How To Do In Java

Singleton design pattern in java

OCTOBER 22, 2012 | LOKESH | 87 COMMENTS

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 can not be overruled either. In this post, we will discuss some good approaches and will work towards our best possible effort.

Sections in this post:

Singleton term is derived from its mathematical counterpart. It wants us, as said above, to have only one instance. Lets see the possible solutions:

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 start up. In singleton pattern, it refers to create the singleton instance irrespective of whether any other class actually asked for its instance or not.

```
1  public class EagerSingleton {
2      private static volatile EagerSingleton instance = new EagerSingleton();
3
4      // private constructor
```

```
    private EagerSingleton() {
    }

    public static EagerSingleton getInstance() {
        return instance;
    }
}
```

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

Lets solve above problem in next method.

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 singleton pattern, it restricts the creation of instance until requested first time. Lets see 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 first invocation, above method will check if instance is already created using instance variable. If there is no instance i.e. instance is null, it will create an instance and will return its reference. If instance is already created, it will simply return the reference of instance.

But, this method also has its own drawbacks. Lets see how. Suppose there are two threads T1 and T2. Both comes to create instance and execute “instance==null”, now both threads have identified instance variable to null thus assume they must create an instance. They sequentially goes to synchronized block and create the instances. At 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 synchronized block in given below way:

```
public class EagerSingleton {
```

```
    private static volatile EagerSingleton instance = null;

    // private constructor
    private EagerSingleton() {
    }

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

Above code is the correct implementation of singleton pattern.

> Please ensure to use "**volatile**" keyword with instance variable otherwise you can run into out of order write error scenario, where reference of 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 refer to uninitialized object may throw null pointer exception and can even crash the whole application.

Static block initialization

If you have little idea about class loading sequence, you can connect to the fact that static blocks are executed during the loading of class and even before the constructor is called. We can use this feature in our singleton pattern also 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() {
        // ...
    }
}
```

Above code has one drawback. Suppose there are 5 static fields in class and 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 we require it or not.

Next section will overcome this problem.

Bill pugh solution

Bill pugh was main force behind java memory model changes. His principle "Initialization-on-demand holder idiom" also uses static block but in different way. It suggest to use static inner class.

```
1  public class BillPughSingleton {
2      private BillPughSingleton() {
3      }
4
5      private static class LazyHolder {
6          private static final BillPughSingleton INSTANCE = new BillPughSingleton()
7      }
8
9      public static BillPughSingleton getInstance() {
10         return LazyHolder.INSTANCE;
11     }
12 }
```

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 also use it in my all projects.***

Using Enum

This type of implementation recommend the use of enum. Enum, as written in java docs, provide implicit support for thread safety and only one instance is guaranteed. This is also a good way to have singleton with minimum effort.

```
1  public enum EnumSingleton {
2      INSTANCE;
3      public void someMethod(String param) {
4          // some class member
5      }
6  }
```

Adding readResolve()

So, till now you must have taken your decision that how you would like to implement your single-

ton. Now lets see other problems that may arise even in interviews also.
Lets say your application is distributed and it frequently serialize the objects in file system, only to read them later when required. Please note that, de-serialization always creates a new instance. Lets understand using an example:

Our singleton class is:

```
1  public class DemoSingleton implements Serializable {
2      private volatile static DemoSingleton instance = null;
3
4      public static DemoSingleton getInstance() {
5          if (instance == null) {
6              instance = new DemoSingleton();
7          }
8          return instance;
9      }
10
11     private int i = 10;
12
13     public int getI() {
14         return i;
15     }
16
17     public void setI(int i) {
18         this.i = i;
19     }
20 }
```

Lets serialize this class and de-serialize it after making some changes:

```
1  public class SerializationTest {
2      static DemoSingleton instanceOne = DemoSingleton.getInstance();
3
4      public static void main(String[] args) {
5          try {
6              // Serialize to a file
7              ObjectOutput out = new ObjectOutputStream(new FileOutputStream(
8                      "filename.ser"));
9              out.writeObject(instanceOne);
10             out.close();
11
12             instanceOne.setI(20);
13
14             // Serialize to a file
15             ObjectInput in = new ObjectInputStream(new FileInputStream(
16                     "filename.ser"));
17             DemoSingleton instanceTwo = (DemoSingleton) in.readObject();
18             in.close();
19
20             System.out.println(instanceOne.getI());
21             System.out.println(instanceTwo.getI());
22
23         } catch (IOException e) {
24             e.printStackTrace();
25         } catch (ClassNotFoundException e) {
26             e.printStackTrace();
```

```
            }
        }
    }

    Output:
    20
    10
```

Unfortunately, both variables have different value of variable “i”. Clearly, there are two instances of our class. So, again we are in same problem of multiple instances in application.
To solve this issue, we need to include readResolve() method in our DemoSingleton class. This method will be invoked when you will de-serialize the object. Inside this method, you must return the existing instance to ensure 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
```

Adding serial version id

So far so good. Till now, we have solved the problem of synchronization and serialization both. Now, we are just one step behind our correct and complete implementation. And missing part is serial version id.

This is required in condition when you class structure can change in between you serialize the instance and go again to de-serialize it. Changed structure of class will cause JVM to give exception

while 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 class. It will prevent the compiler to throw the exception by telling that both classes are same, and will load the available instance variables only.

Conclusion

After having discussed so many possible approaches and other possible error cases, i will recommend you below code template to design your singleton class which shall ensure only one instance of class in 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 make you understand the most common approaches for singleton pattern. Let me know of you thoughts please.

Happy Learning !!

Update: 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()

Related Posts:

1. **Prototype design pattern in java**
2. **Factory design pattern in java**
3. **Adapter Design Pattern in Java**
4. **Chain of responsibility design pattern**
5. **Decorator design pattern**
6. **Visitor design pattern example tutorial**
7. **Abstract factory pattern in java**
8. **Builder Pattern in java**

BILL PUGH DESIGN PATTERN HOW TO SINGLETON

87 THOUGHTS ON "SINGLETON DESIGN PATTERN IN JAVA"

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

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?

★ Lokesh

OCTOBER 5, 2014 AT 6:07 AM

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

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) “

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

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

★ Lokesh

JULY 24, 2014 AT 7:12 AM

It’s lazy loaded on demand only.

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.

★ Lokesh

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.

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.

prathap

MAY 7, 2014 AT 2:14 PM

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

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.

> **★ Lokesh**
>
> MAY 4, 2014 AT 8:30 AM
>
> I will try to find time (I am usually hell of busy most of the time).

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”

> **★ Lokesh**
>
> 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?

Gaurav Pathak

APRIL 23, 2014 AT 7:49 PM

Hi Lokesh Sir, i am new in java and tryning 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?

> **★ Lokesh**
>
> APRIL 24, 2014 AT 4:05 AM

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

Gaurav Pathak

APRIL 24, 2014 AT 4:39 AM

okey Thanks,

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!

★ Lokesh

APRIL 22, 2014 AT 6:52 PM

NO. It is not good idea.

David Zhao

APRIL 22, 2014 AT 8:12 PM

Thanks!

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.

★ Lokesh

APRIL 1, 2014 AT 11:19 AM

I will check and update. Thanks for your contribution.

agurrion

MARCH 13, 2014 AT 8:14 PM

Actually useful article!.

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.

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.

★ Lokesh

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.

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

★ Lokesh

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.

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 bahaves 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 wont 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 http://www.jroller.com/VelkaVrana/entry/modify_enum_with_reflection). 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 developerand app-developer who have to use these containers to run their business' apps.

★ 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.

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.

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?

★ 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 broken?

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 reflxion???

★ 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.

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 it's 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

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?

★ Lokesh Gupta

DECEMBER 9, 2013 AT 9:56 PM

yup

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 helperclasse for almost entire the functionality ad we create single ton instance of each class.Now suppose I have a helperclass CustomeHelper 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 may teller from banks…..Than 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….

★ 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 CustomeHelper 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.

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.

Karthikaiselvan R

NOVEMBER 19, 2013 AT 5:50 PM

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

★ Lokesh Gupta

NOVEMBER 19, 2013 AT 10:56 PM

Yes.

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.

★ Lokesh Gupta

NOVEMBER 18, 2013 AT 11:57 PM

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

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.

★ 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.

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.

★ 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".

Veeru

APRIL 16, 2014 AT 10:07 AM

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

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

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.

vnoth

OCTOBER 13, 2013 AT 9:23 PM

thanks, great work!!

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…

★ 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.

Dowlw

SEPTEMBER 30, 2013 AT 12:30 AM

heyy lokesh why do we need of the method “readResolve()” here?

> **★ Lokesh Gupta**
>
> SEPTEMBER 30, 2013 AT 9:33 AM
>
> Everything is explained in post itself. Any specific query or concern?

Manish

SEPTEMBER 26, 2013 AT 9:25 AM

What about Clone ??????????????????????????????????

> **★ Lokesh Gupta**
>
> SEPTEMBER 26, 2013 AT 9:29 AM
>
> adding clone method which throws CloneNotSupportedException will be a good addition.

chandu

SEPTEMBER 2, 2013 AT 12:44 PM

really good work thank you very much

sonia

AUGUST 8, 2013 AT 2:21 PM

Thanks lokesh..its nicely explained

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.

★ Lokesh Gupta

AUGUST 3, 2013 AT 12:04 AM

Fair enough !!

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?

★ 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.

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.

Pingback: Singleton design pattern in java | back2dev | S...

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 abso-

lutely 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://efectivejava.blogspot.in/ for more details on this

Rameshwar

JULY 5, 2013 AT 8:46 AM

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

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.

Lokesh Gupta

MARCH 2, 2013 AT 11:56 AM

I doubt because i have not implemented Cloneable interface either. But agree, a

clone methos which throws CloneNotSupportedException will be a good addition.

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

myviews2express

NOVEMBER 26, 2012 AT 3:54 AM

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

1.Eager initialization: The code example stated under this heading is actually a 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.

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!!

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.

Pingback: How to do deep cloning using in memory serialization in java « How to do in JAVA

Arek Szulakiewicz (@szulak)

NOVEMBER 20, 2012 AT 5:39 AM

Thank you for this article, it was useful in writting my own post about singleton pattern in C#.
http://szulak.com/singleton-pattern-in-csharp/

Pingback: Singleton design pattern in java | singleton in java | Scoop.it

Pingback: Cheatsheet: 2012 10.17 ~ 10.23 - gOODiDEA.NET

Pingback: Implementing factory design pattern in java « How to do in "JAVA"

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.

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 pre-ferred. 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.

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 ??

Note:- In comment box, please put your code inside **[java] ... [/java]** OR **[xml] ... [/xml]** tags otherwise it may not appear as intended.