# Java Singleton Design Pattern Best Practices with Example

**Introduction**

*Java Singleton Pattern* is one of the [*Gangs of Four Design patterns*](https://www.digitalocean.com/community/tutorials/gangs-of-four-gof-design-patterns) and comes in the *Creational Design Pattern* category. From the definition, it seems to be a straightforward design pattern, but when it comes to implementation, it comes with a lot of concerns.

In this article, we will learn about singleton design pattern principles, explore different ways to implement the singleton design pattern, and some of the best practices for its usage.

## Singleton Pattern Principles

* Singleton pattern restricts the instantiation of a class and ensures that only one instance of the class exists in the Java Virtual Machine.
* The singleton class must provide a global access point to get the instance of the class.
* Singleton pattern is used for [logging](https://www.digitalocean.com/community/tutorials/logger-in-java-logging-example), drivers objects, caching, and [thread pool](https://www.digitalocean.com/community/tutorials/threadpoolexecutor-java-thread-pool-example-executorservice).
* Singleton design pattern is also used in other design patterns like [Abstract Factory](https://www.digitalocean.com/community/tutorials/abstract-factory-design-pattern-in-java), [Builder](https://www.digitalocean.com/community/tutorials/builder-design-pattern-in-java), [Prototype](https://www.digitalocean.com/community/tutorials/prototype-design-pattern-in-java), [Facade](https://www.digitalocean.com/community/tutorials/facade-design-pattern-in-java), etc.
* Singleton design pattern is used in core Java classes also (for example, java.lang.Runtime, java.awt.Desktop).

## Java Singleton Pattern Implementation

To implement a singleton pattern, we have different approaches, but all of them have the following common concepts.

* Private constructor to restrict instantiation of the class from other classes.
* Private static variable of the same class that is the only instance of the class.
* Public static method that returns the instance of the class, this is the global access point for the outer world to get the instance of the singleton class.

In further sections, we will learn different approaches to singleton pattern implementation and design concerns with the implementation.

## 1. Eager initialization

In eager initialization, the instance of the singleton class is created at the time of class loading. The drawback to eager initialization is that the method is created even though the client application might not be using it. Here is the implementation of the static initialization singleton class:

package com.journaldev.singleton;

public class EagerInitializedSingleton {

private static final EagerInitializedSingleton instance = new EagerInitializedSingleton();

// private constructor to avoid client applications using the constructor

private EagerInitializedSingleton(){}

public static EagerInitializedSingleton getInstance() {

return instance;

}

}

Copy

If your singleton class is not using a lot of resources, this is the approach to use. But in most of the scenarios, singleton classes are created for resources such as File System, Database connections, etc. We should avoid the instantiation unless the client calls the getInstance method. Also, this method doesn’t provide any options for exception handling.

## 2. Static block initialization

[Static block](https://www.digitalocean.com/community/tutorials/static-keyword-in-java) initialization implementation is similar to eager initialization, except that instance of the class is created in the static block that provides the option for [exception handling](https://www.digitalocean.com/community/tutorials/exception-handling-in-java).

package com.journaldev.singleton;

public class StaticBlockSingleton {

private static StaticBlockSingleton instance;

private StaticBlockSingleton(){}

// static block initialization for exception handling

static {

try {

instance = new StaticBlockSingleton();

} catch (Exception e) {

throw new RuntimeException("Exception occurred in creating singleton instance");

}

}

public static StaticBlockSingleton getInstance() {

return instance;

}

}

Copy

Both eager initialization and static block initialization create the instance even before it’s being used and that is not the best practice to use.

## 3. Lazy Initialization

Lazy initialization method to implement the singleton pattern creates the instance in the global access method. Here is the sample code for creating the singleton class with this approach:

package com.journaldev.singleton;

public class LazyInitializedSingleton {

private static LazyInitializedSingleton instance;

private LazyInitializedSingleton(){}

public static LazyInitializedSingleton getInstance() {

if (instance == null) {

instance = new LazyInitializedSingleton();

}

return instance;

}

}

Copy

The preceding implementation works fine in the case of the single-threaded environment, but when it comes to multi-threaded systems, it can cause issues if multiple threads are inside the if condition at the same time. It will destroy the singleton pattern and both threads will get different instances of the singleton class. In the next section, we will see different ways to create a [thread-safe](https://www.digitalocean.com/community/tutorials/thread-safety-in-java) singleton class.

## 4. Thread Safe Singleton

A simple way to create a thread-safe singleton class is to make the global access method [synchronized](https://www.digitalocean.com/community/tutorials/thread-safety-in-java) so that only one thread can execute this method at a time. Here is a general implementation of this approach:

package com.journaldev.singleton;

public class ThreadSafeSingleton {

private static ThreadSafeSingleton instance;

private ThreadSafeSingleton(){}

public static synchronized ThreadSafeSingleton getInstance() {

if (instance == null) {

instance = new ThreadSafeSingleton();

}

return instance;

}

}

Copy

The preceding implementation works fine and provides thread-safety, but it reduces the performance because of the cost associated with the synchronized method, although we need it only for the first few threads that might create separate instances. To avoid this extra overhead every time, double-checked locking principle is used. In this approach, the synchronized block is used inside the if condition with an additional check to ensure that only one instance of a singleton class is created. The following code snippet provides the double-checked locking implementation:

public static ThreadSafeSingleton getInstanceUsingDoubleLocking() {

if (instance == null) {

synchronized (ThreadSafeSingleton.class) {

if (instance == null) {

instance = new ThreadSafeSingleton();

}

}

}

return instance;

}

Copy

Continue your learning with [Thread Safe Singleton Class](https://www.digitalocean.com/community/tutorials/thread-safety-in-java-singleton-classes).

## 5. Bill Pugh Singleton Implementation

Prior to Java 5, the Java memory model had a lot of issues, and the previous approaches used to fail in certain scenarios where too many threads tried to get the instance of the singleton class simultaneously. So [Bill Pugh](https://en.wikipedia.org/wiki/William_Pugh_(computer_scientist)) came up with a different approach to create the singleton class using an [inner static helper class](https://www.digitalocean.com/community/tutorials/java-inner-class). Here is an example of the Bill Pugh Singleton implementation:

package com.journaldev.singleton;

public class BillPughSingleton {

private BillPughSingleton(){}

private static class SingletonHelper {

private static final BillPughSingleton INSTANCE = new BillPughSingleton();

}

public static BillPughSingleton getInstance() {

return SingletonHelper.INSTANCE;

}

}

Copy

Notice the private inner static class that contains the instance of the singleton class. When the singleton class is loaded, SingletonHelper class is not loaded into memory and only when someone calls the getInstance() method, this class gets loaded and creates the singleton class instance. This is the most widely used approach for the singleton class as it doesn’t require synchronization.

## 6. Using Reflection to destroy Singleton Pattern

Reflection can be used to destroy all the previous singleton implementation approaches. Here is an example class:

package com.journaldev.singleton;

import java.lang.reflect.Constructor;

public class ReflectionSingletonTest {

public static void main(String[] args) {

EagerInitializedSingleton instanceOne = EagerInitializedSingleton.getInstance();

EagerInitializedSingleton instanceTwo = null;

try {

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

for (Constructor constructor : constructors) {

// This code will destroy the singleton pattern

constructor.setAccessible(true);

instanceTwo = (EagerInitializedSingleton) constructor.newInstance();

break;

}

} catch (Exception e) {

e.printStackTrace();

}

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

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

}

}

Copy

When you run the preceding test class, you will notice that hashCode of both instances is not the same which destroys the singleton pattern. Reflection is very powerful and used in a lot of frameworks like Spring and Hibernate. Continue your learning with [Java Reflection Tutorial](https://www.digitalocean.com/community/tutorials/java-reflection-example-tutorial).

## 7. Enum Singleton

To overcome this situation with Reflection, [Joshua Bloch](https://en.wikipedia.org/wiki/Joshua_Bloch) suggests the use of enum to implement the singleton design pattern as Java ensures that any enum value is instantiated only once in a Java program. Since [Java Enum](https://www.digitalocean.com/community/tutorials/java-enum) values are globally accessible, so is the singleton. The drawback is that the enum type is somewhat inflexible (for example, it does not allow lazy initialization).

package com.journaldev.singleton;

public enum EnumSingleton {

INSTANCE;

public static void doSomething() {

// do something

}

}

## 8. Serialization and Singleton

Sometimes in distributed systems, we need to implement Serializable interface in the singleton class so that we can store its state in the file system and retrieve it at a later point in time. Here is a small singleton class that implements Serializable interface also:

package com.journaldev.singleton;

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 serialized singleton class is that whenever we deserialize it, it will create a new instance of the class. Here is an example:

package com.journaldev.singleton;

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();

// deserialize 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());

}

}

Copy

That code produces this output:

Output

instanceOne hashCode=2011117821

instanceTwo hashCode=109647522

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

protected Object readResolve() {

return getInstance();

}

Copy

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

Read about [Java Serialization](https://www.digitalocean.com/community/tutorials/objectoutputstream-java-write-object-file) and [Java Deserialization](https://www.digitalocean.com/community/tutorials/serialization-in-java).

## Conclusion

This article covered the singleton design pattern.

Continue your learning with more [Java tutorials](https://www.digitalocean.com/community/tags/java).

# Java Singleton Design Pattern Practices with Examples

* **Difficulty Level :** [Medium](https://www.geeksforgeeks.org/medium/)
* **Last Updated :** 13 Jun, 2022

 Read

 Discuss

In [previous](https://www.geeksforgeeks.org/singleton-design-pattern-introduction/) articles, we discussed about singleton design pattern and singleton class [implementation](https://www.geeksforgeeks.org/singleton-design-pattern/)in detail.   
In this article, we will see how we can create singleton classes. After reading this article you will be able to create your singleton class according to your requirement, which is simple and without bottlenecks.   
There are many ways this can be done in Java. All these ways differ in their implementation of the pattern, but in the end, they all achieve the same end result of a single instance. 

1. **Eager initialization:** This is the simplest method of creating a singleton class. In this, object of class is created when it is loaded to the memory by JVM. It is done by assigning the reference of an instance directly.   
   It can be used when program will always use instance of this class, or the cost of creating the instance is not too large in terms of resources and time.

* JAVA

|  |
| --- |
| // Java code to create singleton class by  // Eager Initialization  **public** **class** GFG  {    // public instance initialized when loading the class  **private** **static** **final** GFG instance = **new** GFG();    **private** GFG()    {      // private constructor    }  **public** **static** GFG getInstance(){  **return** instance;      }  } |

1. **Pros:**
   1. Very simple to implement.
   2. May lead to resource wastage. Because instance of class is created always, whether it is required or not.
   3. CPU time is also wasted in creation of instance if it is not required.
   4. Exception handling is not possible.
2. **Using static block:** This is also a sub part of Eager initialization. The only difference is object is created in a static block so that we can have access on its creation, like exception handling. In this way also, object is created at the time of class loading.   
   It can be used when there is a chance of exceptions in creating object with eager initialization.

* JAVA

|  |
| --- |
| // Java code to create singleton class  // Using Static block  **public** **class** GFG  {    // public instance  **public** **static** GFG instance;    **private** GFG()    {      // private constructor    }  **static**    {      // static block to initialize instance      instance = **new** GFG();    }  } |

1. **Pros:**
   1. Very simple to implement.
   2. No need to implement getInstance() method. Instance can be accessed directly.
   3. Exceptions can be handled in static block.
   4. May lead to resource wastage. Because instance of class is created always, whether it is required or not.
   5. CPU time is also wasted in creation of instance if it is not required.
2. **Lazy initialization:** In this method, object is created only if it is needed. This may prevent resource wastage. An implementation of getInstance() method is required which return the instance. There is a null check that if object is not created then create, otherwise return previously created. To make sure that class cannot be instantiated in any other way, constructor is made final. As object is created with in a method, it ensures that object will not be created until and unless it is required. Instance is kept private so that no one can access it directly.   
   It can be used in a single threaded environment because multiple threads can break singleton property as they can access get instance method simultaneously and create multiple objects.

* JAVA

|  |
| --- |
| //Java Code to create singleton class  // With Lazy initialization  **public** **class** GFG  {    // private instance, so that it can be    // accessed by only by getInstance() method  **private** **static** GFG instance;    **private** GFG()    {      // private constructor    }      //method to return instance of class  **public** **static** GFG getInstance()    {  **if** (instance == **null**)      {        // if instance is null, initialize        instance = **new** GFG();      }  **return** instance;    }  } |

1. **Pros:**
   1. Object is created only if it is needed. It may overcome wastage of resource and  CPU time.
   2. Exception handling is also possible in method.
   3. Every time a condition of null has to be checked.
   4. instance can’t be accessed directly.
   5. In multithreaded environment, it may break singleton property.
2. **Thread Safe Singleton:** A thread safe singleton is created so that singleton property is maintained even in multithreaded environment. To make a singleton class thread safe, getInstance() method is made synchronized so that multiple threads can’t access it simultaneously.

* JAVA

|  |
| --- |
| // Java program to create Thread Safe  // Singleton class  **public** **class** GFG  {    // private instance, so that it can be    // accessed by only by getInstance() method  **private** **static** GFG instance;    **private** GFG()    {      // private constructor    }     //synchronized method to control simultaneous access  **synchronized** **public** **static** GFG getInstance()    {  **if** (instance == **null**)      {        // if instance is null, initialize        instance = **new** GFG();      }  **return** instance;    }  } |

1. **Pros:**
   1. Lazy initialization is possible.
   2. It is also thread safe.
   3. getInstance() method is synchronized so it causes slow performance as multiple threads can’t access it simultaneously.
2. **Lazy initialization with Double check locking:** In this mechanism, we overcome the overhead problem of synchronized code. In this method, getInstance is not synchronized but the block which creates instance is synchronized so that minimum number of threads have to wait and that’s only for first time.

* JAVA

|  |
| --- |
| // Java code to explain double check locking  **public** **class** GFG  {    // private instance, so that it can be    // accessed by only by getInstance() method  **private** **static** GFG instance;    **private** GFG()    {      // private constructor    }    **public** **static** GFG getInstance()    {  **if** (instance == **null**)      {        //synchronized block to remove overhead  **synchronized** (GFG.**class**)        {  **if**(instance==**null**)          {            // if instance is null, initialize            instance = **new** GFG();          }          }      }  **return** instance;    }  } |

1. **Pros:**
   1. Lazy initialization is possible.
   2. It is also thread safe.
   3. Performance overhead gets reduced because of synchronized keyword.
   4. First time, it can affect performance.
2. **Bill Pugh Singleton Implementation:** Prior to Java5, memory model had a lot of issues and above methods caused failure in certain scenarios in multithreaded environment. So, Bill Pugh suggested a concept of inner static classes to use for singleton.

* JAVA

|  |
| --- |
| // Java code for Bill Pugh Singleton Implementation  **public** **class** GFG  {    **private** GFG()    {      // private constructor    }      // Inner class to provide instance of class  **private** **static** **class** BillPughSingleton    {  **private** **static** **final** GFG INSTANCE = **new** GFG();    }    **public** **static** GFG getInstance()    {  **return** BillPughSingleton.INSTANCE;    }  } |

1. When the singleton class is loaded, inner class is not loaded and hence doesn’t create object when loading the class. Inner class is created only when getInstance() method is called. So it may seem like eager initialization but it is lazy initialization.   
   This is the most widely used approach as it doesn’t use synchronization.

**When to use What**

1. Eager initialization is easy to implement but it may cause resource and CPU time wastage. Use it only if cost of initializing a class is less in terms of resources or your program will always need the instance of class.
2. By using Static block in Eager initialization we can provide exception handling and also can control over instance.
3. Using synchronized we can create singleton class in multi-threading environment also but it can cause slow performance, so we can use Double check locking mechanism.
4. Bill Pugh implementation is most widely used approach for singleton classes. Most developers prefer it because of its simplicity and advantages.

Prerequisite: [Singleton Pattern](https://www.geeksforgeeks.org/singleton-design-pattern-introduction/) In this article, we will see what various concepts can break the singleton property of a class and how to avoid them. There are mainly 3 concepts that can break the singleton property of a class. Let’s discuss them one by one.

**Reflection:** [Reflection](https://www.geeksforgeeks.org/reflection-in-java/) can be caused to destroy singleton property of the singleton class, as shown in the following example:

* Java

|  |
| --- |
| // Java code to explain effect of Reflection  // on Singleton property    **import** java.lang.reflect.Constructor;    // Singleton class  **class** Singleton {      // public instance initialized when loading the class  **public** **static** Singleton instance = **new** Singleton();    **private** Singleton()      {          // private constructor      }  }    **public** **class** GFG {    **public** **static** **void** main(String[] args)      {          Singleton instance1 = Singleton.instance;          Singleton instance2 = **null**;  **try** {              Constructor[] constructors                  = Singleton.**class**.getDeclaredConstructors();  **for** (Constructor constructor : constructors) {                  // Below code will destroy the singleton                  // pattern                  constructor.setAccessible(**true**);                  instance2                      = (Singleton)constructor.newInstance();  **break**;              }          }    **catch** (Exception e) {              e.printStackTrace();          }            System.out.println("instance1.hashCode():- "                             + instance1.hashCode());          System.out.println("instance2.hashCode():- "                             + instance2.hashCode());      }  } |

**Output**

instance1.hashCode():- 1995265320

instance2.hashCode():- 1746572565

After running this class, you will see that hashCodes are different which means, 2 objects of the same class are created and the singleton pattern has been destroyed.

**Overcome reflection issue:** To overcome issues raised by reflection, [enums](https://www.geeksforgeeks.org/enum-in-java/) are used because java ensures internally that the enum value is instantiated only once. Since java Enums are globally accessible, they can be used for singletons. Its only drawback is that it is not flexible i.e it does not allow lazy initialization.

* Java

|  |
| --- |
| // Java program for Enum type singleton  **public** **enum** Singleton {      INSTANCE;  } |

As the constructor for an enum is **package-private** or **private access**, It automatically creates the constants that are defined at the beginning of the enum body. You cannot invoke an enum constructor yourself, so it is not possible for Reflection to utilize it. Hence, reflection can’t break singleton property in the case of enums.

**Serialization:-** [Serialization](https://www.geeksforgeeks.org/serialization-in-java/) can also cause breakage of singleton property of singleton classes. Serialization is used to convert an object of a byte stream and save in a file or send it over a network. Suppose you serialize an object of a singleton class. Then if you de-serialize that object it will create a new instance and hence break the singleton pattern.

* Java

|  |
| --- |
| // Java code to explain effect of  // Serialization on singleton classes  **import** java.io.FileInputStream;  **import** java.io.FileOutputStream;  **import** java.io.ObjectInput;  **import** java.io.ObjectInputStream;  **import** java.io.ObjectOutput;  **import** java.io.ObjectOutputStream;  **import** java.io.Serializable;    **class** Singleton **implements** Serializable {        // public instance initialized      // when loading the class  **public** **static** Singleton instance = **new** Singleton();    **private** Singleton()      {          // private constructor      }  }    **public** **class** GFG {    **public** **static** **void** main(String[] args)      {  **try** {              Singleton instance1 = Singleton.instance;              ObjectOutput out = **new** ObjectOutputStream(  **new** FileOutputStream("file.text"));              out.writeObject(instance1);              out.close();                // deserialize from file to object              ObjectInput in = **new** ObjectInputStream(  **new** FileInputStream("file.text"));                Singleton instance2                  = (Singleton)in.readObject();              in.close();                System.out.println("instance1 hashCode:- "                                 + instance1.hashCode());              System.out.println("instance2 hashCode:- "                                 + instance2.hashCode());          }    **catch** (Exception e) {              e.printStackTrace();          }      }  } |

**Output**

instance1 hashCode:- 1721931908

instance2 hashCode:- 193064360

As you can see, the hashCode of both instances is different, hence there are 2 objects of a singleton class. Thus, the class is no more singleton.

**Overcome serialization issue:-** To overcome this issue, we have to implement the method readResolve() method.

* Java

|  |
| --- |
| // Java code to remove the effect of  // Serialization on singleton classes    **import** java.io.FileInputStream;  **import** java.io.FileOutputStream;  **import** java.io.ObjectInput;  **import** java.io.ObjectInputStream;  **import** java.io.ObjectOutput;  **import** java.io.ObjectOutputStream;  **import** java.io.Serializable;    **class** Singleton **implements** Serializable {        // public instance initialized when loading the class  **public** **static** Singleton instance = **new** Singleton();    **private** Singleton()      {          // private constructor      }        // implement readResolve method  **protected** Object readResolve() { **return** instance; }  }    **public** **class** GFG {    **public** **static** **void** main(String[] args)      {  **try** {              Singleton instance1 = Singleton.instance;              ObjectOutput out = **new** ObjectOutputStream(  **new** FileOutputStream("file.text"));                out.writeObject(instance1);              out.close();                // deserialize from file to object              ObjectInput in = **new** ObjectInputStream(  **new** FileInputStream("file.text"));              Singleton instance2                  = (Singleton)in.readObject();              in.close();                System.out.println("instance1 hashCode:- "                                 + instance1.hashCode());              System.out.println("instance2 hashCode:- "                                 + instance2.hashCode());          }    **catch** (Exception e) {              e.printStackTrace();          }      }  } |

Output:-

instance1 hashCode:- 1550089733

instance2 hashCode:- 1550089733

Above both hashcodes are the same hence no other instance is created.

**Cloning:** [Cloning](https://www.geeksforgeeks.org/clone-method-in-java-2/) is the concept to create duplicate objects. Using clone we can create copy of object. Suppose, we create clone of a singleton object, then it will create a copy that is there are two instances of a singleton class, hence the class is no more singleton.

* Java

|  |
| --- |
| // Java code to explain cloning  // issue with singleton    **class** SuperClass **implements** Cloneable {    **int** i = 10;        @Override  **protected** Object clone()  **throws** CloneNotSupportedException      {  **return** **super**.clone();      }  }    // Singleton class  **class** Singleton **extends** SuperClass {      // public instance initialized when loading the class  **public** **static** Singleton instance = **new** Singleton();    **private** Singleton()      {          // private constructor      }  }    **public** **class** GFG {  **public** **static** **void** main(String[] args)  **throws** CloneNotSupportedException      {          Singleton instance1 = Singleton.instance;          Singleton instance2 = (Singleton)instance1.clone();          System.out.println("instance1 hashCode:- "                             + instance1.hashCode());          System.out.println("instance2 hashCode:- "                             + instance2.hashCode());      }  } |

**Output**

instance1 hashCode:- 746292446

instance2 hashCode:- 989110044

Two different hashCode means there are 2 different objects of the singleton class.

**Overcome Cloning issue:**To overcome this issue, override clone() method and throw an exception from clone method that is CloneNotSupportedException. Now, whenever user will try to create clone of singleton object, it will throw an exception and hence our class remains singleton.

* Java

|  |
| --- |
| // Java code to explain overcome  // cloning issue with singleton    **class** SuperClass **implements** Cloneable {  **int** i = 10;        @Override  **protected** Object clone()  **throws** CloneNotSupportedException      {  **return** **super**.clone();      }  }    // Singleton class  **class** Singleton **extends** SuperClass {        // public instance initialized when loading the class  **public** **static** Singleton instance = **new** Singleton();    **private** Singleton()      {          // private constructor      }        @Override  **protected** Object clone()  **throws** CloneNotSupportedException      {  **throw** **new** CloneNotSupportedException();      }  }    **public** **class** GFG {        // Main driver method  **public** **static** **void** main(String[] args)  **throws** CloneNotSupportedException      {          Singleton instance1 = Singleton.instance;          Singleton instance2 = (Singleton)instance1.clone();            System.out.println("instance1 hashCode:- "                             + instance1.hashCode());          System.out.println("instance2 hashCode:- "                             + instance2.hashCode());      }  } |

Output:-

Exception in thread "main" java.lang.CloneNotSupportedException

at GFG.Singleton.clone(GFG.java:29)

at GFG.GFG.main(GFG.java:38)

Now we have stopped users to create clones of the singleton class. If you don’t want to throw an exception you can also return the same instance from the clone method.

* Java

|  |
| --- |
| // Java code to explain overcome  // cloning issue with singleton    **class** SuperClass **implements** Cloneable {  **int** i = 10;        @Override  **protected** Object clone()  **throws** CloneNotSupportedException      {  **return** **super**.clone();      }  }    // Singleton class  **class** Singleton **extends** SuperClass {      // public instance initialized when loading the class  **public** **static** Singleton instance = **new** Singleton();    **private** Singleton()      {          // private constructor      }        @Override  **protected** Object clone()  **throws** CloneNotSupportedException      {  **return** instance;      }  }    // Main class  **public** **class** GFG {        // Main driver method  **public** **static** **void** main(String[] args)  **throws** CloneNotSupportedException      {          Singleton instance1 = Singleton.instance;          Singleton instance2 = (Singleton)instance1.clone();          System.out.println("instance1 hashCode:- "                             + instance1.hashCode());          System.out.println("instance2 hashCode:- "                             + instance2.hashCode());      }  } |

**Output:**

instance1 hashCode:- 366712642

instance2 hashCode:- 366712642

Now, as hashcode of both instances is the same that means they represent a single instance.

This article is contributed by **Vishal Garg**. If you like GeeksforGeeks and would like to contribute, you can also write an article using [write.geeksforgeeks.org](http://write.geeksforgeeks.org/) or mail your article to review-team@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or if you want to share more information about the topic discussed above.