**How to make the perfect Singleton?**

[Keval Patel](https://medium.com/@kevalpatel2106?source=post_page-----de6b951dfdb0--------------------------------)

[Keval Patel](https://medium.com/@kevalpatel2106?source=post_page-----de6b951dfdb0--------------------------------)

[Mar 27, 2017·6 min read](https://medium.com/@kevalpatel2106/how-to-make-the-perfect-singleton-de6b951dfdb0?source=post_page-----de6b951dfdb0--------------------------------)

Design Patterns are popular among software developers. A design pattern is a well described solution to a common software problem. The Singleton is one of the [Creational Design Patterns](https://en.wikipedia.org/wiki/Creational_pattern) in Java.

**What is the purpose of Singleton?**

The purpose of the Singleton class is to control object creation, limiting the number of objects to only one. The singleton allows only one entry point to create the new instance of the class.

Since there is only one Singleton instance, any instance fields of a Singleton will occur only once per class, just like static fields. Singletons are often useful where you have to control the resources, such as database connections or sockets.

It seems to be a simple design pattern but when it comes to implementation, it comes with a lot of implementation concerns. The implementation of Singleton pattern has always been a controversial topic among developers. Here, you are going to discuss how to create a Singleton class that fulfills its purpose :

*Restrict the instantiation of a class and ensures that only one instance of the class exists in the java virtual machine.*

Let’s create Singleton class in java and test it in different conditions.

Create Singleton class

To implement the Singleton class, the simplest way is to make the constructor of the class as private. There are two approaches for the initialization.

**1. Eager initialization:**

In eager initialization, the instance of Singleton Class is created at the time of class loading, this is the easiest method to create a Singleton class.

By making the [constructor](https://en.wikipedia.org/wiki/Constructor_%28object-oriented_programming%29) as private you are not allowing other class to create a new instance of the class you want to create the Singleton. Instead, you are creating one public static method (commonly name as for*getInstance()*) to provide the single entry point to create the new instance of the class.

This approach has one drawback. Here instance is created even though client application might not be using it. This might be a considerable issue if your  
Singleton class in creating a database connection or creating a socket. This may cause the memory leak problem. The solution is to create the new instance of the class, when needed. This can be achieved by Lazy Initialization method.

2. Lazy initialization:

Opposite to Eager initialization, here you are going to initialize new instance of the class in *getInstance()*method it self.This method will check if there is any instance of that class is already created? If yes, then our method (*getInstance()*) will return that old instance and if not then it creates a new instance of the singleton class in JVM and returns that instance. This approach is called as Lazy initialization.

We all know that in Java if the two objects are same then, their hash key have to be equal. Let’s test that. If the above Singleton is correctly implemented than below code should return the same hash key.

Below is the output log with the hash code of both the instances.

Image for post

Image for post

Both have the same hash.

You can see that both the instances are having the same hash code. So, that means above code will make the perfect Singleton. Right???? No.

**Make Singleton reflection proof**

In above Singleton class, by using reflection you can create more than one instance. If you don’t know what is the [Java Reflection API](https://docs.oracle.com/javase/tutorial/reflect/) is, Java Reflection is a process of examining or modifying the run-time behavior of a class at run time.

You can make the new instance of the Singleton class by changing the constructor visibility as public in run-time and create new instance using that constructor. Run the below code and see it our Singleton class survives?

Here is the output of hash codes of both instances.

Image for post

Image for post

Testing reflection

Both the instances have a different hash code. That clearly indicates that Singleton class failed this test.

**Solution:**

To prevent Singleton failure while due to reflection you have to throw a run-time exception in constructor, if the constructor is already initialized and some class to initialize it again. Let’s update *SingletonClass.java*.

Make Singleton thread safe

If two threads try to initialize the Singleton class at almost the same time, what happens? Let’s test below code in which two threads are created almost simultaneously and they are calling *getInstance()*.

If you run this code many times, you will see sometimes both the threads creates different instances.

Image for post

Image for post

That means your Singleton class is not Thread safe. Both the threads calls out *getInstance()* method at the same time, the *sSoleInstance == null* condition will return for both the thread. So, two different instances of the same class will be created. That will break the singleton principle.

Solution:

1. Make *getInstance()*synchronized:

Let’s make getInstance() method synchronized.

As you made your *getInstance()* class synchronized the second thread will have to wait until the *getInstance()* method is completed for the first thread. This way we can achieve thread safety.

But, there are some cons of using this approach:

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

2. Double check locking method:

You can overcome this issue if you use **Double check locking** method to create the Singleton.

In this, you will make the Singleton class in the synchronized block if the instance is null. So, the synchronized block will be executed only when the *sSoleInstance*is null and prevent unnecessary synchronization once the instance variable is initialized.

3. Use volatile keyword:

On the surface, this method looks perfect, as you only need to pay price for synchronized block one time, but it still was broken, until you make *sSoleInstance*variable [*volatile*](http://www.javamex.com/tutorials/synchronization_volatile.shtml).

Without volatile modifier, it’s possible for another thread in Java to see half initialized state of *sSoleInstance* variable, but with volatile variable guaranteeing happens-before relationship, all the write will happen on volatile *sSoleInstance*before any read of *sSoleInstance*variable.

Now, above Singleton class is thread safe. Making Singleton thread safe is especially required in multi-threaded application environment like in Android Applications.

**Make Singleton safe from Serialization**

Sometimes in distributed systems, you need to implement [*Serializable*](http://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html)interface in Singleton class. By doing that you can store its state in the file system and retrieve it at later point of time.

Let’s test singleton class whether it maintains single instance after serializable and deserializable operations?

Image for post

Image for post

You can see that the hash code of both the instances are different. That is clearly violates singleton principle. The problem with above serialized singleton class is that whenever we deserialize it, it will create a new instance of the class.

To prevent creation of another instance you have to provide the implementation of *readResolve()* method.*readResolve(*) replaces the object read from the stream. This ensures that nobody can create another instance by serializing and deserializing the singleton.

**Conclusion:**

At end of the article, you can make your class a Singleton class that is thread, reflection and serialization safe. This Singleton is still not the perfect Singleton. You can violate the Singleton principle by creating more than one instance of the Singleton class by using cloning or using multiple class loaders. But for the most of the applications, above implementation of Singleton will work perfectly.