Singleton Design Pattern

08 November 2021 18:36

- **Singleton pattern** is a design pattern which restricts a class to instantiate its multiple objects. It is nothing but a way of defining a class. Class is defined in such a way that only one instance of the class is created
  A singleton class shouldn't have multiple instances in any case and at any cost.
  Singleton classes are used for logging, driver objects, caching and thread pool, database connections.

- **Examples** of Singleton class
  java.lang.Runtime : Java provides a class Runtime in its lang package which is singleton in nature. Every Java application has a single instance of class Runtime that allows the application to interface with the environment in which the application is running. The current runtime can be obtained from the getRuntime() method.
  An application cannot instantiate this class so multiple objects can't be created for this class. Hence Runtime is a singleton class.

  In the real-time situation, This Singleton Design Pattern can be used in the below use cases
  - Accessing the Configuration files
  - Creating the Data base connection.
  - Accessing the Loggers

- **Steps to Create the Singleton class**

  1. Whichever the class that you want to have only 1 object throughout the application, that class should have the constructor as Private. In this way no one can create the object from other classes.
  2. Whichever the class that you want to have only 1 object or which you want to have only single object, Make that class as Final so that inheritance cannot be possible and no one can create the object.
  3. Create the global instance of the class like below and make that object as static and private
     static SingletonExample s= new SingletonExample();
  4. Create a method called getInstance() or any other meaning full name and return the above instance using this method
  5. By using this getInstance() method object is returned
  6. Creating the object at class level is called, Early initialization
  7. In Early initialization, even though object is created without calling the getInstance() method. This is the drawback of early initialization hence we go for lazy initialization

8. In Lazy initialization object is create like below

```
static SingletonExample s= null;
public static SingletonExample getInstance() throws Exception {
            if(s==null)
            {
                  s= new SingletonExample();
            }
            return s;
      }
```

9. In the above code when the getInstance() is called for the first time, global object is null and hence object will be created, and from the second time onwards, global object is not null and hence no more new object gets created.

- **How to Break the Singleton Design Pattern**

- **Reflection:** Using Reflection, Even though we create the private constructor, this constructor can be called using reflection and which will create the new object.
  To avoid this in the constructor, we should throw exception when the object is not null or assign the global instance with the instance that is already present instead of creating object with new Keyword.

- **De Serialization**: During the de Serialization, readObject method of Object Input Stream will call the readResolve method and which will create the new object.
  To Avoid this override the readResolve method and return the same object instead of creating the new object

- **Cloning:** Using the cloning concept we can create the new object because default implementation of clone method will provide the new object like (return super.clone)
  To avoid this, Override the clone method, and return the object that is created instead of creating the new object or throw the exception.

- **Multi-Threading**: In Multi-Threading, since the getInstance is static method there may be a possibility that 2 or more threads can access the getInstance method at a time and end of creation of multiple objects.
  To avoid this make the getInstance method as Synchronized so that when one thread is accessing the get Instance method, Class lock will acquire and no other thread can access the getInstance method at the same time
  Instead of creating the Synchronized method writing the synchronized block will be better

- **Double Checked locking:**
  Inside the getInstance method, we are writing the null check 2 times, this is referred as Double checked locking.
  When multiple threads are working, when there is an object no need to enter the synchronized loop hence one more condition is added.

Real time uses of Singleton Design Pattern

- Hardware access
- Database connections
- Config files
- Configuration File: This is another potential candidate for Singleton pattern because this has a performance benefit as it prevents multiple users to repeatedly access and read the configuration file or properties file. It creates a single instance of the configuration file which can be accessed by multiple calls concurrently as it will provide static config data loaded into in-memory objects. The application only reads from the configuration file for the first time and thereafter from second call onwards the client applications read the data from in-memory objects.

When you use Logger, you use Singleton pattern for the Logger class. In case it is not Singleton, every client will have its own Logger object and there will be concurrent access on the Logger instance in Multithreaded environment, and multiple clients will create/write to the Log file concurrently, this leads to data corruption.