

# Java Singleton Pattern Explained



Last Updated: January 25,  
2022



By: Lokesh  
Gupta



Creational  
Patterns



Design Patterns,  
Singleton

**Singleton pattern** is a design solution where an application wants to have one and only one instance of any class, in all possible scenarios without any exceptional condition. It has been debated long enough in java community regarding possible approaches to make any class singleton. Still, you will find people not satisfied with any solution you give. They cannot be overruled either. In this post, we will discuss some good approaches and will work towards our best possible effort.

## Table of Contents:

1. [Singleton with eager initialization](#)
2. [Singleton with lazy initialization](#)
3. [Singleton with static block initialization](#)
4. [Singleton with bill pugh solution](#)
5. [Singleton using Enum](#)
6. [Add readResolve\(\) to singleton objects](#)
7. [Add serialVersionUID to singleton objects](#)
8. [Conclusion](#)

Singleton term is derived from its [mathematical counterpart](#). It wants us, as said above, to have only one instance per context. In Java, one instance per JVM.

Lets see the possible solutions to create singleton objects in Java.

## 1. Singleton with eager initialization

This is a design pattern where an instance of a class is created much before it is actually required. Mostly it is done on system startup. In an eager initialization singleton pattern, the singleton instance is created irrespective of whether any other class actually asked for its instance or not.

```
public class EagerSingleton {  
    private static volatile EagerSingleton instance = new EagerSingleton();  
  
    // private constructor  
    private EagerSingleton() {  
    }  
  
    public static EagerSingleton getInstance() {  
        return instance;  
    }  
}
```

The above method works fine, but it has one drawback. The instance is created irrespective of it is required in runtime or not. If this instance is not a big object and you can live with it being unused, this is the best approach.

Let's solve the above problem in the next method.

## 2. Singleton with lazy initialization

In computer programming, [lazy initialization](#) is the tactic of delaying the creation of an object, the calculation of a value, or some other expensive process, until the first time it is needed. In a singleton pattern, it restricts the creation of the instance until it is requested for first time. Lets see this in code:

```
public final class LazySingleton {  
    private static volatile LazySingleton instance = null;  
  
    // private constructor  
    private LazySingleton() {  
    }  
}
```

```

public static LazySingleton getInstance() {
    if (instance == null) {
        synchronized (LazySingleton.class) {
            instance = new LazySingleton();
        }
    }
    return instance;
}
}

```

On the first invocation, the above method will check if the instance is already created using the instance variable. If there is no instance i.e. the instance is null, it will create an instance and will return its reference. If the instance is already created, it will simply return the reference of the instance.

But, this method also has its own drawbacks. Let's see how. Suppose there are two threads T1 and T2. Both come to create the instance and check if “instance==null”. Now both threads have identified instance variable as null thus they both assume they must create an instance. They sequentially go into a synchronized block and create the instances. In the end, we have two instances in our application.

This error can be solved using [double-checked locking](#). This principle tells us to recheck the instance variable again in a synchronized block as given below:

```

public class LazySingleton {
    private static volatile LazySingleton instance = null;

    // private constructor
    private LazySingleton() {
    }

    public static LazySingleton getInstance() {
        if (instance == null) {
            synchronized (LazySingleton.class) {
                // Double check
                if (instance == null) {
                    instance = new LazySingleton();
                }
            }
        }
        return instance;
    }
}

```

```
}
```

Above code is the correct implementation of the singleton pattern.

Please be sure to use "**volatile**" keyword with instance variable otherwise you can run into an out of order write error scenario, where reference of an instance is returned before actually the object is constructed i.e. JVM has only allocated the memory and constructor code is still not executed. In this case, your other thread, which refers to the uninitialized object may throw null pointer exception and can even crash the whole application.

### 3. Singleton with static block initialization

If you have an idea of the class loading sequence, you can use the fact that static blocks are executed during the loading of a class, even before the constructor is called. We can use this feature in our singleton pattern like this:

```
public class StaticBlockSingleton {
    private static final StaticBlockSingleton INSTANCE;

    static {
        try {
            INSTANCE = new StaticBlockSingleton();
        } catch (Exception e) {
            throw new RuntimeException("Uffff, i was not expecting this!", e);
        }
    }

    public static StaticBlockSingleton getInstance() {
        return INSTANCE;
    }

    private StaticBlockSingleton() {
        // ...
    }
}
```

The above code has one drawback. Suppose there are 5 static fields in a class and the application code needs to access only 2 or 3, for which instance creation is not required at all. So, if we use this static initialization, we will have one instance created though it is required or not.

The next section will overcome this problem.

## 4. Singleton with bill pugh solution

Bill Pugh was main force behind the [java memory model](#) changes. His principle "Initialization-on-demand holder idiom" also uses the static block idea, but in a different way. It suggest to use static inner class.

```
public class BillPughSingleton {  
    private BillPughSingleton() {  
    }  
  
    private static class LazyHolder {  
        private static final BillPughSingleton INSTANCE = new BillPughSingleton();  
    }  
  
    public static BillPughSingleton getInstance() {  
        return LazyHolder.INSTANCE;  
    }  
}
```

As you can see, until we need an instance, the LazyHolder class will not be initialized until required and you can still use other static members of BillPughSingleton class. *This is the solution, i will recommend to use. I have used it in my all projects.*

## 5. Singleton using Enum

This type of implementation employs the use of enum. [Enum](#), as written in the java docs, provided implicit support for thread safety and only one instance is guaranteed. **Java enum singleton** is also a good way to have singleton with minimal effort.

```
public enum EnumSingleton {  
    INSTANCE;  
    public void someMethod(String param) {  
        // some class member  
    }  
}
```

## 6. Add readResolve() to Singleton Objects

By now you must have made your decision about how you would like to implement your singleton. Now let's see other problems that may arise even in job interviews.

Let's say your application is distributed and it frequently serializes objects into the file system, only to read them later when required. Please note that de-serialization always creates a new instance. Let's understand using an example:

Our singleton class is:

```
public class DemoSingleton implements Serializable {  
    private volatile static DemoSingleton instance = null;  
  
    public static DemoSingleton getInstance() {  
        if (instance == null) {  
            instance = new DemoSingleton();  
        }  
        return instance;  
    }  
  
    private int i = 10;  
  
    public int getI() {  
        return i;  
    }  
  
    public void setI(int i) {  
        this.i = i;  
    }  
}
```

```

    }
}

```

Let's serialize this class and de-serialize it after making some changes:

```

public class SerializationTest {
    static DemoSingleton instanceOne = DemoSingleton.getInstance();

    public static void main(String[] args) {
        try {
            // Serialize to a file
            ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream(
                "filename.ser"));
            out.writeObject(instanceOne);
            out.close();

            instanceOne.setI(20);

            // Serialize to a file
            ObjectInput in = new ObjectInputStream(new FileInputStream(
                "filename.ser"));
            DemoSingleton instanceTwo = (DemoSingleton) in.readObject();
            in.close();

            System.out.println(instanceOne.getI());
            System.out.println(instanceTwo.getI());

        } catch (IOException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}

```

Output:

```

20
10

```

Unfortunately, both variables have different values of the variable "i". Clearly, there are two instances of our class. So, again we are in the same problem of multiple instances in our application.

To solve this issue, we need to include a `readResolve()` method in our `DemoSingleton` class. This method will be invoked when you will de-serialize the

object. Inside of this method, you must return the existing instance to ensure a single instance application wide.

```
public class DemoSingleton implements Serializable {
    private volatile static DemoSingleton instance = null;

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

    protected Object readResolve() {
        return instance;
    }

    private int i = 10;

    public int getI() {
        return i;
    }

    public void setI(int i) {
        this.i = i;
    }
}
```

Now when you execute the class `SerializationTest`, it will give you correct output.

```
20
20
```

## 7. Add `serialVersionUID` to singleton objects

So far so good. Untill now, we have solved both of the problems of synchronization and serialization. Now, we are just one step away from a correct and complete implementation. The only missing part is a serial version id.



This is required in cases where your class structure changes between serialization and deserialization. A changed class structure will cause the JVM to give an exception in the de-serializing process.

```
java.io.InvalidClassException: singleton.DemoSingleton; local class incompatible:
at java.io.ObjectStreamClass.initNonProxy(Unknown Source)
at java.io.ObjectInputStream.readNonProxyDesc(Unknown Source)
at java.io.ObjectInputStream.readClassDesc(Unknown Source)
at java.io.ObjectInputStream.readOrdinaryObject(Unknown Source)
at java.io.ObjectInputStream.readObject0(Unknown Source)
at java.io.ObjectInputStream.readObject(Unknown Source)
at singleton.SerializationTest.main(SerializationTest.java:24)
```

This problem can be solved only by adding a unique serial version id to the class. It will prevent the compiler from throwing the exception by telling it that both classes are same, and will load the available instance variables only.

## 8. Conclusion

After having discussed so many possible approaches and other possible error cases, I will recommend to you the code template below, to design your singleton class which shall ensure only one instance of a class in the whole application in all above discussed scenarios.

```
public class DemoSingleton implements Serializable {
    private static final long serialVersionUID = 1L;

    private DemoSingleton() {
        // private constructor
    }

    private static class DemoSingletonHolder {
        public static final DemoSingleton INSTANCE = new DemoSingleton();
    }

    public static DemoSingleton getInstance() {
        return DemoSingletonHolder.INSTANCE;
    }
}
```

```
protected Object readResolve() {  
    return getInstance();  
}  
}
```

I hope this post has enough information to help make you understand the most common approaches for the **singleton pattern** and **singleton best practices**. Let me know your thoughts.

Happy Learning !!

**Realtime Singleton Examples** – I just thought to add some examples which can be referred for further study and mention in interviews:

- [java.awt.Desktop#getDesktop\(\)](#)
- [java.lang.Runtime#getRuntime\(\)](#)

## Was this post helpful?

Let us know if you liked the post. That's the only way we can improve.

Yes

No

## Recommended Reading:

1. [Java Factory Pattern Explained](#)
2. [Abstract Factory Pattern Explained](#)
3. [Prototype design pattern in Java](#)
4. [Builder Design Pattern](#)
5. [Decorator Design Pattern in Java](#)

6. [Adapter Design Pattern in Java](#)

7. [Composite Design Pattern](#)

8. [Strategy Design Pattern](#)

9. [Memento Design Pattern](#)

10. [Observer Design Pattern](#)

## Join 7000+ Awesome Developers

Get the latest updates from industry, awesome resources, blog updates and much more.

**Email Address**

**Subscribe**

*\* We do not spam !!*

## 105 thoughts on “Java Singleton Pattern Explained”

**Thangamma**

December 16, 2019 at 12:21 pm

Hi Lokesh,

I would want to know the practical usage of readResolve() method. When i serialize an object , while deserializing i expect to get the content that was serialized. With the ReadResolve() method, if i return the existing singleton instance i end up returning the existing instance and i will not get the values that were actually serialized.

I am not able to get the intent of ReadResolve() method

[Reply](#)

**purna padhi**

May 21, 2019 at 4:54 pm

Can you please explain how to block this as well.

[Reply](#)**ajinkya rasal**[January 21, 2019 at 1:21 pm](#)

Will the Bill Pugh solution work in case of multiple threads? As mentioned in the Drawback of singleton with Lazy case.

What will happen if multiple threads try to get the instance of the singleton class

[Reply](#)**Brajesh**[March 4, 2019 at 11:19 pm](#)

Yes it does. JVM will ensure sequential initialization of the static classes.

[https://en.wikipedia.org/wiki/Initialization-on-demand\\_holder\\_idiom](https://en.wikipedia.org/wiki/Initialization-on-demand_holder_idiom)

[Reply](#)**Ram Srinivasan**[October 24, 2017 at 3:10 am](#)

Hello Lokesh,

Why your recommended versions, removed the line  
private volatile static DemoSingleton instance = null;

I am confused, you introduced it and then removed in your two recommended versions. why?

Regards,

Ram Srinivasan

[Reply](#)

**abhinav**

[February 5, 2017 at 12:31 pm](#)

Hi Lokesh,

I was asked in an Interview that you are creating private constructor so that you can not access the class from outside, then why we are declaring the object reference variable(private static volatile EagerSingleton instance) as private.

[Reply](#)

**amit kumar**

[October 10, 2016 at 1:22 am](#)

very helpful, thanks for clearing all the doubts of singleton pattern, have never seen such a good blog.

thanks once again

[Reply](#)**sumit**[August 31, 2016 at 7:51 pm](#)

I understood that deserialization creates another instance of Singleton. But Why ? does it calls private constructor internally ?

On what instance does readResolve method is invoked when the deserialized instance is a different instance altogether ?

[Reply](#)**Lokesh Gupta**[September 1, 2016 at 1:21 pm](#)

Please read : [How Deserialization Process Happen in Java?](#)

[Reply](#)**shramik**[August 19, 2016 at 12:32 pm](#)

```
public class BillPughSingleton {  
    private BillPughSingleton() {  
    }  
  
    private static class LazyHolder {
```

```
private static final BillPughSingleton INSTANCE = new BillPughSing
}

public static BillPughSingleton getInstance() {
    return LazyHolder.INSTANCE;
}
}
```

As per you recommend above code template to design singleton class.  
But how Thread safety achieve in above code ??

[Reply](#)

**Lokesh Gupta**

August 19, 2016 at 2:21 pm

It's already threadsafe because java static field/class initialization is thread safe – at JVM level. Static initialization is performed once per class-loader and JVM ensures the single copy of static fields. So even if two threads access above code, only one instance of class will be created by JVM.

[Reply](#)

**DArshit PAtel**

December 16, 2015 at 1:03 pm

Hi Lokesh ! I have read your warning to use volatile keyword but i can't understand it properly. Will you please elaborate it and explain it with any example ?



[Reply](#)**Sampat Mali**

November 28, 2015 at 5:20 pm

Hi Lokesh sir,

Nice post i am very good clarity about Singleton class but i am confused readResolve() method please explain me which class declare the readResolve() method and who will call?

[Reply](#)**Lokesh Gupta**

November 28, 2015 at 5:40 pm

The class which you are making singleton will declare readResolve() method. You do not need to call it in any class, JVM internally uses this method.

[Reply](#)**Sampat Mali**

November 28, 2015 at 5:44 pm

Hi Lokesh,

I am leave here my singleton code please say me is all angle correct or not...  
package com.sampat.stp;

```
import java.io.Serializable;

import com.sampat.commans.ComminsUtils;

public class PrinterUtil extends ComminsUtils {

    /**
     *
     */
    private static final long serialVersionUID = 1L;
    // create an object of PrinterUtil
    // private static PrinterUtil instance=new PrinterUtil();
    private static boolean isInstantiated = false;
    private static PrinterUtil instance;

    /**
     * static{ instance=new PrinterUtil(); }
     */

    // make the constructor private so that this class cannot be
    // instantiated
    private PrinterUtil() throws InstantiationException {

        if (isInstantiated == true) {
            throw new InstantiationException();
        } else {
            isInstantiated = true;
        }
        System.out.println("PrintUtil:0-param constructor");
        // no task
    }

    // Get the only object available
    public static PrinterUtil getInstance() throws InstantiationException {
        if (instance == null) {
```

```
synchronized (PrinterUtil.class) {  
    // Double Check  
    if (instance == null) {  
        try {  
            Thread.sleep(1000);  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
  
        instance = new PrinterUtil();  
    }  
}  
  
return instance;  
}  
  
// Instead of the object we're on, return the class variable singleton  
public Object readResolve() {  
    return instance;  
}  
  
@Override  
public Object clone() throws CloneNotSupportedException {  
  
    return new CloneNotSupportedException();  
}  
}
```

[Reply](#)

**Mayur**

August 7, 2015 at 11:42 am

In final example, why dont you declare variable as volatile. How your final solution will work in multithreaded enviornment

[Reply](#)

**Lokesh Gupta**

August 7, 2015 at 12:48 pm

Which example and what problem? Please elaborate.

[Reply](#)

**sivadurga**

March 12, 2015 at 1:08 pm

Hi lokesh ,  
can you plese explain why you are going to synchronize the object in lazy initialization case

[Reply](#)

**Lokesh Gupta**

March 15, 2015 at 2:44 am

It's already explained in respective section.

[Reply](#)**prabir**[October 10, 2014 at 6:24 am](#)

Hi

Every thing you have written in this post is excellent .  
I have some doubts. Please clarify it.

Singleton means only one instance of class with in a jvm. There is two web application wants to use same singletone class.But whenever , web application uses this singleton class it creates one instance for an application means per application one instnace .But my web application runs under jvm with same server

[Reply](#)**Alps**[October 4, 2014 at 11:43 am](#)

Hi Lokesh,

Thanks for the nice explanation. I have one query- how can we ensure or use singelton behaviour in clustered environment?

[Reply](#)

**Lokesh Gupta**

October 5, 2014 at 6:07 am

Singleton is "one instance per JVM", so each node will have its own copy of singleton.

[Reply](#)

**anzar ansari**

February 5, 2015 at 8:30 pm

hi Lokesh,

Thanks for giving such a good content, but I think in definition of Singleton there is little correction so it will become perfect "one instance per Java Virtual Machine"

according to me

"one instance of a class is created within hierarchy of ClassLoader or (within the scope of same application) "

[Reply](#)

**Ram**

February 21, 2015 at 12:11 am

I could not understand why you went further than the "Bill pugh solution" is it because "Bill pugh solution" is wrong or you wanted to show alternative to "Bill pugh solution"?

[Reply](#)

**Lokesh Gupta**

[February 21, 2015 at 1:55 am](#)

Solution is correct but enums are best where they may fit in. It's about knowing your all options only.

[Reply](#)

**Madhav**

[August 15, 2014 at 10:37 am](#)

I was asked in an interview to Create singleton class using public constructor. is it possible?if yes, could you please provide the details

[Reply](#)

**saurabh moghe**

[April 1, 2015 at 9:01 am](#)

Hi Madhav , yes it is possible. please go through following code

```
public class Singleton {  
  
    private static Singleton instance;  
    public static synchronized Singleton getInstance() {  
        return (instance != null) ? instance : new Singleton();  
    }  
    public Singleton() {  
        System.out.println("in constructor");  
        synchronized (Singleton.class) {  
            if (instance != null) {  
                throw new IllegalStateException();  
            }  
            instance = this;  
        }  
    }  
}
```

[Reply](#)**ravi**

July 24, 2014 at 7:07 am

Hi Lokesh, Thanks for info and i have small doubt weather "Bill pugh solution" is Lazy loading or Eager loading. Thanks in advance

[Reply](#)**Lokesh Gupta**

July 24, 2014 at 7:12 am



It's lazy loaded on demand only.

[Reply](#)

## Venky

July 23, 2014 at 6:57 am

Instead of double checking, why not move the `if(instance == null)` check in the synchronized block itself – so that only one thread enters, checks and decides to initialize. Please share your thoughts. Thanks.

[Reply](#)

## Lokesh Gupta

July 23, 2014 at 7:24 am

You made valid argument. It will definitely cut-down at least 2-3 lines of code needed to initialize the object. BUT, when application is running and there are N threads which want to check if object is null or not; then N-1 will be blocked because null check is in synchronized block. If we write logic as in double checking then there will not be any locking for "ONLY" checking the instance equal to null and all N threads can do it concurrently.

[Reply](#)

## Alan Amaral

January 11, 2022 at 3:49 am

I had the same idea about moving the if inside the synchronized block. It never dawned on me that this would cause extra locking every time someone asked for the instance. Very good explanation as to why using the double check is much better.

[Reply](#)

## Prateek

[July 16, 2014 at 9:17 am](#)

Hi Lokesh,

I have read a number of articles on Singleton But after reading this,I think i can confidently say in an interview that I am comfortable with this TOPIC.And also love the active participation of everyone which leads to some nice CONCLUSIONS.

[Reply](#)

## prathap

[May 7, 2014 at 2:14 pm](#)

This is the best explanation i ever saw about singleton DP.....thnx for sharing

[Reply](#)

**HIMANSU NAYAK**

May 4, 2014 at 8:05 am

Hi Lokesh,

Your way of converting difficult topic to easier is remarkable. Apart from Gang Of Four Design Patterns, if you can take time out in explaining J2EE Design patterns also then that will be really great.

[Reply](#)**Lokesh Gupta**

May 4, 2014 at 8:30 am

I will try to find time (I am usually hell of busy most of the time).

[Reply](#)**Bala**

April 25, 2014 at 6:53 am

They sequentially goes to synchronized block and create the instance.

In the above line the word "Synchronized" has to be modified as "static"

[Reply](#)

**Lokesh Gupta**

April 25, 2014 at 7:11 am

I again read the section of lazy initialization. It is correctly written. Any reason why you think so?

[Reply](#)

**Gaurav Pathak**

April 23, 2014 at 7:49 pm

Hi Lokesh Sir, i am new in java and trying to make a slideshow in my project.can you please help me what i can do.I have 2 option 1st using JavaScript and 2nd using widget.which are best for any website?

[Reply](#)

**Lokesh Gupta**

April 24, 2014 at 4:05 am

Use any already existing widget. No use of re-inventing the wheel.

[Reply](#)

**Gaurav Pathak**

April 24, 2014 at 4:39 am

okey Thanks,

[Reply](#)

## David Zhao

[April 22, 2014 at 6:24 pm](#)

Great article, thanks! BTW, is it a good idea to use the singleton for a configuration object which can be changed after creation?

Thanks!

[Reply](#)

## Lokesh Gupta

[April 22, 2014 at 6:52 pm](#)

NO. It is not good idea.

[Reply](#)

## David Zhao

[April 22, 2014 at 8:12 pm](#)

Thanks!

[Reply](#)

**Taufique S Shaikh**

April 1, 2014 at 9:28 am

Hello Lokesh,  
Very nice and informative article.

There is one typo that I have observed.  
While explaining double-checked locking , you are referring to class name as EagerSingleton but it is lazy singleton.  
I would be more clear if you make the name LazySingleton.

Thanks this is just a suggestion to make this excellent article a bit better.

Thanks Taufique Shaikh.

[Reply](#)

**Lokesh Gupta**

April 1, 2014 at 11:19 am

I will check and update. Thanks for your contribution.

[Reply](#)

**agurriion**

March 13, 2014 at 8:14 pm

Actually useful article!.

[Reply](#)

## Nagendra

February 21, 2014 at 7:31 am

Nice article... i was looking for this concept with multiple doubts. My all doubts are cleared by reading all these comments.

Thanks again LOKESH.

[Reply](#)

## bryan jacobs

February 19, 2014 at 5:44 pm

Lokesh I would love to understand why you favor the Bill pugh's solution over the Enum solution?

The enum appears to solve synchronization, serialization, and serial version id issues.

[Reply](#)

## Lokesh Gupta

February 19, 2014 at 6:38 pm

- 1) enums do not support lazy loading. Bill pugh solution does.
- 2) Though it's very very rare but if you changed your mind and now want to convert your singleton to multiton, enum would not allow this.

If above both cases are no problem for anybody, the enum is better.

Anyway, java enums are converted to classes only with additional methods e.g. values() valueOf()... etc.

[Reply](#)

**Amit**

[February 16, 2014 at 5:06 am](#)

Hi Lokesh,

Thanks for writing such a good informative blog. But i had one doubt on singleton. As per my understanding singleton is "Single instance of class per JVM " . But when application is running production clustered environment with multiple instances of JVM running can break the singleton behavior of the class? i am not sure if does that make any sense or not but question stricked in my mind an thought of clearing it.

Amit

[Reply](#)

**Lokesh Gupta**

[February 16, 2014 at 9:45 am](#)



You are right.. In clustered deployment, there exist multiple instances of singleton. You are right, singleton is one per JVM. That's why never use singleton to store runtime data. Use it for storing global static data.

[Reply](#)

**rovome**

January 7, 2014 at 7:22 am

One major drawback of the singletons presented here (as well as enums in general) is however that they can never be garbage collected which further leads to the classloader which loaded them (and all of the classes it loaded) can never be unloaded and therefore will raise a memory leak. While for small applications this does not matter that much, for larger applications with plugin-support or hot-deployment feature this is a major issue as the application container has to be restarted regularly!

While "old-style" singletons offer a simple workaround on using a WeakSingleton pattern (see the code below) enums still have this flaw. A WeakSingleton simply is created like this:

```
public class WeakSingleton
{
    private static WeakReference REFERENCE;

    private WeakSingleton()
    {

    }
}
```

```
public final static WeakSingleton getInstance()
{
    if (REFERENCE == null)
    {
        synchronized(WeakReference.class)
        {
            if (REFERENCE == null)
            {
                WeakSingleton instance = new WeakSingleton();
                REFERENCE = new WeakReference(instance);
                return instance;
            }
        }
    }
    WeakSingleton instance = REFERENCE.get();
    if (instance != null)
        return instance;

    synchronized(WeakSingleton.class)
    {
        WeakSingleton instance = new WeakSingleton();
        REFERENCE = new WeakReference(instance);
        return instance;
    }
}
```

It behaves actually like a real singleton. If however no strong reference is pointing at the WeakSingleton it gets eligible for garbage collection – so it may get destroyed and therefore lose any state. If at least one class is keeping a strong reference to the weak singleton it won't get unloaded. This way it is possible to provide singleton support in large application containers and take care of perm-gen out of memory exceptions on un/reloading multiple applications at runtime.

However if the weak singleton is used wrongly, it may lead to surprises:

```
// some code here. Assume WeakSingleton was not invoked before so no  
other object has a reference to the singleton
```

```
...  
{  
WeakSingleton someManager = WeakSingleton.getInstance();  
manager.setSomeStateValue(new StateValue());  
...  
StateValue value = manager.getValue(); // safe operation  
...  
}  
...  
// outside of the block – assume garbage collection hit just the millisecond  
before  
StateValue value = manager.getValue(); // might be null, might be the value set  
before, might be a default value
```

As mentioned before enum singletons don't provide such a mechanism and therefore will prevent the GC from collecting enums ever. Java internally converts enums to classes and adds a couple of methods like `name()`, `ordinal()`, ... Basically an enum like:

```
public enum Gender  
{  
FEMALE,  
MALE;  
}
```

will be converted to

```
public class Gender  
{
```

```
public final static Gender FEMALE = new Gender();  
public final static Gender MALE = new Gender();  
  
....  
}
```

As on declaring Gender.FEMALE somewhere a strong reference will be created for FEMALE that points to itself and therefore will prevent the enum from being garbage collected ever. The workarround with WeakReferences is not possible in that case. The only way currently possible is to set the instances via reflection to null (see [here](#)). If this enum singleton is however shared among a couple of applications nulling out the instance will with certainty lead to a NullPointerException somewhere else.

As enums prevent the classloader which loaded the enum from being garbage collected and therefore prevent the cleanup of space occupied by all the classes it loaded, it leads to memory leaks and nightmares for application container developer and app-developer who have to use these containers to run their business' apps.

[Reply](#)

**Lokesh Gupta**

January 7, 2014 at 12:08 pm

Thanks for your comment here. This is real value addition to this post. Yes, I agree that singleton are hard to garbage collect, but frameworks (e.g. Spring) have used them as default scope for a reason and benefits they provide. Regarding weak references, i agree with your analysis.

[Reply](#)

**rovome**

January 7, 2014 at 7:25 pm

small correction of the WeakSingleton code above as it was already late yesterday when I wrote the post: The class within the first synchronized(...) statemend should be WeakSingleton.class not WeakReference.class. Moreover, the comment-software removed the generic-syntax-tokens (which the compiler is doing too internally but additionally adds a cast for each generic call instead) – so either a generic type has to be added to the WeakReference of type WeakSingleton or a cast on REFERENCE.get() to actually return a WeakSingleton instead of an Object.

Moreover, the transformation code for the enum into class is not 100% exact and therefore correct – so don't quote me on that. It was just a simplification to explain why enums create a memory leak.

[Reply](#)**RAMJANE**

January 5, 2014 at 1:20 pm

Hi Lokesh,

It seems like the this can be broken using reflexion API as well.  
is the static inner class will be loaded at first call? or at class load time?

[Reply](#)

**Lokesh Gupta**

January 5, 2014 at 4:38 pm

It will be loaded at first call. Can you please elaborate more on how it can be broken?

[Reply](#)

**RAMJANE**

January 5, 2014 at 4:54 pm

Thanks Lokesh .But we can use static inner class variable and mark as null using reflexion.

Is any way to save the private variable from reflexion???

[Reply](#)

**Lokesh Gupta**

January 5, 2014 at 5:36 pm

Unfortunately not. Because java does not prevent you from changing private variables through reflexion. Reflexion is sometimes villain, isn't it.

[Reply](#)

**RAMJANE**

January 5, 2014 at 5:48 pm

Yes. Never got the way to block Reflexion. Thanks nice artical

**Mos**

May 4, 2014 at 9:32 am

in order to block Reflection you just need to throw `IllegalStateException` from the private constructor.

Like this :

```
private Singleton() {  
    // Check if we already have an instance  
    if (INSTANCE != null) {  
        throw new IllegalStateException("Singleton" +  
            " instance already created.");  
    }  
    System.out.println("Singleton Constructor Running...");  
}
```

**Anonymouse**

July 15, 2014 at 1:01 pm

Hi Lokesh,

You can actually change the values of private variables using reflection API. This can be done using `getDeclaredField(String name)` and `getDeclaredFields()` methods in `Class` class. This way you can get the an object of the field and then you can call the `set(Object obj, Object value)`

method on the Field object to change its value. For Singleton classes, the private constructor should always be written as follow:

```
private Singleton()  
{  
    if (DemoSingletonHolder.INSTANCE != null)  
    {  
        throw new IllegalStateException("Cannot create second instance of this  
class");  
    }  
}
```

**Lokesh Gupta**

July 15, 2014 at 1:30 pm

You are right. In-fact, very good suggestion.

**Nitish**

December 9, 2013 at 4:54 pm

Thanks for the article.

Making EagerSingleton instance volatile does not have any significance. As it is a static variable, it will be initialized only once when class is loaded. Thus, it will always be safely published. Agree?

[Reply](#)



**Lokesh Gupta**

December 9, 2013 at 9:56 pm

yup

[Reply](#)**Pankaj Josi**

November 30, 2013 at 7:41 pm

Lokesh, Many thanks for sharing the wonderful information....I have got one problem in my mind from my product only. We have helperclass for almost entire the functionality and we create single ton instance of each class. Now suppose I have a helperclass CustomHelper and there is method called createCustomer(). Singleton instance of this class is present. Now this is very important class of my application. And access by so many teller from banks....Then if I will get the second request than it would not be process because my class is single ton...1st thread is using the instance of singleton class....Could you pls share your point....

[Reply](#)**Lokesh Gupta**

November 30, 2013 at 8:10 pm

Singleton means only one instance of class. It does not add any other specific behavior implicitly, So createCustomer() will be synchronized only if you declare it to be, not because CustomHelper class is singleton.

Regarding your request would be blocked, this depends on cost of synchronization. If cost is not much, the you can do it without any problem. This happens all the time at DAO layer.

[Reply](#)

## H Singh

[November 20, 2013 at 8:06 pm](#)

why we need this readResolve() method and how it will solve the issue, is not clear.

Can u please explain how it is resolving the issue of more than one instances of singleton class.

[Reply](#)

## Karthikaiselvan R

[November 19, 2013 at 5:50 pm](#)

Yes, Lokesh is right. Actually clone method will not work if you don't implement Cloneable interface. I hope it will through clone not supported exception.

[Reply](#)

## Lokesh Gupta

[November 19, 2013 at 10:56 pm](#)

Yes.

[Reply](#)

**H Singh**

[November 18, 2013 at 8:09 pm](#)

Hi Lokesh,

Very good post...

I have a doubt – is it possible to generate a duplicate object of a singleton class, using serialization?

Can u plz provide some ideas that how to make sure to make singleton class more safe.

Thanks in advance.

[Reply](#)

**Lokesh Gupta**

[November 18, 2013 at 11:57 pm](#)

Yes, it's possible. Please re-read the "Adding readResolve()" section again.

[Reply](#)

**nagarjunachary**

November 15, 2013 at 12:13 am

Hi Lokesh,

You have done a great job. Every thing you have written in this post is excellent. I have some doubts. Please clarify them.

Can we go for a class with all static methods instead of Singleton? Can we achieve the same functionality like Singleton ?

Give me some explanation please.

[Reply](#)

**Lokesh Gupta**

November 15, 2013 at 12:23 am

Both are different things. Singleton means one instance per JVM. Making all methods static does not stop you from creating multiple instances of it. Though all methods are static so they will be present at class level only i.e. single copy. These static methods can be called with/without creating instances of class but actually you are able to create multiple instances and that's what singleton prevents.

So in other words, You may achieve/implement the application logic but it is not completely same as singleton.

[Reply](#)

**nagarjunachary**

November 19, 2013 at 3:54 pm

Suppose If I make constructor private and write all methods as static. Like almost equal to Math class(final class, private constructor, all static methods), then what will be the difference.

I am in confusion with these two approaches. When to use what. Pls clarify Lokesh.

[Reply](#)

**Lokesh Gupta**

November 19, 2013 at 11:01 pm

Usually singletons classes are present in form of constant files or configuration files. These classes mostly have a state (data) associated with it, which can not be changed once initialized.

Math class (or similar classes) acts as singleton, but I prefer to call them "Utility classes".

[Reply](#)

**Veeru**

April 16, 2014 at 10:07 am

You can actually make use of inheritance and extend parent class in case of Singleton and its not possible if we have final class with all static methods.

[Reply](#)

## **ANILKUMAR REDDY**

[November 7, 2013 at 7:18 am](#)

Hello Lokesh,

Thank you, Your article is much useful and answered many questions which i had. Thanks again.

Regards,  
Anil Reddy

[Reply](#)

## **Trinh Phuc Tho**

[October 24, 2013 at 11:19 am](#)

Hello, I found that in case the constructor is declared to throw an exception, the solution Bill pugh can not be used.

[Reply](#)

**vnoth**

October 13, 2013 at 9:23 pm

thanks, great work!!

[Reply](#)**Konstantinos Margaritis**

October 12, 2013 at 10:14 pm

Hello Lokesh,

Congratulations for your interesting and informative article.

Please take a look at the Bill pugh solution, because the method `BillPughSingletongetInstance` is missing the return type which must be `BillPughSingleton`.

Keep up the good work...

[Reply](#)**Lokesh Gupta**

October 12, 2013 at 11:32 pm

My bad. Actually there should be a space between "BillPughSingleton" and "getInstance". Typo error. I will fix it. Thanks for pointing out.

[Reply](#)**Dowlw**[September 30, 2013 at 12:30 am](#)

hey y lokesh why do we need of the method "readResolve()" here?

[Reply](#)**Lokesh Gupta**[September 30, 2013 at 9:33 am](#)

Everything is explained in post itself. Any specific query or concern?

[Reply](#)**Manish**[September 26, 2013 at 9:25 am](#)

What about Clone ??

[Reply](#)**Lokesh Gupta**[September 26, 2013 at 9:29 am](#)



adding clone method which throws CloneNotSupportedException will be a good addition.

[Reply](#)

**chandu**

[September 2, 2013 at 12:44 pm](#)

really good work thank you very much

[Reply](#)

**sonia**

[August 8, 2013 at 2:21 pm](#)

Thanks lokesh..its nicely explained

[Reply](#)

**John**

[August 2, 2013 at 11:55 pm](#)

Hi,

How would you handle a singleton where the constructor can throw an

exception?

To get around the threading issue, I have been using the following:

```
private static ClassName inst = null;
private static Object lock = new Object();

private ClassName() throws SomeException
{
    do some operations which may throw exception...
}

public static ClassName getInstance() throws Exception
{
    if (inst == null)
        synchronized (lock)
        {
            if (inst == null)
                inst = new ClassName();
            return inst;
        }
}
```

I want to consider the Bill Pugh method, but without a reliable way of handling exceptions, it's not really universal.

[Reply](#)

**Lokesh Gupta**

August 3, 2013 at 12:04 am

Fair enough !!

[Reply](#)

**abhineet**

July 25, 2013 at 7:11 pm

Hi,

Do you really think in case of Eager Initialization example we need to have a synchronized method and even a null check is required?

[Reply](#)**Lokesh Gupta**

July 25, 2013 at 8:15 pm

You are right. No need to make synchronized. I will update the post. Thanks for pointing out.

[Reply](#)**abhineet**

July 25, 2013 at 6:53 pm

Hi,

Do u think in case of eager example we need to have a synchronized method and even this null check is required?

Because instance would have got initialized at class load itself.

[Reply](#)

**manoj**

July 9, 2013 at 2:26 pm

What is instance control?

Instance control basically refers to single instance of the class OR singleton design pattern .

Java 1.5 onwards we should always prefer ENUM to create singleton instance. It is absolutely safe . JVM guarantees that. All earlier mechanical of controlling instance to single are already broken.

So in any interview you can confidently say ENUM= provides perfect singleton implementation .

Now what is readResolve?

readResolve is nothing but a method provided in serializable class . This method is invoked when serialized object is deserialized. Through readResolve method you can control how instance will be created at the time of deserialization.Lets try to understand some code

```
public class President {  
  
    private static final President singlePresident = new President();  
  
    private President(){  
  
    }  
  
}
```

This class generates single instance of President class. singlePresident is static instance and same will be used across.

But do you see it breaks anywhere?

Please visit <http://effectivejava.blogspot.com/> for more details on this

[Reply](#)

### Rameshwar

[July 5, 2013 at 8:46 am](#)

A perfect tutorial for understanding singleton pattern. Thanks for sharing your thoughts.

[Reply](#)

### Antaryami Das

[March 2, 2013 at 11:47 am](#)

Nice post Lokesh but u have not overridden the clone method because one can create a cloned object which also violates singleton pattern.

[Reply](#)

### Lokesh Gupta

[March 2, 2013 at 11:56 am](#)

I doubt because i have not implemented Cloneable interface either. But agree, a clone method which throws CloneNotSupportedException will be a good addition.

[Reply](#)

## Java Experience (@javaexper)

[February 1, 2013 at 4:25 am](#)

this is the most complete singleton pattern implementation tutorial I have come across. I got inspired and wrote about it on my blog at Singleton code in Java. Do let me know if I missed anything.

Thanks

[Reply](#)

## myviews2express

[November 26, 2012 at 3:54 am](#)

Hey Lokesh this is indeed a good article however I would like to propose some correction in that.

1.Eager initialization: The code example stated under this heading is actually an example of lazy initialization. Your static block initialization example can come under eager initialization.

2. In "double-checked locking" we do not make method as synchronized. Rest of the things that you did in that code example are perfect. The advantage of double checked locking is ; once the instance is created, there is no need to take lock on the singleton class thereafter.

[Reply](#)

## Admin

November 26, 2012 at 4:06 am

Hello there,

Thanks for pointing out them. First was typo error and indeed was mistake. Regarding second, I checked the wiki page and you was correct. Still, I believe that making this method 'synchronized' is a good addition, and does not remove the necessity of double checking.

Thanks!!

[Reply](#)

## myviews2express

November 26, 2012 at 7:35 am

The main aim of double check is to get rid of taking a lock over and over again once singleton object is created. And we all aware the acquiring and releasing the locks is costly affair.

[Reply](#)

**Arek Szulakiewicz (@szulak)**

November 20, 2012 at 5:39 am

Thank you for this article, it was useful in writing my own post about singleton pattern in C#.

[https://www.hugedomains.com/domain\\_profile.cfm?d=szulak&e=com](https://www.hugedomains.com/domain_profile.cfm?d=szulak&e=com)

[Reply](#)

**javinpaul (@javinpaul)**

October 22, 2012 at 7:32 am

Hi Lokesh, Thanks for your comment on my post [10 interview question on Java Singleton pattern](#). I see you have also addressed issues quite well. In my opinion Enum is most easier way to implement this. You may like to see my post [Why Enum Singleton is better in Java](#).

[Reply](#)

**Admin**

October 22, 2012 at 9:20 am

Thanks @Javin for your appreciation. Yes, if you are absolutely sure that whatever the condition is, your class will always be singleton, then enum should be preferred. But, if you have a single doubt in mind that later in some stage you might switch to normal mode, or switch to multiton (allow multiple instances), then there is no easy way out in enum.



Also, enums does not support lazy loading. So again, one need to see what suites him.

[Reply](#)

**shashank**

[November 22, 2013 at 2:03 am](#)


Hi Lokesh , nice post , i need some explanation for this

```
private BillPughSingleton() {  
  
}
```

why you need this private constructor in 4th methof BillPughSingleton ??

[Reply](#)

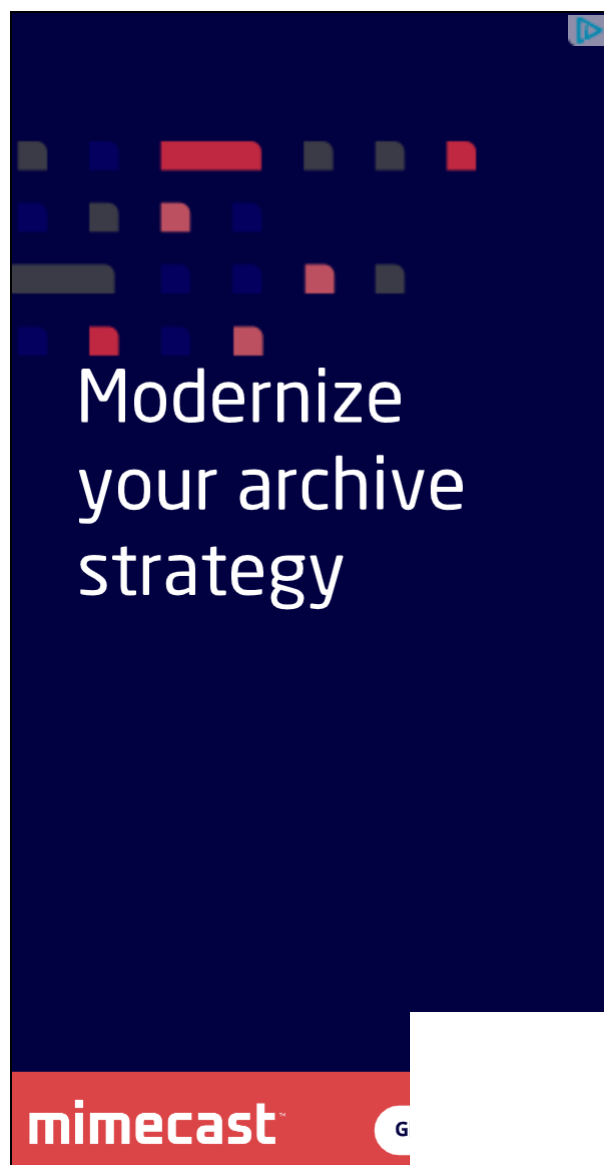
## Leave a Comment

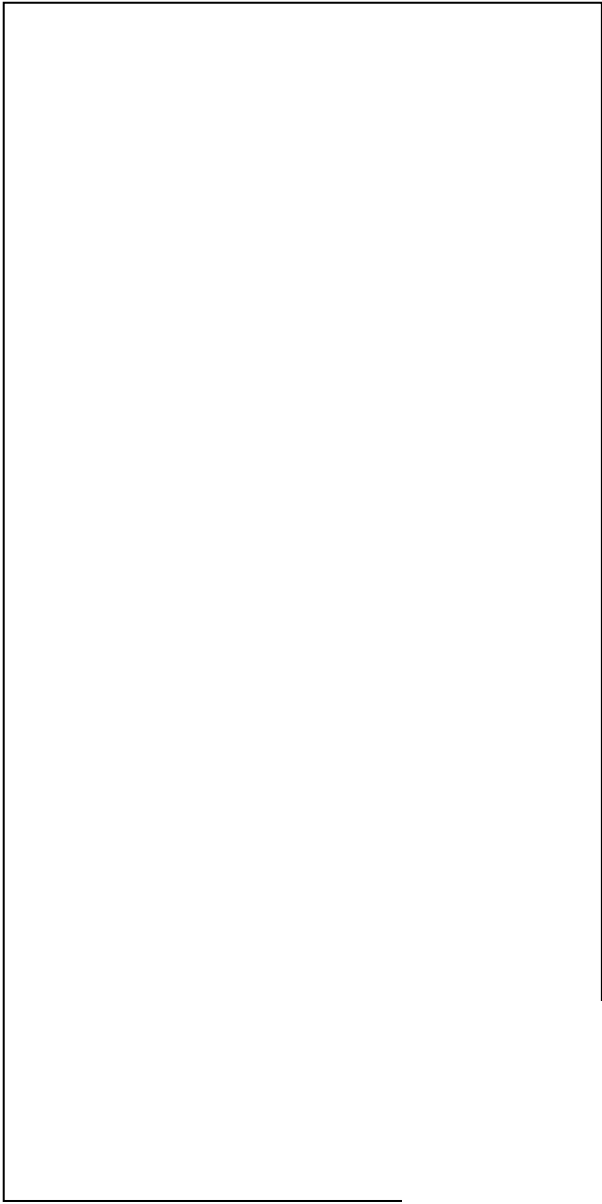


Name \*

☐ Add me to your newsletter and keep me updated whenever you publish new blog posts

## Post Comment







## HowToDoInJava

A blog about Java and related technologies, the best practices, algorithms, and interview questions.

### Meta Links

➤ [About Me](#)

- [Contact Us](#)
- [Privacy policy](#)
- [Advertise](#)
- [Guest Posts](#)

## Blogs

[REST API Tutorial](#)



Copyright © 2022 · Hosted on [Cloudways](#) · [Sitemap](#)