ABOUT ROHIT JOSHI

Rohit Joshi is a Senior Software Engineer from India. He is a Sun Certified Java Programmer and has worked on projects related to different domains. His expertise is in Core Java and J2EE technologies, but he also has good experience with front-end technologies like Javascript, JQuery, HTML5, and JQWidgets.

Singleton Design Pattern

Posted by: Rohit Joshi in Core Java September 30th, 2015

This article is part of our Academy Course titled Java Design Patterns.

In this course you will delve into a vast number of Design Patterns and see how those are implemented and utilized in Java. You will understand the reasons why patterns are so important and learn when and how to apply each one of them. Check it out here!

Table Of Contents

1. Singleton Pattern
2. How to create a class using the Singleton Pattern
3. When to use Singleton
4. Download the Source Code

1. Singleton Pattern

Sometimes it's important for some classes to have exactly one instance. There are many objects we only need one instance of them 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.

You may require only one object of a class, for example, when you are a creating the context of an application, or a thread manageable pool, registry settings, a driver to connect to the input or output console etc. More than one object of that type clearly will cause inconsistency to your program.

The Singleton Pattern ensures that a class has only one instance, and provides a global point of access to it. However, although the Singleton is the simplest in terms of its class diagram because there is only one single class, its implementation is a bit trickier.

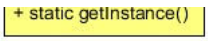

Figure 1

In this lesson, we will try different ways to create only a single object of the class and will also see how one way is better than the other.

2. How to create a class using the Singleton Pattern

There could be many ways to create such type of class, but still, we will see how one way is better than the other.

Let's start with a simple way.

What if, we provide a global variable that makes an object accessible? For example:

```
1  package com.javacodegeeks.patterns.singletonpattern;
2
3  public class SingletonEager {
4      public static SingletonEager sc = new SingletonEager();
5
6  }
```

As we know, there is only one copy of the static variables of a class, we can apply this. As far as, the client code is using this

```
sc
```

static variable its fine. But, if the client uses a new operator there would be a new instance of this class.

To stop the class to get instantiated outside the class, let's make the constructor of the class as private.

```
1  package com.javacodegeeks.patterns.singletonpattern;
2
3  public class SingletonEager {
4      public static SingletonEager sc = new SingletonEager();
5      private SingletonEager(){}
6
7  }
```

Is this going to work? I think yes. By keeping the constructor private, no other class can instantiate this class. The only way to get the object of this class is using the

```
sc
```

static variable which ensures only one object is there.

But as we know, providing a direct access to a class member is not a good idea. We will provide a method through which the sc variable will get access, not directly.

```
1  package com.javacodegeeks.patterns.singletonpattern;
2
3  public class SingletonEager {
4      private static SingletonEager sc = new SingletonEager();
5      private SingletonEager(){}
6      public static SingletonEager getInstance(){
7          return sc;
8      }
9  }
```

So, this is our singleton class which makes sure that only one object of the class gets created and even if there are several requests, only the same instantiated object will be returned.

The one problem with this approach is that, the object would get created as soon as the class gets loaded into the JVM. If the object is never requested, there would be an object useless inside the memory.

It's always a good approach that an object should get created when it is required. So, we will create an object on the first call and then will return the same object on other successive calls.

```
01 package com.javacodegeeks.patterns.singletonpattern;
02
03 public class SingletonLazy {
04
05     private static SingletonLazy sc = null;
06     private SingletonLazy(){}
07     public static SingletonLazy getInstance(){
08         if(sc==null){
09             sc = new SingletonLazy();
10         }
11         return sc;
12     }
13 }
```

In the

```
getInstance()
```

method, we check if the static variable

is null, then instantiate the object and return it. So, on the first call when sc would be null the object gets created and on the next successive calls it will return the same object.

This surely looks good, doesn't it? But this code will fail in a multi-threaded environment. Imagine two threads concurrently accessing the class, thread t1 gives the first call to the

```
getInstance()
```

method, it checks if the static variable sc is null and then gets interrupted due to some reason. Another thread t2 calls the

```
getInstance()
```

method successfully passes the if check and instantiates the object. Then, thread t1 gets awake and it also creates the object. At this time, there would be two objects of this class.

To avoid this, we will use the

```
synchronized
```

keyword to the

```
getInstance()
```

method. With this way, we force every thread to wait its turn before it can enter the method. So, no two threads will enter the method at the same time. The synchronized comes with a price, it will decrease the performance, but if the call to the

```
getInstance()
```

method isn't causing a substantial overhead for your application, forget about it. The other workaround is to move to eager instantiation approach as shown in the previous example.

```
01 package com.javacodegeeks.patterns.singletonpattern;
02
03 public class SingletonLazyMultithreaded {
04
05     private static SingletonLazyMultithreaded sc = null;
06     private SingletonLazyMultithreaded(){}
07     public static synchronized SingletonLazyMultithreaded getInstance(){
08         if(sc==null){
09             sc = new SingletonLazyMultithreaded();
10         }
11         return sc;
12     }
13 }
```

But if you want to use synchronization, there is another technique known as "double-checked locking" to reduce the use of synchronization. With the 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.

```
01 package com.javacodegeeks.patterns.singletonpattern;
02
03 public class SingletonLazyDoubleCheck {
04
05     private volatile static SingletonLazyDoubleCheck sc = null;
06     private SingletonLazyDoubleCheck(){}
07     public static SingletonLazyDoubleCheck getInstance(){
08         if(sc==null){
09             synchronized(SingletonLazyDoubleCheck.class){
10                 if(sc==null){
11                     sc = new SingletonLazyDoubleCheck();
12                 }
13             }
14         }
15         return sc;
16     }
17 }
```

Apart from this, there are some other ways to break the singleton pattern.

1. If the class is Serializable.
2. If it's Clonable.
3. It can be break by Reflection.
4. And also if, the class is loaded by multiple class loaders.

The following example shows how you can protect your class from getting instantiated more than once.

```
01 package com.javacodegeeks.patterns.singletonpattern;
02
03 import java.io.ObjectStreamException;
04 import java.io.Serializable;
05
06 public class Singleton implements Serializable{
07
08     private static final long serialVersionUID = -1093810940935189395L;
09     private static Singleton sc = new Singleton();
10     private Singleton(){
11         if(sc!=null){
```

```
        return sc;
    }

    private Object readResolve() throws ObjectStreamException{
        return sc;
    }

    private Object writeReplace() throws ObjectStreamException{
        return sc;
    }

    public Object clone() throws CloneNotSupportedException{
        throw new CloneNotSupportedException("Singleton, cannot be clonned");
    }

    private static Class getClass(String classname) throws ClassNotFoundException {
        ClassLoader classLoader = Thread.currentThread().getContextClassLoader();
        if(classLoader == null)
            classLoader = Singleton.class.getClassLoader();
        return (classLoader.loadClass(classname));
    }

}
```

1. Implement the

   `readResolve()`

   and

   `writeReplace()`

   methods in your singleton class and return the same object through them.
2. You should also implement the

   `clone()`

   method and throw an exception so that the singleton cannot be cloned.
3. An "if condition" inside the constructor can prevent the singleton from getting instantiated more than once using reflection.
4. To prevent the singleton getting instantiated from different class loaders, you can implement the

   `getClass()`

   method. The above

   `getClass()`

   method associates the classloader with the current thread; if that classloader is null, the method uses the same classloader that loaded the singleton class.

Although we can use all these techniques, there is one simple and easier way of creating a singleton class. As of JDK 1.5, you can create a singleton class using enums. The Enum constants are implicitly static and final and you cannot change their values once created.

```
package com.javacodegeeks.patterns.singletonpattern;

public class SingletoneEnum {

    public enum SingleEnum{
        SINGLETON_ENUM;
    }
}
```

You will get a compile time error when you attempt to explicitly instantiate an Enum object. As Enum gets loaded statically, it is thread-safe. The clone method in Enum is final which ensures that enum constants never get cloned. Enum is inherently serializable, the serialization mechanism ensures that duplicate instances are never created as a result of deserialization. Instantiation using reflection is also prohibited. These things ensure that no instance of an enum exists beyond the one defined by the enum constants.

3. When to use Singleton

1. There must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.
2. When the sole instance should be extensible by sub-classing, and clients should be able to use an extended instance without modifying their code.

4. Download the Source Code

This was a lesson on Singleton Pattern. You may download the source code here: **SingletonPattern-Project**

Tagged with: DOMAIN DRIVEN DESIGN