**Singleton Design Pattern**

*In software engineering, the singleton pattern is a software design pattern that restricts the instantiation of a class to one object. This is useful when exactly one object is needed to coordinate actions across the system.*

Singleton design pattern comes under creational design pattern and is a part of “Gang of Four Design Pattern”.

Singleton design pattern ensures that there would be only instance created per JVM and the single object will be shared across all the classes or throughout the application. For example multiple we save data in multiple created DAO classes using DB connection, so does it make any sense to create multiple object in each class to save data in DB. So we create singleton object for DB connection and share it throughout the application or to DAO layer.

There are following ways to create Singleton object.

1. Eager Loading
2. Lazy Loading
3. By Make getInstance() synchronized
4. [Double Checked Locking](https://en.wikipedia.org/wiki/Double-checked_locking)
5. Using enum

**How to create Singleton design pattern**

To create Singleton class we have to follow following steps:

1. **Static member:** This contains the instance of Singleton object.
2. **Private constructor:** This will prevent our class from anybody to create object from outside.
3. **Static factory method:** this is the factory method for outside word to get only one / single instance of our class.

1. **Eager Loading:** eager loading is the concept in which we create object before its demand. Like in some parties we see that on reception there are already water filled glasses are available. They fill glass with water in advance before relatives come and serve them those already available glasses of water.

Already filled

One glass of water please

Figure 1

![im1 eagerloading]( https://github.com/PiyushMittl/java/blob/master/java\_designpattern\_singleton\_im1\_eagerload.png)

Singleton Eager loading Example:

public class EagerA{

private static EagerA a = new EagerA(); // eagerly loading of object

private EagerA(){}

public static getA(){

return a;

}

}

**Problem with the approach:**

1. **Lazy Loading :** lazy loading is the concept in which we create object on its demand. If we take previous example we already had water filled glasses available but in case of Lazy loading the difference would be they will fill the glass with water in when relatives come and ask for the glass of water instead of already available glasses of water.

One glass of water please

One glass of water filled on demand and given

![im2 lazyloading]( https://github.com/PiyushMittl/java/blob/master/java\_designpattern\_singleton\_im1\_eagerload.png)

public class A{

private static A a=null;

private A(){}

public static getA(){

if(a==null)

{

a = new A();//lazy loading

}

return a;

}

}

**Problem with the approach:**

1. **By Make getInstance() synchronized :** Above both example would run perfectly and will generate single object but there is a issue with above approach with lazy loading i.e

The method getA() is not thread safe and can create if two threads enters in it simultaneously. If two threads enter in this method at same time they would found a==null and object will be created for each of the thread.

**T2**

**T1**

public static getA(){

if(a==null)

{

a = new A();//lazy loading

}

return a;

}

![im3 synchro]( https://github.com/PiyushMittl/java/blob/master/java\_designpattern\_singleton\_im1\_eagerload.png)

Here thread T1 and T2 both have entered in getA method and checking “if condition” if class has alredy created new object for A or not and since they have entered in getA same time they are checking if condition in same time too. As we can see in above diagram they found a==null and create one object for each thread. That means two object would be created for each thred. Which is not in the case of Singleton disingn patterns, here it will break our check for single instance.

public class A{

private static A a=null;

private A(){}

public static **synchronized** getA(){

if(a==null)

{

a = new A();//lazy loading

}

return a;

}

}

Here using synchronized will ensure that in case of multithreaded environment we only one thread would enter to create object.

**Problem with the approach:** The main disadvantage of this is method is that using synchronized every time while creating the singleton object is expensive and may decrease the performance of your program. However if performance of getInstance() is not critical for your application this method provides a clean and simple solution.

1. [**Double Checked Locking**](https://en.wikipedia.org/wiki/Double-checked_locking) **:** if we notice carefully we can see that once the object is created **synchronization** is not required. now obj will not be null and any sequence of operations will lead to consistent results.  
   So we will only acquire lock on the getInstance() once, when the obj is null. This way we only synchronize the first way through, just what we want.

public class A{

private **volatile** static A a=null;

private A(){}

public static getA(){

**synchronized (Singleton.class)**

{

if(a==null)

{

a = new A();//lazy loading

}

return a;

}

}

}

We have declared the obj [volatile](https://www.geeksforgeeks.org/volatile-keyword-in-java/) which ensures that multiple threads offer the obj variable correctly when it is being initialized to Singleton instance. This method drastically reduces the overhead of calling the synchronized method every time.

1. **Using enum :** if we notice carefully we can see that once the object is created **synchronization** is not required. now obj will not be null and any sequence of operations will lead to consistent results.

Public enum A{

INSTANCE;

Public void doStuff{

System.out.println(singleton using enum);

}

}

Public static void main(String a[]){

A.INSTANCE.doStuff();

}

**Destroy Singleton using Reflection and Serialization**

**Using Reflection:**

import java.lang.reflect.Constructor;

public class ReflectionATest {

public static void main(String[] args) {

EagerA instanceOne = EagerA.getA();

EagerA instanceTwo = null;

try {

Constructor[] constructors = EagerA.class.getDeclaredConstructors();

for (Constructor constructor : constructors) {

//Below code will destroy the singleton pattern

constructor.setAccessible(true);

instanceTwo = (EagerA) constructor.newInstance();

break;

}

} catch (Exception e) {

e.printStackTrace();

}

System.out.println(instanceOne.hashCode());

System.out.println(instanceTwo.hashCode());

}

}

When you run the above test class, you will notice that hashCode of both the instances are not same that destroys the singleton pattern. Reflection is very powerful and used in a lot of frameworks like Spring and Hibernate

**Using Serialization:**

import java.io.Serializable;

public class SerializedSingleton implements Serializable{

private static final long serialVersionUID = -7604766932017737115L;

private SerializedSingleton(){}

private static class SingletonHelper{

private static final SerializedSingleton instance = new SerializedSingleton();

}

public static SerializedSingleton getInstance(){

return SingletonHelper.instance;

}

}

The problem with above serialized singleton class is that whenever we deserialize it, it will create a new instance of the class. Let’s see it with a simple program.

import java.io.FileInputStream;

import java.io.FileNotFoundException;

import java.io.FileOutputStream;

import java.io.IOException;

import java.io.ObjectInput;

import java.io.ObjectInputStream;

import java.io.ObjectOutput;

import java.io.ObjectOutputStream;

public class SingletonSerializedTest {

public static void main(String[] args) throws FileNotFoundException, IOException, ClassNotFoundException {

SerializedSingleton instanceOne = SerializedSingleton.getInstance();

ObjectOutput out = new ObjectOutputStream(new FileOutputStream(

"filename.ser"));

out.writeObject(instanceOne);

out.close();

//deserailize from file to object

ObjectInput in = new ObjectInputStream(new FileInputStream(

"filename.ser"));

SerializedSingleton instanceTwo = (SerializedSingleton) in.readObject();

in.close();

System.out.println("instanceOne hashCode="+instanceOne.hashCode());

System.out.println("instanceTwo hashCode="+instanceTwo.hashCode());

}

}

Output of the above program is;

instanceOne hashCode=2011117821

instanceTwo hashCode=109647522

So it destroys the singleton pattern, to overcome this scenario all we need to do it provide the implementation of readResolve() method.

protected Object readResolve() {

return getInstance();

}

After this you will notice that hashCode of both the instances are same in test program.

**FAQs:**

Question: Why can’t we use a static class instead of singleton?  
Answer:

* One of the key advantages of singleton over static class is that it can implement interfaces and extend classes while the static class cannot (it can extend classes, but it does not inherit their instance members). If we consider a static class it can only be a nested static class as top level class cannot be a static class. Static means that it belongs to a class it is in and not to any instance. So it cannot be a top level class.
* Another difference is that static class will have all its member as static only unlike Singleton.
* Another advantage of Singleton is that it can be lazily loaded whereas static will be initialized whenever it is first loaded.
* Singleton object stores in Heap but, static object stores in stack.
* We can clone the object of Singleton but, we can not clone the static class object.
* Singleton can use the Object Oriented feature of polymorphism but static class cannot.

Question: Can the singleton class be subclassed?  
Answer: Frankly speaking singleton is just a design pattern and it can be subclassed. However it is worth to understand the logic or requirement behind subclassing a singleton class as the child class might not inherit the singleton pattern objective by extending the Singleton class. However the subclassing can be prevented by using the final keyword in the class declaration.

Question: Can there be multiple instance of singleton using cloning?  
Answer: That was a good catch! What do we do now? To prevent the another instance to be created of the singleton instance we can throw exception from inside the clone() method.

Question: What is the impact if we are creating another instance of singleton using serialization and deserialization?  
Answer: When we serialize a class and deserialize it then it creates another instance of the singleton class. Basically as many times as you deserialize the singleton instance it will create multiple instance. Well in this case the best way is to make the singleton as enum. In that way the underlying Java implementation takes care of all the details. If this is not possible then we will need to override the readobject() method to return the same singleton instance.

Question: Which other pattern works with Singleton?  
Answer:There are several other pattern like Factory method, builder and prototype pattern which uses Singleton pattern during the implementation.

Question: Which classes in JDK uses singleton pattern?  
Answer: java.lang.Runtime : In every Java application there is only one Runtime instance that allows the application to interface with the environment it is running. The getRuntime is equivalent to the getInstance() method of the singleton class.