Singlton Design Pattern in Deep

Traditional Methods of Making Singletons

There are several popular methods for making singletons.

3. and make it thread safe with synchronized and voletile (read from main memory not cpu cache ) :

ایراد روش دوم اینکه اگه همزمان دو تا thread همزمان بیان داخل و بخوان ابجکت ازشون ساخته بشه حتی با وجود کسری از ثانیه ، ازشون دوتا ابجکت مستقل با هش کد های متفاوت از نخ ها ایجاد میکنه چراکه اصل دیزاین پترن سینگلتون را زیر سوال میبره و دیگه thread های مختلف با هش کدهای یکسان نداریم در نتیجه این ThreadSafe نیست پس این روش قابل قبول نیس

برای حل این مشکل در حالت سوم میایم و از مکانیسم و متدهایی که جاوا در اختیار مون گذاشته استفاده میکنیم یکی از این مکانیزم ها استفاده از voletile است **که میاد چیکارمیکنه؟؟ ببینید هرمتدی که ساخته میشه میاد روی سی پی یو یک کش براش ایجاد میشه در این حالت** voletile میگه که نیا از cpu cache متد را بخون بیا و از main memory مقدار متد ایجاد شده را بخون (منظور متد getInstance())

public class Singleton {  
  
 private static voltile Singleton INSTANCE = null;  
  
 private Singleton() {}  
  
 public static Singleton getInstance() {  
 if (INSTANCE == null) {  
 synchronized (Singleton.class) {  
 if (INSTANCE == null) {  
 INSTANCE = new Singleton();  
 }  
 }  
 }  
 return INSTANCE;  
 }  
  
}

**تا همیشه مقداری را که میخونی تازه باشه و در ادامه با مکانیزم synchronized بیا و اونو همزمان کن و نخ ها را با هش کد مشابهه ایجادش کن تا مشکل مرحله دو حل بشه ؛**

**بعبارتی میاد با ورود نخ اول به متد ، یک ابجکت ازش میسازه و اجازه نمیده همزمان با اون نخ شماره دو هم ورود کنه اولی که کارش تموم شد دومی بعد اولی ورود پیدا میکنه حالا دراین حالت چک میکنه اگه نمونه ای از آبجکت ساخته شده بود دیگه برای بار دوم مقدار جدید نمیسازه new نمیکنه بلکه برای نخ دومی هم همون ابجکت نخ دوم درنظر میگیره پس تا اینجا مشکل حل میشه.و اصطلاحا اومد و** ThreadSafe اش کرد اما بازم ایراد داره ...

**اما باز هم یه مشکل وجود داره اینکه میگه تو اومدی و فقط جلو یه مشکل را که new object کردنه را گرفتی اما ما میتونیم به روشهای مختلف یک کلاس را داشته باشیم و ازش استفاده کنیم بعبارتی از حالت های دیگه کلاس استفاده کنیم مثل موارد زیر که دانستن اون خیلی مهمه اینا مارو به برنامه نویس حرفه ای تبدیل میکنه::**

1. Serialization and deserialization
2. Reflection
3. Clone

**این سه خاصیت و حالت ممکن که حتی اگه جلو new کردن این کلاس هم بگیریم بازم ما امکان داشتن ابجکت های جدید از کلاس مدنظرمون را داریم و باز در این شرایط شرط داشتن دیزاین پترن سینگلتون و داشتن ابجکت های مختلف با یک هش کد ثابت در کل برنامه بهم میخوره برای حل این مسئله به سراغ حالت چهارم میریم**

Static fields are initialized at class loading time. Therefore, in Method 1 , singleton instances are created even in a case we don’t use them at runtime. This is not a problem as long as the singleton object is not too big and creating the instance is not too expensive. Method 2 avoids this problem with lazy initialization. In Method 3, the instance is created when we access the singleton object for the first time. Fine-grained synchronization is used to ensure that no more than one object is created with multiple concurrent threads .

All of the above methods work fine until you are not doing serialization and deserialization with a singleton class. Let’s think again: How did we achieve the singleton behavior in above methods? It was done by making the constructor private and making the constructor inaccessible for creating new instances of the class. But isn’t there any other ways to create an instance of a class other than the constructor? The answer isn’t no. There are some other advanced methods.

1. Serialization and deserialization
2. Reflection
3. Clone

4. **Problems With Serialization and Deserialization**

In order to serialize the above singleton classes, we must implement those classes with a Serializable interface. But doing that is not enough. When deserializing a class, new instances are created. Now it doesn't matter the constructor is private or not. Now there can be more than one instance of the same singleton class inside the JVM, violating the singleton property.

در این حالت میاد و public class Singleton implements Serializable{

**را پیاده سازی میکنه تا شرط این دیزاین پترن براورده بشه در غیر این صورت میاد و دو تا ابجکت با دو تا هش کد متفاوت ایجاد میکنه**

public class Singleton implements Serializable {  
 private static volatile Singleton *INSTANCE* = new Singleton();

**در این حالت یکی از شرطهاش اینکه حتما بصورت**

**(**new Singleton();

**) Eager**

**باشه و Lazy جواب نمیده**

**در ادامه هر سه حالت برای پیاده سازی سینگلتون مشکل داره در نهایت**

**Rod johnson اومد به دنبال راه حل گشت چرا که باعث میشد کل مشکل فریمورک های اسپرینگ و توسعه اون حل بشه در نهایت اومد و از یه روش خیلی پیشرفته استفاده کرد بنام Enum ها بشکل زیر مشکل را حل کرد ::**

**Singleton With Enum**

public enum SingletonEnum {  
 INSTANCE;  
  
 int value;  
  
 public int getValue() {  
 return value;  
 }  
  
 public void setValue(int value) {  
 this.value = value;  
 }  
}

**چون اینام غیرقابل تغییره حتی با رفلکشن!!!!!**

**Singleton Design Pattern in Deep**

[kiarash shamaii](https://medium.com/@kiarash.shamaii?source=post_page-----34225f6adac4--------------------------------)

·

Following

5 min read

·

2 days ago

2

A singleton is a class that is supposed to have only one instance per JVM. The Same instance of the singleton class is reused by multiple threads. Most often, we use singletons to represent system configurations and window managers, since those instances should be common to all threads and objects within a JVM.

**Traditional Methods of Making Singletons**

There are several popular methods for making singletons.

**Singleton With Public Static Final Field**

1.

public class Singleton {  
  
 public static final Singleton INSTANCE = new Singleton();  
  
 private Singleton() {}  
  
}  
  
  
//test part in other class  
  
public class SingletonTest {  
 public static void main(String[] args) {  
 Singleton object1 = Singleton.getInstance();  
 Singleton object2 = Singleton.getInstance();  
 System.out.println("Hashcode of Object 1 - " + object1.hashCode());  
 System.out.println("Hashcode of Object 2 - " + object2.hashCode());  
 }  
}

and get result like this : Here is output; you can see it has the same hashCode for objectOne and objectTwo:

Hashcode of Object 1 - 1836019240  
Hashcode of Object 2 - 1836019240

2. we can initialized this pattern in lazy form :

public class Singleton {  
  
 private static Singleton INSTANCE = null;  
  
 private Singleton() {}  
  
 public static Singleton getInstance() {  
 if (INSTANCE == null) {  
 INSTANCE = new Singleton();  
 }  
 }  
 return INSTANCE;  
 }  
  
}

3.and make it thread safe with synchronized and voletile (read from main memory not cpu cache ) :

public class Singleton {  
  
 private static voltile Singleton INSTANCE = null;  
  
 private Singleton() {}  
  
 public static Singleton getInstance() {  
 if (INSTANCE == null) {  
 synchronized (Singleton.class) {  
 if (INSTANCE == null) {  
 INSTANCE = new Singleton();  
 }  
 }  
 }  
 return INSTANCE;  
 }  
  
}

Static fields are initialized at class loading time. Therefore, in Method 1 , singleton instances are created even in a case we don’t use them at runtime. This is not a problem as long as the singleton object is not too big and creating the instance is not too expensive. Method 2 avoids this problem with lazy initialization. In Method 3, the instance is created when we access the singleton object for the first time. Fine-grained synchronization is used to ensure that no more than one object is created with multiple concurrent threads .

All of the above methods work fine until you are not doing serialization and deserialization with a singleton class. Let’s think again: How did we achieve the singleton behavior in above methods? It was done by making the constructor private and making the constructor inaccessible for creating new instances of the class. But isn’t there any other ways to create an instance of a class other than the constructor? The answer isn’t no. There are some other advanced methods.

1. Serialization and deserialization
2. Reflection
3. Clone

**Problems With Serialization and Deserialization**

In order to serialize the above singleton classes, we must implement those classes with a Serializable interface. But doing that is not enough. When deserializing a class, new instances are created. Now it doesn't matter the constructor is private or not. Now there can be more than one instance of the same singleton class inside the JVM, violating the singleton property.

public class SerializeDemo implements Serializable {  
  
 public static void main(String[] args) {  
 Singleton singleton = Singleton.INSTANCE;  
 singleton.setValue(1);  
  
 // Serialize  
 try {  
 FileOutputStream fileOut = new FileOutputStream("out.ser");  
 ObjectOutputStream out = new ObjectOutputStream(fileOut);  
 out.writeObject(singleton);  
 out.close();  
 fileOut.close();  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
  
 singleton.setValue(2);  
  
 // Deserialize  
 Singleton singleton2 = null;  
//we can use try with resources   
 try {  
 FileInputStream fileIn = new FileInputStream("out.ser");  
 ObjectInputStream in = new ObjectInputStream(fileIn);  
 singleton2 = (Singleton) in.readObject();  
//this part can pmitted in try with resources  
 in.close();  
 fileIn.close();  
//---  
 } catch (IOException i) {  
 i.printStackTrace();  
 } catch (ClassNotFoundException c) {  
 System.out.println("singletons.SingletonEnum class not found");  
 c.printStackTrace();  
 }  
  
 if (singleton == singleton2) {  
 System.out.println("Two objects are same");  
 } else {  
 System.out.println("Two objects are not same");  
 }  
  
 System.out.println(singleton.getValue());  
 System.out.println(singleton2.getValue());  
  
 }  
  
}

The output for the above code is:

Two objects are not same  
2  
1

*Here, singleton and singleton2 are two different instances having two different values as their field variables. This violates the singleton property. The solution is that we have to implement the readResolve method, which is called when preparing the deserialized object before returning it to the caller. The solution is as follows.*

public class Singleton implements Serializable{  
  
 public static final Singleton INSTANCE = new Singleton();  
  
 private Singleton() {  
 }  
  
 protected Object readResolve() {  
 return INSTANCE;  
 }  
  
}

Now the output for above code will be:

Two objects are same  
2  
2

Now the singleton property is preserved, and only one instance of the singleton class exists within the JVM.

**Problems With Reflection**

An advanced user can change the private access modifier of the constructor to anything they want at runtime using reflection. If this happens, our only mechanism for non instantiability breaks. Let’s see how this can be done.

public class ReflectionDemo {  
  
 public static void main(String[] args) throws Exception {  
 Singleton singleton = Singleton.INSTANCE;  
  
 Constructor constructor = singleton.getClass().getDeclaredConstructor(new Class[0]);  
 constructor.setAccessible(true);  
  
 Singleton singleton2 = (Singleton) constructor.newInstance();  
  
 if (singleton == singleton2) {  
 System.out.println("Two objects are same");  
 } else {  
 System.out.println("Two objects are not same");  
 }  
  
 singleton.setValue(1);  
 singleton2.setValue(2);  
  
 System.out.println(singleton.getValue());  
 System.out.println(singleton2.getValue());  
  
 }  
}

Output:

Two objects are not same  
1  
2

In this way, the non-accessible private constructor becomes accessible and the whole idea of making the class a singleton breaks.

**Cloning Problem:**

Using the “clone" method, we can create a copy of the original object; it's the same thing if we applied clone in the singleton pattern. It will create two instances: one original and another one cloned object. In this case, we will break the Singleton principle, as shown in the below code.

Implement the “ Cloneable" interface and override the clone method in the above Singleton class.

public class Singleton implements Cloneable {  
  
 private static Singleton INSTANCE = null;  
  
 private Singleton() {}  
  
 public static Singleton getInstance() {  
 if (INSTANCE == null) {  
 INSTANCE = new Singleton();  
 }  
 }  
 return INSTANCE;  
 }  
 @Override  
 protected Object clone() throws CloneNotSupportedException {  
 return super.clone();  
 }  
  
}

test this situation :

public class CloningSingleton {  
 public static void main(String[] args) throws CloneNotSupportedException, Exception {  
 Singleton instanceOne = Singleton.getInstance();  
 Singleton instanceTwo = (Singleton) instanceOne.clone();  
 System.out.println("hashCode of instance 1 - " + instanceOne.hashCode());  
 System.out.println("hashCode of instance 2 - " + instanceTwo.hashCode());  
 }  
  
}

Output :

hashCode of instance 1 - 1836019240  
hashCode of instance 2 - 325040804

If we see the above output, two instances have different hashcodes. This means these instances are not the same.

**Prevent Singleton Pattern From Cloning**

In the above code, it breaks the Singleton principle, i. e created two instances. To overcome the above issue, we need to implement/override the clone() method and throw an exception CloneNotSupportedException from the clone method. If anyone tries to create a clone object of Singleton, it will throw an exception, as shown in the below code.

public class SingletonWithoutClone implements Cloneable {  
  
 private static Singleton INSTANCE = null;  
  
 private Singleton() {}  
  
 public static Singleton getInstance() {  
 if (INSTANCE == null) {  
 INSTANCE = new Singleton();  
 }  
 }  
 return INSTANCE;  
 }  
 @Override  
 protected Object clone() throws CloneNotSupportedException {  
 throw new CloneNotSupportedException();  
 }  
  
}

Now, we can run the CloningSingleton class; it will throw CloneNotSupportedException while creating a clone object of the Singleton object.

**Making Singletons With Enum in Java**

All of the above problems can be solved very easily by using the enum type to make singletons.

**Singleton With Enum**

public enum SingletonEnum {  
 INSTANCE;  
  
 int value;  
  
 public int getValue() {  
 return value;  
 }  
  
 public void setValue(int value) {  
 this.value = value;  
 }  
}

The three lines above make a singleton without any of the problems discussed. Since enums are inherently serializable, we don’t need to implement it with a serializable interface. The reflection problem is also not there. Therefore, it is 100% guaranteed that only one instance of the singleton is present within a JVM. Thus, this method is recommended as the **best method of making singletons in Java.**

Hope you found this post useful!