**Singleton Design pattern**:-

it seems to be a very simple design pattern but when it comes to implementation, it creates a lot of implementation concerns. Also, the implementation of Java Singleton pattern has always been a controversial topic among developers. Here, we will learn about Singleton Design Pattern in Java with all Scenarios, different ways to implement Singleton design pattern and some of the best practices for its usage.

Sometimes, we get the requirement for some classes to have exactly one instance. There are many occasions when we need only one instance of the Object and if we instantiate more than one, we’ll run into all sorts of problems like incorrect program behavior, overuse of resources, or inconsistent results.

Some of the usage of the Singleton pattern are in thread pool, logging, caching, driver objects etc. We can even see the usage of  Singleton design pattern in core java classes also. For example, java.lang.Runtime, java.awt.Desktop

**In order to implement a Singleton pattern, we have different scenarios, but each of them has the following common approach**

* 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 outer world to get the instance of the singleton class.
* **Thread-Safe Singleton (Eager Initialization)**

**We will not have any thread safety issue if we choose to go with eager initialization.**

public class Singleton{  
   
 //**Eager Initialization** //Below instance is guaranteed to be thread safe.  
  
   private static Singleton uniqueInstance= new Singleton();  
  
   private Singleton() { }  
  
   public static Singleton getInstance() {  
  
 //We have already got an instance, so just return it.  
  
    return uniqueInstance;  
  
 }.

Using this approach, we rely on the JVM to create the unique instance of the Singleton when the class is loaded. The JVM guarantees that the instance will be created before any thread accesses the static uniqueInstance variable. Now here is one catch! We have to think about thread safety. In that case, we go with lazy initialization**.**

**---------=========================-------------------------**

There are some examples where Singleton pattern violation situation can be found in a thread-safe environment. However, we will use the hash code comparison technique to verify the equality of two objects. **If two objects are equal**, **they MUST have** **the same hash code**.

## Singleton Violation on Using Reflection

Using reflection, we can set the private constructor to become accessible at runtime as shown in the example below.

import java.lang.reflect.Constructor;  
  
public class SingletonR {  
   
 public static SingletonR instance= new SingletonR();  
   
 private SingletonR() {  
 System.out.println("creating instance.....");  
   
 }  
   
 public static SingletonR getInstance() {  
 return instance;  
 }  
  
 public static void main(String[] args) throws Exception{  
 SingletonR s1 = SingletonR.getInstance();  
 SingletonR s2 = SingletonR.getInstance();  
 System.out.println("Hashcode of Object s1: " +s1.hashCode());  
 System.out.println("Hashcode of Object s2: " +s2.hashCode());  
 //reflection  
 Class clazz = Class.forName("com.dev.dp.creational.singleton.SingletonR");  
 Constructor<SingletonR> ctr= clazz.getDeclaredConstructor();  
 ctr.setAccessible(true);  
 SingletonR s3 = ctr.newInstance();  
 System.out.println("Hashcode of Object s3: " +s3.hashCode());  
  
 }  
   
}

o/p

creating instance.....  
Hashcode of Object s1: 366712642  
Hashcode of Object s2: 366712642  
creating instance.....  
Hashcode of Object s3: 1829164700

### How to Fix Violation

Throw Runtime Exception if someone tries to make instance, in case one instance already exists. Below code will go into the private constructor of the Singleton class accordingly.

**if(instance != null)**

private SingletonR() {  
 System.out.println("creating instance.....");  
 **if(instance != null)** {  
 throw new RuntimeException("Can't create instance. Please use getInsance() to create it.");  
 }   
}

## Singleton Violation on Object Cloning

If we try to make an instance by cloning it, the generated hash code of cloned copy doesn’t **match** with the actual object so it also violates the Singleton principle.

public class SingletonC implements Cloneable{  
   
 public static SingletonC instance= new SingletonC();  
   
 private SingletonC() {  
 System.out.println("creating instance.....");  
 }  
   
 public static SingletonC getInstance() {  
 return instance;  
 }  
  
 protected Object clone() throws CloneNotSupportedException {  
 return super.clone();  
 }  
   
 public static void main(String[] args) throws Exception{  
 SingletonC s1 = SingletonC.getInstance();  
 SingletonC s2 = SingletonC.getInstance();  
 System.out.println("Hashcode of Object s1: " +s1.hashCode());  
 System.out.println("Hashcode of Object s2: " +s2.hashCode());  
   
 SingletonC s3 = (SingletonC)s2.clone();  
 System.out.println("Hashcode of Object s3: " +s3.hashCode());  
  
 }   
}

**Output:**

Output

creating instance.....  
Hashcode of Object s1: 366712642  
Hashcode of Object s2: 366712642  
Hashcode of Object s3: 1829164700

### How to Fix Violation

Throw CloneNotSupportedException from the clone () method if someone tries to make another instance of it. Add below code in clone() method of above SingletonR class.

protected Object clone() throws CloneNotSupportedException {  
 if(instance != null) {  
 throw new CloneNotSupportedException("Can't create instance. Please use getInsance() to create it.");  
 }  
 return super.clone();  
}

## Singleton Violation on Serialization/Deserialization

When we serialize an object and deserialize it again there are different hash code values generated as shown in the example below. Therefore, our Singleton principle breaks in case of object serialization/deserialization also

public class SingletonS implements Serializable{  
   
 public static SingletonS instance= new SingletonS();  
   
 private SingletonS() {  
 System.out.println("creating instance.....");  
 }  
   
 public static SingletonS getInstance() {  
 return instance;  
 }  
  
 public static void main(String[] args) throws Exception{  
 SingletonS s1 = SingletonS.getInstance();  
 SingletonS s2 = SingletonS.getInstance();  
 System.out.println("Hashcode of Object s1: " +s1.hashCode());  
 System.out.println("Hashcode of Object s2: " +s2.hashCode());  
   
 ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("D:/tmp/s2.ser"));  
 oos.writeObject(s2);  
   
 ObjectInputStream ois = new ObjectInputStream(new FileInputStream("D:/tmp/s2.ser"));  
 SingletonS s3= (SingletonS)ois.readObject();  
   
 System.out.println("Hashcode of Object s3: " +s3.hashCode());  
  
 }   
}

### **Output**

Output

creating instance.....  
Hashcode of Object s1: 2055281021  
Hashcode of Object s2: 2055281021  
Hashcode of Object s3: 772777427

### How to Fix Violation

Implement a new readResolve () method in the Singleton class as shown below.

private Object readResolve() {  
 System.out.println("Applying readResolve()......");  
 return SingletonS.getInstance();  
}

## Singleton in Multithreaded Environment

Singleton will work properly in multi-threaded environment only if eager instantiation has been done because in this case instance creation will happen at the time of class loading only. But for Lazy instantiation we will have to take care of multiple things. If we want to delay the instantiation because of cost, we will have to go with lazy.

### Lazy vs Eager Initialization:

Lazy initialization will be beneficial when we want to delay the initialization until it is not needed. On the other hand, if we use eager initialization and if initialization fails, there is no chance to get the instance further while in lazy initialization we may get it in second chance. In Lazy initialization we will not get instance until we call the getInstance method while in eager initialization, it creates instance at the time of class loading itself.

Following code demonstrates the behavior of Singleton instance, when two threads are getting executed by comparing their hash code values. Be careful while running the following code as it will work only in Java 8 and later versions. Moreover, we have used [Method Reference](https://javatechonline.com/method-reference-in-java-8/) in the code.

Singleton in Lazy Initialization

package com.dev.dp.creational.singleton;  
  
import java.util.concurrent.ExecutorService;  
import java.util.concurrent.Executors;  
  
  
public class SingletonT {  
  
 private static SingletonT instance=null; **//lazy initialization**  
   
 private SingletonT(){  
 System.out.println("Creating...");  
 }  
   
 public static SingletonT getInstance(){  
 if (instance == null) {  
 instance = new SingletonT();  
 }  
 return instance;  
 }  
   
 static void useSingleton(){  
 SingletonT singleton = SingletonT.getInstance();  
 System.out.println("Hashcode of Singleton Object: "+singleton.hashCode());  
 }  
   
 public static void main(String[] args) throws Exception {  
 ExecutorService service = Executors.newFixedThreadPool(2);  
 service.submit(SingletonT::useSingleton);  
 service.submit(SingletonT::useSingleton);  
 service.shutdown();  
 }   
}

### **Output**

Output

//Output On running first time  
  
Creating...  
Object : singleton, Hashcode: 1598725298  
Object : singleton, Hashcode: 1598725298  
  
//Output On running second time  
  
Creating...  
Creating...  
Object : singleton, Hashcode: 1598725298  
Object : singleton, Hashcode: 2124340618

### Synchronizing method

As shown by the output of the program, It is noticeable that in multithreaded environment sometimes Singleton principle works while sometimes it violates as multiple threads are trying to create instance. Therefore, we need to synchronize the getInstance () method as shown below.

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

### Synchronizing block of code

After applying synchronize keyword in the getInstance () method the program may execute properly without any issue, but in Java instead of synchronizing whole method we can synchronize only the block of code which is affected while creating instance to escape the extra overhead as below.

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

### Double Checked Locking

double-checked locking, we first check to see if an instance is created, and if not, then we synchronize. This way, we only synchronize the first time through, just what we want.

public static SingletonT getInstance(){  
 if (instance == null) { //check1  
 synchronized (SingletonT.class) {  
 if (instance == null) { //check2  
 instance = new SingletonT();  
 }  
 }  
 }  
 return instance;  
}

### Using volatile keyword

To address above situation, use **volatile** keyword at the time of instance declaration. Value of volatile variable will be published only when the change completes. Change to write operation happens before read operation in volatile variable. In fact, all threads will see the same value of variable.

Use of volatile Keyword

private static volatile SingletonT instance=null; //lazy initialization

If performance is an issue in using getInstance() method, then this method of implementing the Singleton can drastically reduce the overhead.

## What is the purpose of creating a Singleton class?

The main objective of creating a Singleton class is to restrict the creation of number of objects. Moreover, By using the Singleton class, the object won’t be created each time a new request is made. Instead, a single object will be used repeatedly. This is the reason the Singleton pattern in Java is mostly used with multi-threaded and database applications. For example, we use the concept of singleton class while creating a database connection. In this case, we restrict the memory wastage by not creating multiple database connections.