How to prevent Singleton Pattern from Reflection, Serialization and Cloning?

Prerequisite: Singleton Pattern

In this article, we will see that what are various concepts which can break singleton property of a class and how to avoid them. There are mainly 3 concepts which can break singleton property of a class. Let's discuss them one by one.

1. **Reflection:** Reflection can be caused to destroy singleton property of singleton class, as shown in following example:

```
// 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());
        }
}
```

Run on IDE

```
Output:-
instance1.hashCode():- 366712642
instance2.hashCode():- 1829164700
```

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

Overcome reflection issue: To overcome issue raised by reflection, enums are used because java ensures internally that 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 program for Enum type singleton
public enum GFG
{
  INSTANCE;
}
```

Run on IDE

As enums don't have any constructor so it is not possible for Reflection to utilize it. Enums have their by-default constructor, we can't invoke them by ourself. **JVM handles the creation and invocation of enum constructors internally.** As enums don't give their constructor definition to the program, it is not possible for us to access them by Reflection also. Hence, reflection can't break singleton property in case of enums.

2. **Serialization:-** Serialization can also cause breakage of singleton property of singleton classes. Serialization is used to convert an object of byte stream and save in a file or send 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 code to explain effect of
// Serilization 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();

            // deserailize 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();
        }
    }
}
```

Run on IDE

```
Output:-
instance1 hashCode:- 1550089733
instance2 hashCode:- 865113938
```

As you can see, 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 method readResolve() method.

```
// 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();

            // deserailize 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();
        }
    }
}
```

Run on IDE

```
Output:-
instance1 hashCode:- 1550089733
instance2 hashCode:- 1550089733
```

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

3. **Cloning:** Cloning is a concept to create duplicate objects. Using clone we can create copy of object. Suppose, we ceate clone of a singleton object, then it wil create a copy that is there are two instances of a singleton class, hence the class is no more singleton.

```
// 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 CloneNotSupportedExc...on
  {
    Singleton instance1 = Singleton.instance;
```

```
    Singleton instance2 = (Singleton) instance1.clone();
    System.out.println("instance1 hashCode:- "
                       + instance1.hashCode());
    System.out.println("instance2 hashCode:- "
                       + instance2.hashCode());
  }
}
```

Run on IDE

```
Output :-
instance1 hashCode:- 366712642
instance2 hashCode:- 1829164700
```

Two different hashCode means there are 2 different objects of 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 exception and hence our class remains singleton.

```
// 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
{
  public static void main(String[] args) throws CloneNotSupportedExc on
  {
    Singleton instance1 = Singleton.instance;
    Singleton instance2 = (Singleton) instance1.clone();
```

```
    System.out.println("instance1 hashCode:- "
                       + instance1.hashCode());
    System.out.println("instance2 hashCode:- "
                       + instance2.hashCode());
  }
}
```

Run on IDE

```
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 user to create clone of singleton class. If you don;t want to throw exception you can also return the same instance from clone method.

```
// 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;
  }
}

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());
```

```
    }
}
```

Run on IDE

```
Output:-
instance1 hashCode:- 366712642
instance2 hashCode:- 366712642
```

Now, as hashcode of both the instances is 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 contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

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

Design Pattern Java

Login to Improve this Article

Please write to us at contribute@geeksforgeeks.org to report any issue with the above content.

Recommended Posts:

Decorator Pattern | Set 1 (Background)
Java Singleton Design Pattern Practices with Examples
How to design a parking lot using object-oriented principles?
Singleton Design Pattern | Introduction
Reflection in Java
Dependency Inversion Principle (SOLID)
Design Video Sharing System Like Youtube
Design Scalable System like Foursquare
Design Scalable System like Instagram
Mediator Design Pattern

(Login to Rate)

3.3 Average Difficulty : **3.3/5.0**
Based on **6** vote(s)

Basic Easy Medium Hard Expert

Add to TODO List

Mark as DONE

Writing code in comment? Please use ide.geeksforgeeks.org, generate link and share the link here.

Load Comments

Share this post!

A computer science portal for geeks

710-B, Advant Navis Business Park,
Sector-142, Noida, Uttar Pradesh - 201305
feedback@geeksforgeeks.org