Singleton Design Pattern

GOF : **Ensure a class only has one instance, and provide a global point of access to it.**

Here we will discuss Singleton Design Concept on the following aspects.

**1. Normal way of creating Singleton Design Class**

**2. Singleton Class in case Reflection and Introspection**

**3. Singleton Class using Security Manager**

**4. Singleton Class in case of Serialization**

**5. Singleton Class in case of Multi threading**

**6. Singleton Class using Double Checked Locking mechanism**

**7. Singleton Class using Initialization on Demand mechanism(Bill Pugh's Solution)**

**8. Singleton Class using Java's Enum Concept suggested by Joshua Bloch**

**9. Cracking of Singleton using a nasty approach**

A typical fragile Singleton design pattern is given below.

package com.ddlab.rnd.patterns;

public class SingletonType1

{

private static SingletonType1 instance = null;

private SingletonType1()

{

super();

}

public static SingletonType1 getInstance()

{

if( instance == null )

instance = new SingletonType1();

return instance;

}

}

To use and implement we have to write the following code.

SingletonType1 instance = SingletonType1.getInstance();

It is fine and seems to be correct. If you write the above code 10 times or more you will get the same instance. If you do not override the hashcode() and toString() method , you will get the same evertime when you invoke. Let us consider another test.Create the instances of the above class by invoking "SingletonType1.getInstance()" the code and put all the instances in a Set. As you know that Set does not allow duplicates. So at the end if you are getting the size of the set 1, then it is the correct implementation. But the above design can be broken and we can create multiple instances of the Singleton design pattern using reflection.

**Note: You can use java's Reflection or Introspection utility to invoke the instance of Singleton design class if there is no adequete protection .**

Let us see the code below.

package com.ddlab.rnd.patterns;

import java.lang.reflect.Constructor;

import java.lang.reflect.Field;

import java.lang.reflect.Method;

public class TestSingletonType1 {

public void createMultiInstances()

{

System.out.println("\n\*\*\*\*\* MULTIPLE INSTANCES FROM SINGLETON \*\*\*\*\*\*\*\*\*\*\*\n");

/\*

\* Using Reflection you can break singleton

\*/

try {

**Class clazz = Class.forName("com.ddlab.rnd.patterns.SingletonType1");**

**Constructor constructor = clazz.getDeclaredConstructors()[0];**

**constructor.setAccessible(true);**

**SingletonType1 instance1 = (SingletonType1)constructor.newInstance(null);**

SingletonType1 instance2 = (SingletonType1)constructor.newInstance(null);

SingletonType1 instance3 = (SingletonType1)constructor.newInstance(null);

System.out.printf( "%-15s %-15s %n", "SERIAL NO", "MULTI INSTANCES");

System.out.printf( "%-15s %-15s %n", "---------", "---------------");

System.out.format("%-15s %-15s %n", "INSTANCE 1 ",instance1);

System.out.format("%-15s %-15s %n", "INSTANCE 2 ",instance2);

System.out.format("%-15s %-15s %n", "INSTANCE 3 ",instance3);

}

catch (Exception e) {

e.printStackTrace();

}

}

public void createMultiInstances1() {

System.out.println("\n\*\*\*\*\*\*\*\* MULTIPLE INSTANCES FROM SINGLETON \*\*\*\*\*\*\*\*\n");

/\*

\* Using Reflection you can break singleton

\*/

try

{

Class clazz = Class.forName("com.ddlab.rnd.patterns.SingletonType1");

Method method = clazz.getDeclaredMethods()[0];

Field field = clazz.getDeclaredFields()[0];

field.setAccessible(true);

SingletonType1 instance1 = (SingletonType1)method.invoke(clazz, null);

field.set(clazz, null);

SingletonType1 instance2 = (SingletonType1)method.invoke(clazz, null);

field.set(clazz, null);

SingletonType1 instance3 = (SingletonType1)method.invoke(clazz, null);

System.out.printf( "%-15s %-15s %n", "SERIAL NO", "MULTI INSTANCES");

System.out.printf( "%-15s %-15s %n", "---------", "---------------");

System.out.format("%-15s %-15s %n", "INSTANCE 1 ",instance1);

System.out.format("%-15s %-15s %n", "INSTANCE 2 ",instance2);

System.out.format("%-15s %-15s %n", "INSTANCE 3 ",instance3);

}

catch (Exception e) {

e.printStackTrace();

}

}

public void createInstances()

{

System.out.println("\n\*\*\*\*\*\*\*\* SINGLE INSTANCES FROM SINGLETON \*\*\*\*\*\*\*\*\*\*\n");

SingletonType1 instance1 = SingletonType1.getInstance();

SingletonType1 instance2 = SingletonType1.getInstance();

SingletonType1 instance3 = SingletonType1.getInstance();

System.out.printf( "%-15s %-15s %n", "SERIAL NO", "INSTANCES");

System.out.printf( "%-15s %-15s %n", "---------", "----------");

System.out.format("%-15s %-15s %n", "INSTANCE 1 ",instance1);

System.out.format("%-15s %-15s %n", "INSTANCE 2 ",instance2);

System.out.format("%-15s %-15s %n", "INSTANCE 3 ",instance3);

}

public static void main(String[] args)

{

new TestSingletonType1().createInstances();

new TestSingletonType1().createMultiInstances();

new TestSingletonType1().createMultiInstances1();

}

}

If you run the above program you will be able to see the many instances of the defined singleton class.

However we got to know to know the private constructor approach of Singleton can be broken using reflection. In the above case we are able to create the instance of a class having private constructor and also we are able to access the private field. We are also able to invoke the getInstance() method by accessing the private field.

**If you want to use Java's Introspection utility, the code is given below.**

public void doIt()

{

try

{

MethodDescriptor[] mds = Introspector.getBeanInfo(SingletonType2.class).getMethodDescriptors();

for( MethodDescriptor methodDescs : mds )

{

Method method = methodDescs.getMethod();

if( method.getName().equals("getInstance"))

{

SingletonType2 instance1 = (SingletonType2)method.invoke(SingletonType2.class, null);

System.out.println("First Instance :::"+instance1);

Field underLyingField = method.getDeclaringClass().getDeclaredField("instance");

underLyingField.setAccessible(true);

underLyingField.set(Introspector.getBeanInfo(SingletonType2.class), null);

SingletonType2 instance2 = (SingletonType2)method.invoke(SingletonType2.class, null);

System.out.println("Second Instance :::"+instance2);

}

}

}

catch (Exception e)

{

e.printStackTrace();

}

}

To gurad against the reflection and Introspection attack, we can write a protective code not the robust one. Now let us see how to write a better code so that other developers will not be able to break using reflection.

package com.ddlab.rnd.patterns;

import java.lang.reflect.ReflectPermission;

import java.security.Permission;

public class SingletonType2

{

static

{

getInstance();

}

private static SingletonType2 instance = null;

**private SingletonType2()**

**{**

**super();**

**//Add the following piece of code so that it can not be invoked using relection**

**System.setSecurityManager(new SecurityManager() {**

**@Override**

**public void checkPermission(Permission perm)**

**{**

**if (perm instanceof ReflectPermission )**

**{**

**System.out.println("\nYes I will not allow you to create the instance using Reflection...\n");**

**throw new SecurityException();**

**}**

**else**

**{**

**//Do nothing**

**}**

**}**

**});**

**}**

public static SingletonType2 getInstance() {

if( instance == null )

instance = new SingletonType2();

return instance;

}

}

Now it is true, your reflection attack will not impact the above code. You will get Exception if you use reflection to create another instance. The following piece of code will help you against the reflection attack.

**System.setSecurityManager(new SecurityManager() {**

**@Override**

**public void checkPermission(Permission perm) {**

**if (perm instanceof ReflectPermission ) {**

**System.out.println("\nYes I will not allow you to create the instance using Reflection...\n");**

**throw new SecurityException();**

**}**

**else {**

**//Do nothing**

**}**

**}**

**});**

However you can use java's introspection utility to create the instances of the Singleton Class, but you will always get the same instance.

**What about serialization ?**

We can not serialize the class SingletonType2 as it does not implement Serializable interface and also we do not allow reflection. However we can not serialize a class which does not implement Serilizable interface. However sometimes it is required to persist the Singleton object for a day. In this case we have to implement Serilizable interface in singleton class.

Let us see the Singleton class with Serilizable interface.

package com.ddlab.rnd.patterns;

import java.io.Serializable;

public class SingletonType11 implements Serializable

{

private static final long serialVersionUID = -4137189065490862968L;

private static SingletonType11 instance = null;

private SingletonType11()

{

super();

}

public static SingletonType11 getInstance()

{

if( instance == null )

instance = new SingletonType11();

return instance;

}

}

Let us see how we can again break the concept of Singleton through object serilization.

Let use write a small class like this.

package com.ddlab.rnd.patterns;

import java.io.Serializable;

public class BreakSingleton implements Serializable

{

private static final long serialVersionUID = 5904306999023481976L;

private SingletonType11 instance2 = SingletonType11.getInstance();

public SingletonType11 getInstance2() {

return instance2;

}

public void setInstance1(SingletonType11 instance2) {

this.instance2 = instance2;

}

}

Let us see the test harness class for the above.

package com.ddlab.rnd.patterns;

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.InputStream;

import java.io.ObjectInputStream;

import java.io.ObjectOutputStream;

import java.io.OutputStream;

public class TestBreakSingleton

{

public static void main(String[] args) throws Exception

{

BreakSingleton bs = new BreakSingleton();

OutputStream out = new FileOutputStream("data/a.ser");

ObjectOutputStream oout = new ObjectOutputStream(out);

oout.writeObject(bs);

oout.flush();

oout.close();

out.flush();

out.close();

InputStream in = new FileInputStream("data/a.ser");

ObjectInputStream oin = new ObjectInputStream(in);

BreakSingleton bs1 = (BreakSingleton)oin.readObject();

oin.close();

in.close();

System.out.println("Instance from Serialization :::"+bs1.getInstance2());

System.out.println("Normal Instance :::"+SingletonType11.getInstance());

InputStream in1 = new FileInputStream("data/a.ser");

ObjectInputStream oin1 = new ObjectInputStream(in1);

BreakSingleton bs2 = (BreakSingleton)oin1.readObject();

oin1.close();

in1.close();

System.out.println("Another Instance from Serialization :::"+bs2.getInstance2());

}

}

If you run the above program, you will get following type of output.

Instance from Serialization :::com.ddlab.rnd.patterns.SingletonType11@2586db54

Normal Instance :::com.ddlab.rnd.patterns.SingletonType11@12276af2

Another Instance from Serialization :::com.ddlab.rnd.patterns.SingletonType11@38a97b0b

So now you got three different instances of a Singleton class. Now let us see the modified singleton java class so that we will be able to serialize the object and at any point of time we will get the consistent singleton class.

package com.ddlab.rnd.patterns;

import java.io.ObjectStreamException;

import java.io.Serializable;

public class SingletonType11 implements Serializable

{

private static final long serialVersionUID = -4137189065490862968L;

private static SingletonType11 instance = null;

private SingletonType11()

{

super();

}

public static SingletonType11 getInstance()

{

if( instance == null )

instance = new SingletonType11();

return instance;

}

private Object readResolve() throws ObjectStreamException

{

return instance;

}

private Object writeReplace() throws ObjectStreamException

{

return instance;

}

}

However still we can create multiple instance using Reflection and we can not prevent reflection as we want to serialize the object. We are giving an opertunity to hacker to create multiple instance.

Let us see the use of thread in case of Singleton class.

Let us consider the same class,

package com.ddlab.rnd.patterns;

public class SingletonType1

{

private static SingletonType1 instance = null;

private SingletonType1()

{

super();

}

public static SingletonType1 getInstance()

{

if( instance == null )

instance = new SingletonType1();

return instance;

}

}

The above approach is called lazy-initialization. In case of multi threading we can get multiple instances. Let see the code below.

package com.ddlab.rnd.patterns;

import java.util.Collections;

import java.util.HashSet;

import java.util.Set;

class Thread1 extends Thread

{

@Override

public void run()

{

SingletonType1 instance = SingletonType1.getInstance();

// System.out.println("In Thread 1 - Singleton Instance ---->"+instance);

TestSingletonType1\_Thread.singletonSet.add(instance);

}

}

class Thread2 extends Thread

{

@Override

public void run()

{

SingletonType1 instance = SingletonType1.getInstance();

// System.out.println("In Thread 2 - Singleton Instance ---->"+instance);

TestSingletonType1\_Thread.singletonSet.add(instance);

}

}

public class TestSingletonType1\_Thread

{

private static Set<SingletonType1> singletonSet1 = new HashSet<SingletonType1>();

public static Set<SingletonType1> singletonSet = Collections.synchronizedSet(singletonSet1);

public static void main(String[] args)

{

//Singleton concept is broken here

for( int i = 0 ; i < 100 ; i++ )

{

new Thread1().start();

new Thread2().start();

if( singletonSet.size() > 1 )

break;

else

continue;

}

System.out.println(singletonSet);

}

}

If you run the above program many times, you will get the different instances of the Singleton class.

Upon running the program, you may get a similar type result like this.

[com.ddlab.rnd.patterns.SingletonType1@60723d7c, com.ddlab.rnd.patterns.SingletonType1@6d9efb05, com.ddlab.rnd.patterns.SingletonType1@8dd20f6]

So what to do ? Can we declare volatile variable, let see now.

Let us have the modified program.

package com.ddlab.rnd.patterns;

public class SingletonType1

{

private static volatile SingletonType1 instance = null;

private SingletonType1()

{

super();

}

public static SingletonType1 getInstance()

{

if( instance == null )

instance = new SingletonType1();

return instance;

}

}

Upon running the program , you may get like this.

[com.ddlab.rnd.patterns.SingletonType1@3f0ef90c, com.ddlab.rnd.patterns.SingletonType1@2e471e30]

However use of volatile does not serve our purpose here.

Still the same problem, can we use synchronized method, yes we can do it.

package com.ddlab.rnd.patterns;

public class SingletonType1

{

private static volatile SingletonType1 instance = null;

private SingletonType1()

{

super();

}

public static synchronized SingletonType1 getInstance()

{

if( instance == null )

instance = new SingletonType1();

return instance;

}

}

But there will be a performance issue. However, when you analyze it you realize that synchronization is required only

for the first invocation of the method. Subsequent invocations do not require synchronization.

In order to increase the efficiency of the above program, let use modify the above program in a different manner.

package com.ddlab.rnd.patterns;

public class SingletonType1

{

private static volatile SingletonType1 instance = null;

private SingletonType1()

{

super();

}

public static SingletonType1 getInstance()

{

if (instance == null)

{

synchronized(SingletonType1.class) {

instance = new SingletonType1();

}

}

return instance;

}

}

The above program looks ok. But still there is a very big problem.

Two threads can get inside of the if statement concurrently when instance is null.

Then, one thread enters the synchronized block to initialize instance,

while the other is blocked. When the first thread exits the synchronized block,

the waiting thread enters and creates another Singleton object.

Note that when the second thread enters the synchronized block, it does not check to see if instance is non-null.

**Double-Checked Locking Idiom in Singleton Design Pattern**

Now let us consider another concept called "double-checked locking" which seems to be famous for a set of developers. Many developers apply in many cases and think to be most robust form of singleton. In software engineering, double-checked locking (also known as "double-checked locking optimization"[1]) is a software design pattern used to reduce the overhead of acquiring a lock by first testing the locking criterion (the "lock hint") without actually acquiring the lock. Only if the locking criterion check indicates that locking is required does the actual locking logic proceed.

The structure is given below.

public static SingletonType1 getInstance()

{

if (instance == null)

{

synchronized(SingletonType1.class) // Mark - 1

{

if (instance == null) // Mark - 2

instance = new SingletonType1(); // Mark - 3

}

}

return instance;

}

The theory behind double-checked locking is that the second check at // Mark - 2 makes it impossible for two different Singleton objects to be created. It may be true for a single threaded application. What about fine grained multi-threaded application ?

Thread 1 enters the getInstance() method.

Thread 1 enters the synchronized block at // Mark - 1 because instance is null.

Thread 1 is preempted by thread 2

Thread 2 enters the getInstance() method.

Thread 2 attempts to acquire the lock at // Mark - 1 because instance is still null. However, because thread 1 holds the lock, thread 2 blocks at // Mark - 1.

Thread 2 is preempted by thread 1.

Thread 1 executes and because instance is still null at // Mark - 2, creates a Singleton object and assigns its reference to instance.

Thread 1 exits the synchronized block and returns instance from the getInstance() method.

Thread 1 is preempted by thread 2.

Thread 2 acquires the lock at // Mark - 1 and checks to see if instance is null.

Because instance is non-null, a second Singleton object is not created and the one created by thread 1 is returned. The theory behind double-checked locking is perfect. Unfortunately, reality is entirely different. The problem with double-checked locking is that there is no guarantee it will work on single or multi-processor machines. The issue of the failure of double-checked locking is not due to implementation bugs in JVMs but to the current Java platform memory model. The memory model allows what is known as "out-of-order writes" and is a prime reason why this idiom fails. However the concept "out-of-order writes" is beyond the scope of our discussion.

The bottom line is that double-checked locking, in whatever form, should not be used because you cannot guarantee that it will work on any JVM implementation.

As we have seen that although "double-checked-locking" may work but it may fail unexpectedly.

**The solution of Bill Pugh**

University of Maryland Computer Science researcher Bill Pugh has written about the code issues

underlying the Singleton pattern when implemented in Java.[9] Pugh's efforts on the "Double-checked locking" idiom led to changes in the Java memory model in Java 5 and to what is generally regarded as the standard method to implement Singletons in Java. The technique known as the initialization on demand holder idiom, is as lazy as possible, and works in all known versions of Java. It takes advantage of language guarantees about class initialization, and will therefore work correctly in all Java-compliant compilers and virtual machines. The nested class is referenced no earlier (and therefore loaded no earlier by the class loader) than the moment that getInstance() is called. Thus, this solution is thread-safe without requiring special language constructs (i.e. volatile or synchronized).

**public class Singleton**

**{**

**// Private constructor prevents instantiation from other classes**

**private Singleton() { }**

**/\*\***

**\* SingletonHolder is loaded on the first execution of Singleton.getInstance()**

**\* or the first access to SingletonHolder.INSTANCE, not before.**

**\*/**

**private static class SingletonHolder {**

**public static final Singleton INSTANCE = new Singleton();**

**}**

**public static Singleton getInstance() {**

**return SingletonHolder.INSTANCE;**

**}**

**}**

The above is called "Initialization-on-demand holder idiom".

Let us understand the concept.

**public class Something {**

**private Something() {**

**}**

**private static class LazyHolder {**

**public static final Something INSTANCE = new Something();**

**}**

**public static Something getInstance() {**

**return LazyHolder.INSTANCE;**

**}**

**}**

How it works

The implementation relies on the well-specified initialization phase of execution within the Java Virtual Machine (JVM); see section 12.4 of Java Language Specification (JLS) for details.

When the class Something is loaded by the JVM, the class goes through initialization. Since the class does not have any static variables to initialize, the initialization completes trivially. The static class definition LazyHolder within it is not initialized until the JVM determines that LazyHolder must be executed. The static class LazyHolder is only executed when the static method getInstance is invoked on the class Something, and the first time this happens the JVM will load and initialize the LazyHolder class. The initialization of the LazyHolder class results in static variable INSTANCE being initialized by executing the (private) constructor for the outer class Something. Since the class initialization phase is guaranteed by the JLS to be serial, i.e., non-concurrent, no further synchronization is required in the static getInstance method during loading and initialization. And since the initialization phase writes the static variable INSTANCE in a serial operation, all subsequent concurrent invocations of the getInstance will return the same correctly initialized INSTANCE without incurring any additional synchronization overhead. However using the concept of "Initialization-on-demand holder idiom" pattern we can achieve a thread-safe singleton construct. Again the question arises, can we break reflectively. Yes we can break the above concept using java reflection mechnism which I have already mentioned. Now question comes , is there any other approach to build a proper singleton design approach. Yes there is another one as suggested by

Joshua Bloch(Chief Technical Architect, Google Innovation Lab and Author of famous Book Effective Java).

**package com.ddlab.rnd.patterns;**

**public enum SingletonType3**

**{**

**INSTANCE;**

**public void doSomething(String arg)**

**{**

**//... perform operation here ...**

**}**

**}**

This is the only robust approach of creating a singleton class which is serializable and completely thread-safe as enums by default thread-safe.

What about reflection, using our above normal approach of reflection, you can not break singleton object as it has no constructor. **What about serialization**, you will be able to serialize it but you will get the same instance every time. So finally we have to imbibe this modern

approach of creation of Singleton design class. However many developers do not know about it.

Still we can not break the above approach of singleton, but we can crack the above by writing a code to create more than one instance.

The code is give below do not use the below code for your commercial product. It is a nasty approach of breaking singleton.

package com.ddlab.rnd.patterns;

import java.lang.reflect.Constructor;

import java.lang.reflect.Method;

import java.util.HashSet;

import java.util.Set;

import sun.reflect.ConstructorAccessor;

public class CrackEnumSingleton

{

public static void main(String[] args)

{

Set set = new HashSet();

try

{

SingletonType3 firstInstance = SingletonType3.INSTANCE;

System.out.println(firstInstance.getClass() + " " + firstInstance + " = " + System.identityHashCode(firstInstance));

set.add(firstInstance);

Constructor constructor = SingletonType3.class.getDeclaredConstructors()[0];

Method acquire = constructor.getClass().getDeclaredMethod("acquireConstructorAccessor");//"acquireConstructorAccessor" fields for cracking

acquire.setAccessible(true);

acquire.invoke(constructor);

Method get = constructor.getClass().getDeclaredMethod("getConstructorAccessor");//"getConstructorAccessor" fields for cracking

get.setAccessible(true);

ConstructorAccessor invoke = (ConstructorAccessor) get.invoke(constructor);

Object secondInstance = invoke.newInstance(new Object[] {null,1});

System.out.println(secondInstance.getClass() + " " + secondInstance + " = " + System.identityHashCode(secondInstance));

set.add(secondInstance);

System.out.println("Total No of Singletons :::"+set.size());

}

catch (Exception e)

{

e.printStackTrace();

}

}

}