Singleton Pattern

Singleton Pattern

Intent

- Ensure that only one instance of a class is created.

- Provide a global point of access to the object.

Implementation

The implementation involves a static member in the "Singleton" class, a private constructor and a static public method that returns a reference to the static member.

Singleton Pattern

```
(function() {
 var po = document.createElement('script'); po.type = 'text/javascript'; po.async = true;
 po.src = 'https://apis.google.com/js/plusone.js';
 var s = document.getElementsByTagName('script')[0]; s.parentNode.insertBefore(po, s);
})();
```

Tweet

Motivation

Sometimes it's important to have only one instance for a class. For example, in a system there should be only one window manager (or only a file system or only a print spooler). Usually singletons are used for centralized management of internal or external resources and they provide a global point of access to themselves.

The singleton pattern is one of the simplest design patterns: it involves only one class which is responsible to instantiate itself, to make sure it creates not more than one instance; in the same time it provides a global point of access to that instance. In this case the same instance can be used from everywhere, being impossible to invoke directly the constructor each time.

Intent

- Ensure that only one instance of a class is created.

- Provide a global point of access to the object.

Implementation

The implementation involves a static member in the "Singleton" class, a private constructor and a static public method that returns a reference to the static member.

The Singleton Pattern defines a getInstance operation which exposes the unique instance which is accessed by the clients. getInstance() is is responsible for creating its class unique instance in case it is not created yet and to return that instance.

```
class Singleton
{
 private static Singleton instance;
 private Singleton()
 {
  ...
 }

 public static synchronized Singleton getInstance()
 {
  if (instance == null)
   instance = new Singleton();

  return instance;
 }
 ...
 public void doSomething()
 {
  ...
 }
}
```

You can notice in the above code that getInstance method ensures that only one instance of the class is created. The constructor should not be accessible from the outside of the class to ensure the only way of instantiating the class would be only through the getInstance method.

The getInstance method is used also to provide a global point of access to the object and it can be used in the following manner: Singleton.getInstance().doSomething();

Applicability & Examples

According to the definition the singleton pattern should be used when there must be exactly one instance of a class, and when it must be accessible to clients from a global access point. Here are some real situations where the singleton is used:

Example 1 - Logger Classes

The Singleton pattern is used in the design of logger classes. This classes are ussualy implemented as a singletons, and provides a global logging access point in all the application components without being necessary to create an object each time a logging operations is performed.

Example 2 - Configuration Classes

The Singleton pattern is used to design the classes which provides the configuration settings for an application. By implementing configuration classes as Singleton not only that we provide a global access point, but we also keep the instance we use as a cache object. When the class is instantiated(or when a value is read) the singleton will keep the values in its internal structure. If the values are read from the database or from files this avoids the reloading the values each time the configuration parameters are used.

Example 3 - Accesing resources in shared mode

It can be used in the design of an application that needs to work with the serial port. Let's say that there are many classes in the application, working in an multi-threading environment, which needs to operate actions on the serial port. In this case a singleton with synchronized methods could be used to be used to manage all the operations on the serial port.

Example 4 - Factories implemented as Singletons

Let's assume that we design an application with a factory to generate new objects(Acount, Customer, Site, Address objects) with their ids, in an multithreading environment. If the factory is instantiated twice in 2 different threads then is possible to have 2 overlapping ids for 2 different objects. If we implement the Factory as a singleton we avoid this problem. Combining Abstract Factory or Factory Method and Singleton design patterns is a common practice.

Specific problems and implementation

Thread-safe implementation for multi-threading use.

A robust singleton implementation should work in any conditions. This is why we need to ensure it works when multiple threads uses it. As seen in the previous examples singletons can be used specifically in multi-threaded application to make sure the reads/writes are synchronized.

Lazy instantiation using double locking mechanism.

The standard implementation shown in the above code is a thread safe implementation, but it's not the best thread-safe implementation beacuse synchronization is very expensive when we are talking about the performance. We can see that the synchronized method getInstance does not need to be checked for syncronization after the object is initialized. If we see that the singleton object is already created we just have to return it without using any syncronized block. This optimization consist in checking in an unsynchronized block if the object is null and if not to check again and create it in an syncronized block. This is called double locking mechanism.

In this case case the singleton instance is created when the getInstance() method is called for the first time. This is called lazy instantiation and it ensures that the singleton instance is created only when it is needed.

```
//Lazy instantiation using double locking mechanism.
class Singleton
{
 private static Singleton instance;

 private Singleton()
 {
 System.out.println("Singleton(): Initializing Instance");
 }

 public static Singleton getInstance()
 {
  if (instance == null)
  {
   synchronized(Singleton.class)
   {
    if (instance == null)
    {
     System.out.println("getInstance(): First time getInstance was invoked!");
     instance = new Singleton();
    }
   }
  }

  return instance;
 }

 public void doSomething()
 {
  System.out.println("doSomething(): Singleton does something!");
 }
}
```

A detialed discussion(double locking mechanism) can be found on http://www-128.ibm.com/developerworks/java/library/j-dcl.html?loc=j

Early instantiation using implementation with static field

In the following implementattion the singleton object is instantiated when the class is loaded and not when it is first used, due to the fact that the instance member is declared static. This is why in we don't need to synchronize any portion of the code in this case. The class is loaded once this guarantee the uniquity of the object.

Singleton - A simple example (java)

```
//Early instantiation using implementation with static field.
class Singleton
{
 private static Singleton instance = new Singleton();
```

```
 private Singleton()
 {
  System.out.println("Singleton(): Initializing Instance");
 }

 public static Singleton getInstance()
 {
  return instance;
 }

 public void doSomething()
 {
  System.out.println("doSomething(): Singleton does something!");
 }
}
```

Protected constructor

It is possible to use a protected constructor to in order to permit the subclassing of the singeton. This techique has 2 drawbacks that makes singleton inheritance impractical:

- First of all, if the constructor is protected, it means that the class can be instantiated by calling the constructor from another class in the same package. A possible solution to avoid it is to create a separate package for the singleton.

- Second of all, in order to use the derived class all the getInstance calls should be changed in the existing code from Singleton.getInstance() to NewSingleton.getInstance().

Multiple singleton instances if classes loaded by different classloaders access a singleton.

If a class(same name, same package) is loaded by 2 diferent classloaders they represents 2 different clasess in memory.

Serialization

If the Singleton class implements the java.io.Serializable interface, when a singleton is serialized and then deserialized more than once, there will be multiple instances of Singleton created. In order to avoid this the readResolve method should be implemented. See Serializable () and readResolve Method () in javadocs.

```
 public class Singleton implements Serializable {
  ...

  // This method is called immediately after an object of this class is deserialized.
  // This method returns the singleton instance.
  protected Object readResolve() {
   return getInstance();
  }
 }
```

Abstract Factory and Factory Methods implemented as singletons.

There are certain situations when the a factory should be unique. Having 2 factories might have undesired effects when

objects are created. To ensure that a factory is unique it should be implemented as a singleton. By doing so we also avoid to instantiate the class before using it.

Hot Spot:

- Multithreading - A special care should be taken when singleton has to be used in a multithreading application.

- Serialization - When Singletons are implementing Serializable interface they have to implement readResolve method in order to avoid having 2 different objects.

- Classloaders - If the Singleton class is loaded by 2 different class loaders we'll have 2 different classes, one for each class loader.

- Global Access Point represented by the class name - The singleton instance is obtained using the class name. At the first view this is an easy way to access it, but it is not very flexible. If we need to replace the Sigleton class, all the references in the code should be changed accordingly.