleton lets you encapsulate and control the creation process by making sure that certain prerequisites are fulfilled or by creating the object lazily on demand.

However, in certain situations, two or more Singletons can mysteriously materialize, disrupting the very guarantees that the Singleton is meant to provide. For example, if your Singleton `Frame` is meant as a global user interface for your application and two are created, your application will have two `Frame`s on the screen -- quite confusing for the user. Further, if two counters are created where one was intended, then clients requesting numbers will not get the desired sequence 1, 2, 3... but rather a multiple sequence such as 1, 1, 2, 2, 3, 3, 3.... Additionally, if several instances of a database-connection Singleton are created, you might start receiving `SQLExceptions` complaining about "too many database connections."

In this article, I'll describe those phenomena and how to avoid them. After discussing how to implement the Singleton, I'll go over the sometimes surprising causes for the phenomena one by one, showing you how they occur and how you can avoid making those mistakes. I hope that in the process you will learn about classloading, multithreading, distributed systems, Design Patterns, and other interesting topics, as I did.

Implementing Singletons

There are a few ways to implement Singletons. Although you can get Singleton-like behavior with static fields and methods [for example, `java.lang.Math.sin(double)`], you gain more flexibility by creating an instance. With Singletons implemented as single instances instead of static class members, you can initialize the Singleton lazily, creating it only when it is first used. Likewise, with a Singleton implemented as single instance, you leave open the possibility of altering the class to create more instances in the future. With some implementations of the Singleton, you allow writers of subclasses to override methods polymorphically, something not possible with static methods.

Most commonly, you implement a Singleton in Java by having a single instance of the class as a static field. You can create that instance at class-loading time by assigning a newly created object to the static field in the field declaration, as seen in Listing 1.

Listing 1

```
public class MySingleton {
 private static MySingleton _instance =
  new MySingleton();

 private MySingleton() {
  // construct object . . .
 }

 public static MySingleton getInstance() {
  return _instance;
 }

 // Remainder of class definition . . .
```

Alternatively, you can instantiate it lazily, on first demand, as seen in Listing 2. You keep the constructor private to prevent instantiation by outside callers.

Listing 2

```
 public class MySingleton {
 private static MySingleton _instance;

 private MySingleton() {
  // construct object . . .
 }

 // For lazy initialization
 public static synchronized MySingleton getInstance() {
  if (_instance==null) {
   _instance = new MySingleton();
  }
  return _instance;
 }
  // Remainder of class definition . . .
}
```

Both Singleton implementations do not allow convenient subclassing, since `getInstance()`, being static, cannot be overridden polymorphically. Other implementations prove more flexible. While I don't have the space to describe alternative implementations in detail, here are a couple of possibilities:

If you implement a factory class with a method that returns the Singleton instance, you allow yourself flexibility in the runtime class of the return value, which can be either `MySingleton` or a subclass thereof. You may have to make the constructor nonprivate to make that work.
You can have a `SingletonFactory` class with a globally accessible map of class names or `Class` objects to Singleton instances. Again, the runtime type of the instances can be either the type indicated by the class name or a subclass, and the constructor will not be private.
Now that we've looked briefly at implementation, let's turn to the heart of this article: how two Singletons can exist simultaneously.

Multiple Singletons in Two or More Virtual Machines

When copies of the Singleton class run in multiple VMs, an instance is created for each machine. That each VM can hold its own Singleton might seem obvious but, in distributed systems such as those using EJBs, Jini, and RMI, it's not so simple. Since intermediate layers can hide the distributed technologies, to tell where an object is really instantiated may be difficult.

For example, only the EJB container decides how and when to create EJB objects or to recycle existing ones. The EJB may exist in a different VM from the code that calls it. Moreover, a single EJB can be instantiated simultaneously in several VMs. For a stateless session bean, multiple calls to what appears, to your code, to be one instance could actually be calls to different instances on different VMs. Even an entity EJB can be saved through a persistence mechanism between calls, so that you have no idea what instance answers your method calls. (The primary key that is part of the entity bean spec is needed precisely because referential identity is of no use in identifying the bean.)

The EJB containers' ability to spread the identity of a single EJB instance across multiple VMs causes confusion if you try to write a Singleton in the context of an EJB. The instance fields of the Singleton will not be globally unique. Because several VMs are involved for what appears to be the

same object, several Singleton objects might be brought into existence.

Systems based on distributed technologies such as EJB, RMI, and Jini should avoid Singletons that hold state. Singletons that do not hold state but simply control access to resources are also not appropriate for EJBs, since resource management is the role of the EJB container. However, in other distributed systems, Singleton objects that control resources may be used on the understanding that they are not unique in the distributed system, just in the particular VM.

Multiple Singletons Simultaneously Loaded by Different Class Loaders

When two class loaders load a class, you actually have two copies of the class, and each one can have its own Singleton instance. That is particularly relevant in servlets running in certain servlet engines (iPlanet for example), where each servlet by default uses its own class loader. Two different servlets accessing a joint Singleton will, in fact, get two different objects.

Multiple class loaders occur more commonly than you might think. When browsers load classes from the network for use by applets, they use a separate class loader for each server address. Similarly, Jini and RMI systems may use a separate class loader for the different code bases from which they download class files. If your own system uses custom class loaders, all the same issues may arise.

If loaded by different class loaders, two classes with the same name, even the same package name, are treated as distinct -- even if, in fact, they are byte-for-byte the same class. The different class loaders represent different namespaces that distinguish classes (even though the classes' names are the same), so that the two `MySingleton` classes are in fact distinct. Since two Singleton objects belong to two classes of the same name, it will appear at first glance that there are two Singleton objects of the same class.

Singleton Classes Destroyed by Garbage Collection, then Reloaded

When a Singleton class is garbage-collected and then reloaded, a new Singleton instance is created. Any class can be garbage-collected when no other object holds reference to the class or its instances. If no object holds a reference to the Singleton object, then the Singleton class may disappear, later to be reloaded when the Singleton is again needed. In that case, a new Singleton object will be created. Any static or instance fields saved for the object will be lost and reinitialized.

This problems exists in older Java Virtual Machines[1]. JDK 1.2 VMs, in particular, conform to a newer class garbage collection model that forbids any class in a given classloader to be collected until all are unreferenced Programming Java threads in the real world, Part 7 in Resources). You can avoid class garbage collection in the older VMs by holding a reference to the Singleton class or object in some other object that persists for the program's life. You can also set your VM to have no class garbage collection (`-Xnoclassgc` on the JRE 1.3, or `-noclassgc` on the IBM JVM). Keep in mind that if you have a long-running program that frequently reloads classes (perhaps through special class loaders such as the remote class loaders), you have to consider whether that could cause a problematic buildup of garbage classes in the VM.

Purposely Reloaded Singleton Classes

Classes are reloaded not only after class garbage-collection; they can also be reloaded at Java programs' request. The servlet specifications allow servlet engines to do that at any time. When the servlet engine decides to unload a servlet class, it calls `destroy()`, then discards the servlet class; later, the servlet engine can reload the servlet class, instantiate the servlet object, and initialize it by calling `init()`. In practice, the process of unloading and reloading may occur in a servlet engine when a servlet class or JSP changes.

Like the previous two cases, the present case involves newly loaded classes. Here, however, classes are ditched on purpose, while a new copy of the class loads.

Depending on the servlet engine, when an old servlet class is discarded, the associated classes might not be, even if they have changed. So if a servlet gets a reference to a Singleton object, you may find that there is one Singleton object associated with the old servlet class and one associated with the new.

As servlet engines differ in their class-reloading policies, the Singleton behavior is unpredictable unless you understand how your servlet engine's class-loading mechanisms work.

Similar problems can occur if you hold a reference from another object to a servlet and some chain of references keeps that object from the garbage collector. Then, when the servlet class should be discarded, it cannot be, and you may find the servlet class loaded twice in the VM.

Multiple Instances Resulting from Incorrect Synchronization

One of the common Singleton implementations uses lazy initialization of the one instance. That means that the instance is not created when the class loads, but rather when it is first used. (See Listing 2.) A common mistake with that implementation is to neglect synchronization, which can lead to multiple instances of the singleton class. (See Listing 3.)

Listing 3

```
// error, no synchronization on method
public static MySingleton getInstance() {
 if (_instance==null) {
  _instance = new MySingleton();
 }

 return _instance;
}
```

Two Singletons will be created if the constructor runs and simultaneously another thread calls the method. Thread-safe code is particularly important in Singletons, since that Design Pattern is meant to give the user a single point of access that hides the complexities of the implementation, including multithreading issues.

Multiple instances can be created even if you add a `synchronized(this)` block to the constructor call, as in Listing 4:

Listing 4

```
// Also an error, synchronization does not prevent
// two calls of constructor.
public static MySingleton getInstance() {
 if (_instance==null) {
   synchronized (MySingleton.class) {
      _instance = new MySingleton();
   }
 }
 return _instance;
}
```

In the correct solution, seen in Listing 5, make `getInstance()` a synchronized method:

Listing 5

```
// correct solution
public static synchronized MySingleton getInstance() {
// . . . continue as in Listing 3
```

Double-checked locking is another common solution but, unfortunately, it does not work (see Listing 6).

Listing 6

```
 // Double-checked locking -- don't use
public static MySingleton getInstance() {
 if (_instance==null) {
   synchronized (MySingleton.class) {
     if (_instance==null) {
        _instance = new MySingleton();
     }
   }
 }
}
```

In this situation, we intend to avoid the expense of grabbing the lock of the Singleton class every time the method is called. The lock is grabbed only if the Singleton instance does not exist, and then the existence of the instance is checked again in case another thread passed the first check an instant before the current thread.

Unfortunately, double-checked locking causes problems. To wit, compiler optimizations can make the assignment of the new Singleton object before all its fields are initialized. The only practical solution is to synchronize the `getInstance()` method (as in Listing 2).

Multiple Singletons Arising when Someone has Subclassed your Singleton

The Singleton Design Pattern is meant to give you control over access to the Singleton class. While I have mostly discussed the control of

instantiation, other code can access your class another way: by subclassing it.

The uniqueness of the class cannot be imposed as a compile-time constraint on the subclass unless you use a private constructor. If you want to allow subclassing, for example, you might make the constructor protected. A subclass could then expose a public constructor, allowing anyone to make instances. Since an instance of a subclass *is* an instance of your superclass, you could find multiple instances of the Singleton.

Multiple Singletons Created by a Factory Specially Asked to Create Multiple Objects

One of the strengths of the Singleton design pattern, as opposed to static methods, is that if you change your mind and want more than one, the Singleton class can be easily altered.

For example, most servlets run as Singletons in their servlet engines. Since that can cause threading problems, in one alternative (not recommended, but available) the servlet can implement `SingleThreadModel`. In that case, the servlet engine may, if necessary, create more than one servlet instance. If you are used to the more common Singleton servlets, you may forget that some servlets can occur in multiple instances.

Copies of a Singleton Object that has Undergone Serialization and Deserialization

If you have a serialized object and deserialize it twice in different `ObjectOutputStreams`, or with calls `ObjectOutputStream.reset()` between deserializations, you get two distinct objects, not two references to the same object.

Likewise, when you serialize an object with an `ObjectOutputStream`, the closure of its graph of references serializes with it. If the `ObjectOutputStream` closes and a new one opens, or if `reset()` is called on the `ObjectOutputStream`, the graph of references breaks and a new one is started. So if you serialize one Singleton twice, the two serialized objects take on separate identities.

The object serialization of the `java.io` package is not the only way to serialize Java objects. New mechanisms of object serialization with XML have been developed, including those associated with SOAP, WDDX, and KOALA, among others. With all those mechanisms, a reconstituted object loses its referential identity, and you have to consider carefully whether your Singleton is still a Singleton.

Multiple Singletons Caused by Problems in a Factory

In some implementations of creational design patterns in the factory family, the factory is a Singleton structured to create an instance of another class. That second class might not have any Singleton-like protection against multiple instantiation, on the grounds that the factory will ensure that it constructs only one instance.

If you accidentally have more than one factory object for one of the reasons above, you will still have two of the created objects, even if each factory object is built correctly.

Conclusion

Singletons are a useful way to control access to a class, making sure that only one instance can exist. In some none-too-uncommon circumstances, however, multiple instances can occur, even in a class coded as a Singleton. By being aware of the possibility, you can be sure that your Singleton really is a Singleton.

In this article, I've presented some ways that the multiple Singleton might emerge. If you've discovered any others, I'd be interested in hearing about them.

RESOURCES:

I have created classes that illustrate some of the points made in this article and offer suggestions for illustrating others. To download the source code, go to:
source code zip file
"Singleton's Rule" by Tony Sintes (*JavaWorld,* December 2000) outlines why Singletons promote good OO design compared to static classes:
Keep on the object-oriented track with singletons
"Programming Java Threads in the Real World, Part 7," Allen Holub (*JavaWorld,* April 1999) describes threading issues relevant to Singletons:
Singletons, critical sections, and reader/writer locks
"Java Tip 67: Lazy Instantiation," by Philip Bishop and Nigel Warren (*JavaWorld*) extensively discusses Singletons in the context of defered object creation:
Balancing performance and resource usage
"Create a Custom Java 1.2-style ClassLoader," Ken McCrary (*JavaWorld,* March 2000):
The Java 1.2 delegation model simplifies class-loading design and implementation
For answers to your pressing design patterns questions, check out the *JavaWorld* **Programming Theory & Practice** discussion:
ITworld.com Forums
To quickly search for other important *JavaWorld* articles based on subject, visit our useful Topical Index:
JavaWorld Topical Index
Sign up for the *JavaWorld This Week* free weekly email newsletter and keep up with what's new at *JavaWorld*:
ITworld.com Newsletters
The Patterns Home Page:
software patterns and pattern languages
"The Double-Checked Locking is Broken Declaration," David Bacon, et al.:
Double -Checked Locking
"Garbage Collection," Chapter 9 of *Inside the Java 2 Virtual Machine,* Bill Venners (McGraw-Hill Professional Publishing, 1999):
Inside The Java Virtual Machine, 2nd Edition
"Implementing the Singleton Pattern in Java," Rodney Waldhoff (August 8, 1998) explains a few ways to implement the Singleton in Java, including a couple that allow subclassing:
Implementing the Singleton Pattern in Java

About the Author

Joshua Fox is senior architect at Surf&Call Network Services at which he develops distributed Java systems. His current project is an Internet service that allows customers and call-center agents to surf the Web together. You can view his software engineering Website at http://www.joshuafox.com.