# Singleton Pattern Explained

## *17/03/2011*

Sonu Ahluwalia
464770
TCS
sonu.ahluwalia@tcs.com

Java has several design patterns Singleton Pattern being the most commonly used. **Java Singleton pattern** belongs to the family of design patterns, that govern the instantiation process. This design pattern proposes that at any time there can only be one instance of a singleton (object) created by the JVM.

The class's default constructor is made private, which prevents the direct instantiation of the object by others (Other Classes). A static modifier is applied to the instance method that returns the object as it then makes this method a class level method that can be accessed without creating an object.

## *Uses:*

Singletons can be used to create a Connection Pool. If programmers create a new connection object in every class that requires it, then it's a clear waste of resources. In this scenario by using a singleton connection class we can maintain a single connection object which can be used throughout the application.

A singletons can be lazy loaded. Only when it is actually needed. That's very handy if the initialisation includes expensive resource loading or database connections.

Static classes are instantiated at runtime. This could be time consuming. Singletons can be instantiated only when needed.

## *Implementing Singleton Pattern*

To implement this design pattern we need to consider the following 4 steps:

Step 1: Provide a default Private constructor

```
public class SingletonObjectDemo {

        // Note that the constructor is private
        private SingletonObjectDemo() {
                // Optional Code
        }
}
```

Step 2: Create a Method for getting the reference to the Singleton Object

```
public class SingletonObjectDemo {

        private static SingletonObject singletonObject;
        // Note that the constructor is private
        private SingletonObjectDemo() {
                // Optional Code
        }
        public static SingletonObjectDemo getSingletonObject() {
```

```
                if (singletonObject == null) {
                        singletonObject = new SingletonObjectDemo();
                }
                return singletonObject;
        }

}
```

We write a public static getter or access method to get the instance of the Singleton Object at runtime. First time the object is created inside this method as it is null. Subsequent calls to this method returns the same object created as the object is globally declared (private) and the hence the same referenced object is returned.

Step 3: Make the Access method Synchronized to prevent Thread Problems.

```
public static synchronized SingletonObjectDemo
getSingletonObject()
```

It could happen that the access method may be called twice from 2 different classes at the same time and hence more than one object being created. This could violate the design patter principle. In order to prevent the simultaneous invocation of the getter method by 2 threads or classes simultaneously we add the synchronized keyword to the method declaration

The below program shows the final Implementation of Singleton Design Pattern in java, by using all the 3 steps mentioned above:

```
class SingletonClass {

        private static SingletonClass singletonObject;
        /** A private Constructor prevents any other class from instantiating.
*/
        private SingletonClass() {
                //      Optional Code
        }
        public static synchronized SingletonClass getSingletonObject() {
                if (singletonObject == null) {
                        singletonObject = new SingletonClass();
                }
                return singletonObject;
        }
        public Object clone() throws CloneNotSupportedException {
                throw new CloneNotSupportedException();
        }
}

public class SingletonObjectDemo {

        public static void main(String args[]) {
                //              SingletonClass obj = new SingletonClass();
                 //Compilation error not allowed
                SingletonClass obj = SingletonClass.getSingletonObject();
                // Your Business Logic
```

```
                System.out.println("Singleton object obtained");
        }
}
```

## *Misconceptions:*

Override the Object clone method to prevent cloning

```
SingletonObjectDemo clonedObject = (SingletonObjectDemo) obj.clone();

public Object clone() throws CloneNotSupportedException {
throw new CloneNotSupportedException();
}
```

There is no need to override the clone() method inherited from the Object class, as you can't clone the object of a class until or unless the class implements Cloneable marker interface. You can't create a copy of the object of the Singleton class.