Thread Safety in Java Singleton Classes with Example Code

PANKAJ — 44 COMMENTS

Singleton is one of the most widely used creational design pattern to restrict the object creation by applications. In real world applications, resources like Database connections or Enterprise Information Systems (EIS) are limited and should be used wisely to avoid any resource crunch. To achieve this, we can implement Singleton design pattern to create a wrapper class around the resource and limit the number of object created at runtime to one.

Thread Safe Singleton in Java

a singleton class:

1. Override the private constructor to avoid any new object creation with new operator.
2. Declare a private static instance of the same class
3. Provide a public static method that will return the singleton class instance variable. If the variable is not initialized then initialize it or else simply return the instance variable.

Using above steps I have created a singleton class that looks like below:

`ASingleton.java`

```
package com.journaldev.designpatterns;

public class ASingleton {

        private static ASingleton instance = null;

        private ASingleton() {
        }

        public static ASingleton getInstance() {
                if (instance == null) {
                        instance = new ASingleton();
                }
                return instance;
        }

}
```

In the above code, getInstance() method is not thread safe. Multiple threads can access it at the same time and for the first few threads when the instance variable is not initialized, multiple threads can enters the if loop and create multiple instances and break our singleton implementation.

There are three ways through which we can achieve thread safety.

1. **Create the instance variable at the time of class loading.**
   Pros:

zation

Cons:

- Early creation of resource that might not be used in the application.
- The client application can't pass any argument, so we can't reuse it. For example, having a generic singleton class for database connection where client application supplies database server properties.

2. **Synchronize the getInstance() method**
   Pros:

   - Thread safety is guaranteed.
   - Client application can pass parameters
   - Lazy initialization achieved

   Cons:

   - Slow performance because of locking overhead.
   - Unnecessary synchronization that is not required once the instance variable is initialized.

3. **Use synchronized block inside the if loop and volatile variable**
   Pros:

   - Thread safety is guaranteed
   - Client application can pass arguments
   - Lazy initialization achieved
   - Synchronization overhead is minimal and applicable only for first few threads when the variable is null.

   Cons:

   - Extra if condition

Looking at all the three ways to achieve thread safety, I think third one is the best option and in that case the modified class will look like:

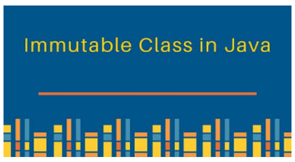
```
package com.journaldev.designpatterns;

public class ASingleton {

        private static volatile ASingleton instance;
```

```
                                 :ex = new Object();

        private ASingleton() {
        }

        public static ASingleton getInstance() {
                ASingleton result = instance;
                if (result == null) {
                        synchronized (mutex) {
                                result = instance;
                                if (result == null)
                                        instance = result = new ASingleton();
                        }
                }
                return result;
        }

}
```

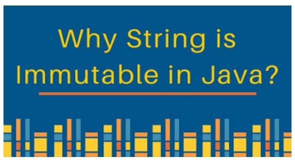
Local variable `result` seems unnecessary. But it's there to improve performance of our code. In cases where instance is already initialized (most of the time), the volatile field is only accessed once (due to "return result;" instead of "return instance;"). This can improve the method's overall performance by as much as 25 percent.

If you think there are better ways to achieve this or the thread safety is compromised in the above implementation, please comment and share with all of us.

Update: String is not a very good candidate to be used in synchronization, so I have updated it with Object, learn more about synchronization and thread safety in java.

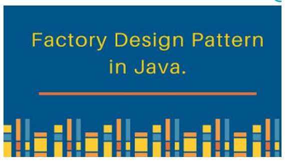


How to Create immutable Class in java

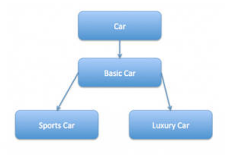


Why String is immutable or final in Java

Java Singleton Design Pattern Example Best Practices

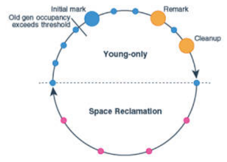


Factory Design Pattern ir Java



Decorator Design Pattern in Java Example

Thread Safety in Java



Garbage Collection in Java

Multithr

NEXT »
Java Singleton Design Pattern Best Practices with Examples

About Pankaj
If you have come this far, it means that you liked what you are reading. Why not reach little more and connect with me directly on **Google Plus**, **Facebook** or **Twitter**. I would love to hear your thoughts and opinions on my articles directly.

Recently I started creating video tutorials too, so do check out my videos on **Youtube**.

FILED UNDER: DESIGN PATTERNS

Comments

Harshit Joshi says
SEPTEMBER 5, 2018 AT 4:33 AM
Hey Nice Solution but i think we can also optimize is by not creating object lock instead of it we can use class lock
example
Copied!

```
package com.journaldev.designpatterns;
public class ASingleton {
private static volatile ASingleton instance;
private ASingleton() {
}
public static ASingleton getInstance() {
ASingleton result = instance;
if (result == null) {
synchronized (ASingleton.class) {
result = instance;
if (result == null)
instance = result = new ASingleton();
}
}
return result;
```

,

I am not sure Please guide me as i am fresher
Reply

Datta says
SEPTEMBER 2, 2018 AT 12:31 AM
This seems ok..create instance on class loading and later return same instance..No synchronization required.
public class Singleton {
private static Singleton singltonInstance = new Singleton();
private Singleton() {
}
public static Singleton getInstance() {
return singltonInstance;
}
@Override
protected Object clone() throws CloneNotSupportedException {
throw new CloneNotSupportedException();
}
}
Reply

jatin says
JUNE 25, 2018 AT 10:40 PM
You don't talk about enum. Enum objects are best suited for singleton classes. It covers both concurrency and serialization issues we face in our custom singleton classes.
Reply

ashish gupta says
MAY 2, 2018 AT 12:19 AM
public class TestSingleton {
public static void main(String[] args) throws NoSuchMethodException, SecurityException, InstantiationException,
IllegalAccessException, IllegalArgumentException, InvocationTargetException {
ASingleton singleton1 = ASingleton.getInstance();
System.out.println(singleton1);
ASingleton singleton2 = ASingleton.getInstance();
System.out.println(singleton2);
Constructor c = ASingleton.class.getDeclaredConstructor((Class[]) null);
c.setAccessible(true);
System.out.println(c);

```
(Object[]) null);
if (singleton1 == singleton2) {
System.out.println("Variable 1 and 2 referes same instance");
} else {
System.out.println("Variable 1 and 2 referes different instances");
}
if (singleton1 == singleton3) {
System.out.println("Variable 1 and 3 referes same instance");
} else {
System.out.println("Variable 1 and 3 referes different instances");
}
}
}
```

Hi Pankaj,
NIce article,
but the solution you provided , fails the above testcase.
Thanks .
Reply

Gandalf says
JULY 20, 2018 AT 12:13 PM

The test case you have mentioned uses Java Reflection API which is not used by the same code base for most cases. Imagine an app that wants to keep the constructor private and still uses reflection to access the private constructor at other place, if that's the case, then why make the constructor private in the first place? So generally speaking the developer of an app that uses private members won't break his own rules to use Java reflection to access private constructors.
Now, let's take the case of third party apps accessing a common library that uses singletons. Now as a developer using the common library, you wanna use it in the proper way rather than using reflection so that your app performs better and your threads do not accidentally create multiple instances of the limited resource.
The purpose of data encapsulation and other Object oriented programming methodology is for encouraging loose coupling and other best practices, these rules are not related to security to prevent something from happening. Its your code, you can do anything with it. These principles are for us developers to follow to ensure our app is maintainable in the long run. If we purposefully decide to not use these principles, then no one is there to restrict us.
Now in case, let's say we are suspicious about our own developers going out of the way and using reflection which is a rare case but still, then the solution is to run Java app with certain security policies that restrict the reflection capabilities. Read more here – https://stackoverflow.com/questions/7566626/how-to-restrict-developers-to-use-reflection-to-access-private-methods-and-const
Reply

Parvesh Kumar says
APRIL 29, 2018 AT 6:24 AM

't will create new instance always.

AB says
APRIL 8, 2018 AT 9:41 PM
C++ implementation using a bool var instead (possibly more efficent)
static ConnectionManager *_currentInstance = nullptr;
static bool instanceAvailable = false;
static std::mutex singletonMutex;
ConnectionManager* ConnectionManager::getInstance()
{
// thread safe implementation of SINGLETON
if (!instanceAvailable)
{
instanceAvailable = true;
std::unique_lock lck(singletonMutex);
if (!_currentInstance) {
_currentInstance = new ConnectionManager;
}
}
return _currentInstance;
}
Reply

Diego says
MARCH 27, 2018 AT 8:47 AM
Thanks Pankaj, great article.
One question: does adding synchronization at the getInstance level (regardless of the approach) remove the need to synchronize access to other resources in the instance (member variables, class variables and methods)?
thanks again
Reply

Pankaj says
MARCH 27, 2018 AT 9:12 AM
No it won't. When you add synchronization to getInstance method, it is to make sure that you have only one instance of the Singleton class. So if two threads are calling getInstance method, both will get reference to the same object, hence any change in class variables or instance variable will be reflected in other thread as well. So you will have to take care of synchronization for these variables too. Ideally, you should avoid any variables in Singleton class.
Reply

MARCH 8, 2018 AT 11:35 PM

we can use "synchronized (ASingleton.class) " instead of below piece of code

```
synchronized (mutex) {
result = instance;
if (result == null)
instance = result = new ASingleton();
}
```

Reply

Raviraj_bk says

MARCH 8, 2018 AT 11:35 PM

is it right?

Reply

Pankaj says

MARCH 9, 2018 AT 12:29 AM

Synchronizing class will lock it's object too, that's why we should always create a temporary variable for synchronizing block of code.

Reply

Edo says

FEBRUARY 22, 2018 AT 3:29 AM

I think that this kind of implementation doesn't avoid the singleton creation through reflection.

My favourite implementation is something like this:

```
public class MySingleton {
private static class Loader {
private static MySingleton INSTANCE = new MySingleton();
}
private MySingleton(){
if(Loader.INSTANCE!=null){
throw new InstantiationError( "Creating of this object is not allowed." );
}
}
public static MySingleton getInstance(){
return MySingleton.Loader.INSTANCE;
}
}
```

What do you think?

Thanks

Edo

Marton Korosi says
NOVEMBER 17, 2017 AT 9:09 AM
I think the 3rd approach is incorrect.
instance variable should be volatile! like this:
private static volatile ASingleton instance= null;
——–
If instance variable is not volatile, JVM can reorder
new ASingleton()
and
instance=
operations which may result in an instance variable which will point to uninitialized memory!
refer to: https://en.wikipedia.org/wiki/Double-checked_locking
Reply

Pankaj says
NOVEMBER 17, 2017 AT 9:48 PM
Thanks for pointing it out, it was written long back when volatile was broken in java 1.4. I have updated the code with correct approach now.
Reply

Noy says
JUNE 30, 2017 AT 3:54 PM
Hi. I have two issues with the selection of the third option. One is that by creating a mutex static object you contradict the fact that you keep a static option that you may never use(mutex).
Second is that i would change the code to contain the if condition one time unsynchronized and obe time in an inner if when it is synchronized on mutex.
Reply

Aarati says
DECEMBER 15, 2017 AT 9:18 AM
As "instance" field is static. So will it make sense to synchronize on (synchronized (ASingleton.class)
Reply

jubi max says
APRIL 24, 2017 AT 3:21 AM
Thank you very much

Prachi says
APRIL 22, 2017 AT 6:26 PM
I replied the same answer to a interviewer but she asked me that object is not created yet since getInstance() is static method who will get object lock in synchronization.
Thanks,
Prachi
Reply

Nayanava says
AUGUST 20, 2017 AT 2:50 AM
by object do you mean the mutex object??
Reply

Raviraj_bk says
MARCH 8, 2018 AT 11:38 PM
Hi,
Since mutex is static varible it will be initialized during class loading time, and then synchnozization happens on the lock of mutex object.
Reply

Ravi says
FEBRUARY 2, 2017 AT 3:25 AM
Can we not use the volatile() instead of using synchroniation?
Reply

Hamid Khan says
DECEMBER 23, 2016 AT 3:15 AM
You haven't made any comment regarding why not making 'instance' volatile.. any thoughts?
Reply

Satya says
MAY 23, 2016 AT 2:06 AM
best to make getInstance() synchronized
Reply

SEPTEMBER 15, 2016 AT 11:39 AM

If the method is synchronized, where huge threads are calling that method, every thread will have to wait when it's being used by other thread. Think is it really required when the object is already created and != null? So making method synchronization is not a good idea.

Reply

vijay says

JANUARY 19, 2017 AT 12:13 AM

prevent from cloning is not mentioned

Reply

Ganesh says

FEBRUARY 8, 2016 AT 10:16 PM

That I really appreciate for this article . I learned good stuff today, and also I red some where below code snipped is very good code for singleton, can you compare with your code in terms pros and cons.

```
package com.journaldev.designpatterns;
public class ASingleton{
private ASingleton(){
}
private static class ASingletonInnerClass{
private static final ASingleton ASINGLETON= new ASingleton();
}
public static ASingleton getInstance(){
return ASingletonInnerClass.ASINGLETON;
}
}
```

Reply

DIVYA PALIWAL says

FEBRUARY 3, 2016 AT 10:21 AM

Hi,
Great article!
I am trying t understand thread safety in Singleton Pttern.
Can you please provide and example code where different threads are trying to implement Singleton pattern.
Thanks,
Divya

Reply

DECEMBER 4, 2015 AT 2:37 AM
Very nice article. But if loop wording is not correct, please change it to if condition.
Reply

Amey Jadiye says
MAY 19, 2015 AT 10:46 AM
I think making instance volatile make much difference than approach given in post
Reply

Archna Sharma says
FEBRUARY 18, 2015 AT 9:44 PM
In your third approach, although it checks the value of instance once again within the synchronized block but the JIT compiler can rearrange the bytecode in a way that the reference to instance is set before the constructor has finished its execution.
This means the method getInstance() returns an object that may not have been initialized completely.
I think, the keyword volatile can be used for the instance variable. Variables that are marked as volatile get only visible to other threads once the constructor of the object has finished its execution completely.
Reply

Naveen J says
JULY 25, 2014 AT 8:40 AM
String is not a very good candidate to be used in synchronization, so I have updated it with Object, learn more about synchronization and thread safety in java
Why string is not good candidate……… Since its immutable its a good candidate to use in synchronization block right.
Reply

Pankaj says
JULY 25, 2014 AT 10:05 PM
We should not use any object that is maintained in a constant pool, for example String should not be used for synchronization because if any other code is also locking on same String, it will try to acquire lock on the same reference object from String pool and even though both the codes are unrelated, they will lock each other.
Reply

Asanka says

JUNE 19, 2014 AT 10:21 AM

I believe this is the best way, it doesn't use any synchronization at all, provides better performance too.
http://en.wikipedia.org/wiki/Initialization-on-demand_holder_idiom
Reply

Pankaj says
JUNE 19, 2014 AT 2:21 PM
Singleton Pattern Approaches – read this article to learn about different ways and their pros-cons.
Reply

Rishi Dev Gupta says
JULY 19, 2013 AT 11:47 AM
there is a good way to implement the Singletons, that will look after all the issue and with lesser code
public enum InputValidatorImpl {
instance;
// add some method
}
Reply

Pankaj says
JULY 20, 2013 AT 5:10 AM
there are several ways to implement singleton pattern and Enum is one of the way but we can't achieve lazy initialization with this. Read more at https://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-with-examples#enum-singleton
Reply

Hesham says
MAY 15, 2013 AT 1:04 PM
double check lock is not thread safe in java this issue listed by PDM tool (block synchronizing)
Reply

Anonymous says
JANUARY 28, 2011 AT 6:27 AM
',~ I am really thankful to this topic because it really gives useful information :-`
Reply

JANUARY 4, 2011 AT 2:06 PM

you can avoid your extra if condition if you create instance described below,
Once we declare static, it will refer the same object all the time
package com.journaldev.designpatterns;
public class ASingleton{
private static ASingleton instance= new ASingleton();
private ASingleton(){ }
public static synchronized ASingleton getInstance(){
return instance;
}
}

Reply

Pankaj says

JANUARY 8, 2011 AT 4:38 AM

Here you are creating the instance at the time of class loading… what if we are expecting some parameters like DB Connection URL, User, Password to be passed in the getInstance method so that it will be generic.

Reply

anon says

DECEMBER 28, 2010 AT 3:58 AM

http://en.wikipedia.org/wiki/Double-checked_locking#Usage_in_Java

Reply

Erik van Oosten says

DECEMBER 27, 2010 AT 1:39 PM

Example 3 is the traditional double check idiom for lazy initialization. The double check is badly broken in java before version 5. The example you have here is broken also because instance is not declared volatile.
The best way is to extract the singleton code to a separate class which is guaranteed to be loaded only when the referring class is instantiated.
For more information see item 71 in "Effective Java" (2nd edition) by Joshua Bloch.
But you'd better avoid singletons completely.

Reply

gaurav says

JANUARY 23, 2014 AT 1:14 AM

invoke Bill Pugh's singleton pattern

ization_on_demand_holder_idiom

Comment Policy: Please submit comments to add value to the post. Comments like "Thank You" and "Awesome Post" will be not published. If you want to post code then wrap them inside <pre> tags. For example **<pre>class Foo { }</pre>**.
If you want to post XML content, then please escape < with &lt; and > with &gt; otherwise they will not be shown properly.

Leave a Reply

Your email address will not be published. Required fields are marked *

Comment

Name *

Email *

Instantly Search Tutorials...

DESIGN PATTERNS TUTORIAL

RECOMMENDED TUTORIALS

Java Tutorials

- Java IO
- Java Regular Expressions
- Multithreading in Java
- Java Logging
- Java Annotations
- Java XML
- Collections in Java
- Java Generics
- Exception Handling in Java
- Java Reflection
- Java Design Patterns
- JDBC Tutorial

Java EE Tutorials

- Servlet JSP Tutorial
- Struts2 Tutorial
- Spring Tutorial
- Hibernate Tutorial
- Primefaces Tutorial
- Apache Axis 2
- JAX-RS
- Memcached Tutorial