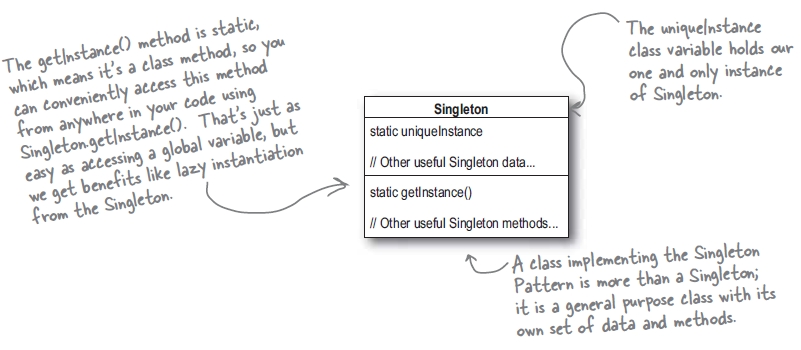
1. **Definition:** Singleton pattern ensures a class has only one instance and provides a global point to access to it.
   1. If the class is resource intensive it should be initialized lazily.



1. There use cases where we need only one instance of a class e.g., thread pools, caches, dialog box etc.
2. **Thread safety:**
   1. **public** **static** **synchronized** SingletonV1 getInstance() {

**if**(instance == **null**) {

instance = **new** SingletonV1();

}

**return** instance;

}

* + 1. here synchronization need only for 1st call when instance is created. For subsequent calls its pure overhead.
    2. If performance of the get instance method is not critical we can live with this implementation.
  1. **public** **static** SingletonV3 getInstance() {// thread safe

**return** *instance*;

}

* + 1. we can use this method if we are anyway creating an object of singleton class and the overhead of creating object is not enormous.
  1. **Double-check locking:**
     1. **public** **static** SingletonV3 getInstance() {

**if**(*instance* == **null**) {

**synchronized** (SingletonV3.**class**) {

**if**(*instance* == **null**) {

*instance* = **new** SingletonV3();

}

}

}

**return** *instance*;

}

* + 1. instance variable must be volatile to ensure multiple threads handle unique instance properly.
    2. This implementation doesn’t work on java 1.4 or earlier because of the implementation of volatile keyword.
    3. If there are two or more class loader each class loader can instantiate an instance of singleton. We can define our own class loader to get around this problem.
    4. The problem with double-checked locking is that there is no guarantee it will work on single or multi-processor machines.
       1. The issue of the failure of double-checked locking is not due to implementation bugs in JVMs but to the current Java platform memory model. The memory model allows what is known as "out-of-order writes" and is a prime reason why this idiom fails.
       2. The problem with this line of code ***instance* = new SingletonV3();** is that the variable instance can become non-null before the body of the Singleton constructor executes. So, if thread 1 preempted by thread 2 before it completes the constructor run, thread 2 would return partially initialized instance of singleton class. This argument is applicable for **jdk 4 or earlier**.
       3. We use volatile in double checking lock for sequential consistency. According to the JLS (see [Related topics](https://www.ibm.com/developerworks/library/j-dcl/index.html#artrelatedtopics)), variables declared volatile are supposed to be sequentially consistent, and therefore, not reordered.
          1. But problem is JVMs doesn’t implement consistency feature of volatile properly.

**private** **volatile** **boolean** stop = **false**;

**private** **volatile** **int** num = 0;

**public** **void** foo() {

num = 100;

//This can happen second

stop = **true**;

//This can happen first //...

}

**public** **void** bar() {

**if** (stop)

num += num;

//num can == 0!

}

* + - * 1. According to the JLS, because stop and num are declared volatile, they should be sequentially consistent. This means that if stop is ever true, num must have been set to 100. However, because many JVMs do not implement the sequential consistency feature of volatile, you cannot count on this behavior. Therefore, if thread 1 called foo and thread 2 called bar concurrently, thread 1 might set stop to true before num is set to 100. This could lead thread 2 to see stop as true, but num still set to 0. There are additional problems with volatile and the atomicity of 64-bit variables.
        2. Under the new memory model, this "fix" to double-checked locking renders the idiom thread-safe. But that still doesn't mean that you should use this idiom! The whole point of double-checked locking was that it was supposed to be a performance optimization, designed to eliminate synchronization on the common code path, largely because synchronization was relatively expensive in very early JDKs. Not only has uncontended synchronization gotten a lot cheaper since then, but the new changes to the semantics of volatile make it relatively more expensive than the old semantics on some platforms. (Effectively, each read or write to a volatile field is like "half" a synchronization -- a read of a volatile has the same memory semantics as a monitor acquire, and a write of a volatile has the same semantics as a monitor release.) If the goal of double-checked locking is supposed to offer improved performance over a more straightforward synchronized approach, this "fixed" version doesn't help very much either.
  1. **public** **static** SingletonV5 getInstance() {// double check lock

**if**(*instance* == **null**) {

**synchronized** (SingletonV5.**class**) {

SingletonV5 ins = *instance*;

**if**(ins == **null**) {

**synchronized** (SingletonV5.**class**) {

ins = **new** SingletonV5();

*instance* = ins;

}

}

}

}

**return** *instance*;

}

1. Even this code should work theoretically but won’t work.
2. The Java Language Specification (JLS) demands that code within a synchronized block not be moved out of a synchronized block. However, it does not say that code not in a synchronized block cannot be moved *into* a synchronized block. So, JVM sees an opportunity of code optimization here and combine ins = **new** SingletonV5(); *instance* = ins; in one line: instance = new Singleton();
   1. **private** **static** **class** SingletonV6Holder {

**public** **static volatile** SingletonV6 *instance* = **new** SingletonV6();

}

**public** **static** **synchronized** SingletonV6 getInstance() {// Initialize-on-demand Holder Class idiom

**return** SingletonV6Holder.*instance*;

}

1. This idiom derives its thread safety from the fact that operations that are part of class initialization, such as static initializers, are guaranteed to be visible to all threads that use that class, and its lazy initialization from the fact that the inner class is not loaded until some thread references one of its fields or methods.
   1. **enum** SingletonV7 {

***INSTANCE***();

**private** SingletonV7() {

}

}

1. Serialization treats enums specially. Basically, it stores only a reference to its class and the name of the constant. Upon deserialization, this information is used to lookup the existing runtime object of the enum type.

Thus, if you deserialize the enum constant within the same runtime, you will get the same runtime instance you have serialized.

However, when deserializing in another JVM, the hashcode might be different. But having the same hashcode is not a required criterion for singletons. The important point is to never have another instance of the class and this is guaranteed as the serialization implementation will never create an instance of an enum type but only lookup existing constants.

* 1. **Classloading issues:** 
     1. A class may be garbage-collected when it is no longer reachable. This behavior can be problematic when the program must maintain the singleton property throughout the entire lifetime of the program. A static singleton becomes eligible for garbage collection when its class loader becomes eligible for garbage collection. This usually happens when a nonstandard (custom) class loader is used to load the singleton.
        1. A simple scheme to prevent garbage collection is to ensure that there is a direct or indirect reference from a live thread to the singleton object that must be preserved.
        2. ObjectPreserver.preserveObject(singleton); // Preserve the object
     2. Because a singleton instance is associated with the class loader that is used to load it, it is possible to have multiple instances of the same class in the Java Virtual Machine. This situation typically occurs in J2EE containers and applets. Technically, these instances are different classes that are independent of each other.
        1. If you want to make sure the same classloader loads your singletons, you must specify the classloader yourself.
        2. private static Class getClass(String classname) throws ClassNotFoundException {

ClassLoader classLoader = Thread.currentThread().getContextClassLoader();

if (classLoader == null)

classLoader = SingletonV1.class.getClassLoader();

return classLoader.loadClass(classname);

}

* 1. **Cloning issue:** To avoid making the singleton class cloneable, do not implement the Cloneable interface and do not derive from a class that already implements it. When the singleton class must indirectly implement the Cloneable interface through inheritance, the object's clone()method must be overridden with one that throws a CloneNotSupportedException exception.
  2. **Serialization issue:** This noncompliant code example implements the java.io.Serializable interface, which allows the class to be serialized. Deserialization of the class implies that multiple instances of the singleton can be created. A singleton's constructor cannot install checks to enforce the requirement that the class is instantiated only once because deserialization can bypass the object's constructor.
     1. Adding a readResolve() method that returns the original instance is insufficient to enforce the singleton property. This technique is insecure even when all the fields are declared transient or static.
     2. If we depend on readResolve() for instance control all instance field with object reference must be declared transient. Otherwise its possible for an attacker to secure a reference to deserialized object before its readResolve() method run.
        1. Write a stealer class with readResolve() and an instance field of type singleton.
        2. In serialization stream replace non-primitive non-transient instance variable with an instance of stealer.

**public** **class** Untrusted **implements** Serializable {

**public** **static** MySingleton *captured*;

**public** MySingleton capture;

**public** Untrusted(MySingleton capture) {

**this**.capture = capture;

}

**private** **void** readObject(java.io.ObjectInputStream in) **throws** Exception {

in.defaultReadObject();

*captured* = capture;

}

}

* + - 1. While deserializing singleton stealers readResolve would be called first where it copies partially deserialized copy of singleton (where readResolve of singleton yet to be run) in a static field of type singleton.
      2. Stealers readResolve() return same type as of non-transient non-primitive instance variable.
      3. The crafted stream can be generated by serializing the following class:

**public** **final** **class** MySingleton **implements** java.io.Serializable {

**private** **static** **final** **long** ***serialVersionUID*** = 6825273283542226860L;

**public** Untrusted untrusted = **new** Untrusted(**this**); // Additional serial field

**public** MySingleton() {

}

}

* + - 1. Safe publication: <https://shipilev.net/blog/2014/safe-public-construction/>
      2. <http://vlkan.com/blog/post/2014/02/14/java-safe-publication/>