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

geeksforgeeks.org/prevent-singleton-pattern-reflection-serialization-cloning

May 18, 2017

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:

```
import java.lang.reflect.Constructor;
class Singleton
{
public static Singleton instance = new Singleton();
private Singleton()
{
}
}
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)
{
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():- 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.

```
public enum GFG
{
INSTANCE;
}
```

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.

```
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 static Singleton instance = new Singleton();
private Singleton()
{
}
}
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();
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:- 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.

```
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 static Singleton instance = new Singleton();
private Singleton()
{
}
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();
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 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.

```
class SuperClass implements Cloneable
{
int i = 10 ;
@Override
protected Object clone() throws CloneNotSupportedException
{
return super .clone();
}
}
class Singleton extends SuperClass
{
public static Singleton instance = new Singleton();
private Singleton()
{
}
}
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:- 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.

```
class SuperClass implements Cloneable
{
int i = 10 ;
@Override
protected Object clone() throws CloneNotSupportedException
{
return super .clone();
}
}
class Singleton extends SuperClass
{
public static Singleton instance = new Singleton();
private Singleton()
{
}
@Override
protected Object clone() throws CloneNotSupportedException
{
throw new CloneNotSupportedException();
}
}
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:-
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.

```
class SuperClass implements Cloneable
{
int i = 10 ;
@Override
protected Object clone() throws CloneNotSupportedException
{
return super .clone();
}
}
class Singleton extends SuperClass
{
public static Singleton instance = new Singleton();
private Singleton()
{
}
@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());
}
}
```

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

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