Singleton design Pattern

- Eager initialization
- Static block initialization
- Lazy Initialization
- Thread Safe Singleton
- Bill Pugh Singleton Implementation
- Using Reflection to destroy Singleton Pattern
- Enum Singleton
- Serialization and Singleton

Eager initialization

In eager initialization, the instance of Singleton Class is created at the time of class loading, this is the easiest method to create a singleton class but it has a drawback that instance is created even though client application might not be using it.

```
public class EagerInitializedSingleton {
        private static final EagerInitializedSingleton instance = new EagerInitializedSingleton();
        //private constructor to avoid client applications to use constructor
        private EagerInitializedSingleton(){}
        public static EagerInitializedSingleton getInstance(){
                return instance;
        }
}
```

If your singleton class is not using a lot of resources, this is the approach to use. But in most of the scenarios, Singleton classes are created for resources such as File System, Database connections etc and we should avoid the instantiation until unless client calls the getInstance method. Also this method doesn't provide any options for exception handling.

Static block initialization

Static block initialization implementation is similar to eager initialization, except that instance of class is created in the static block that provides option for exception handling.

```
public class StaticBlockSingleton {
   private static StaticBlockSingleton instance;
   private StaticBlockSingleton(){}
   //static block initialization for exception handling
   static{
      try{
         instance = new StaticBlockSingleton();
      }catch(Exception e){
```

```
            throw new RuntimeException("Exception occured in creating singleton instance");
        }
    }
    public static StaticBlockSingleton getInstance(){
        return instance;
    }
}
```

Both eager initialization and static block initialization creates the instance even before it's being used and that is not the best practice to use. So in further sections, we will learn how to create Singleton class that supports lazy initialization.

Lazy Initialization

Lazy initialization method to implement Singleton pattern creates the instance in the global access method. Here is the sample code for creating Singleton class with this approach.

```
public class LazyInitializedSingleton {
    private static LazyInitializedSingleton instance;
    private LazyInitializedSingleton(){}
    public static LazyInitializedSingleton getInstance(){
        if(instance == null){
            instance = new LazyInitializedSingleton();
        }
        return instance;
    }
}
```

The above implementation works fine incase of single threaded environment but when it comes to multithreaded systems, it can cause issues if multiple threads are inside the if loop at the same time. It will destroy the singleton pattern and both threads will get the different instances of singleton class. In next section, we will see different ways to create a thread-safe singleton class.

Thread Safe Singleton

The easier way to create a thread-safe singleton class is to make the global access method synchronized, so that only one thread can execute this method at a time. General implementation of this approach is like the below class.

```
public class ThreadSafeSingleton {
    private static ThreadSafeSingleton instance;
    private ThreadSafeSingleton(){}
    public static synchronized ThreadSafeSingleton getInstance(){
        if(instance == null){
            instance = new ThreadSafeSingleton();
        }
```

```
        return instance;
    }
}
```

Above implementation works fine and provides thread-safety but it reduces the performance because of cost associated with the synchronized method, although we need it only for the first few threads who might create the separate instances (Read: Java Synchronization). To avoid this extra overhead every time, double checked locking principle is used. In this approach, the synchronized block is used inside the if condition with an additional check to ensure that only one instance of singleton class is created.

Below code snippet provides the double checked locking implementation.

```
public static ThreadSafeSingleton getInstanceUsingDoubleLocking(){
    if(instance == null){
        synchronized (ThreadSafeSingleton.class) {
            if(instance == null){
                instance = new ThreadSafeSingleton();
            }
        }
    }
    return instance;
}
```

Bill Pugh Singleton Implementation

Prior to Java 5, java memory model had a lot of issues and above approaches used to fail in certain scenarios where too many threads try to get the instance of the Singleton class simultaneously. So Bill Pugh came up with a different approach to create the Singleton class using a inner static helper class. The Bill Pugh Singleton implementation goes like this;

```
public class BillPughSingleton {
    private BillPughSingleton(){}
    private static class SingletonHelper{
        private static final BillPughSingleton INSTANCE = new BillPughSingleton();
    }
    public static BillPughSingleton getInstance(){
        return SingletonHelper.INSTANCE;
    }
}
```

Notice the private inner static class that contains the instance of the singleton class. When the singleton class is loaded, SingletonHelper class is not loaded into memory and only when someone calls the getInstance method, this class gets loaded and creates the Singleton class instance.

This is the most widely used approach for Singleton class as it doesn’t require synchronization. I am using this approach in many of my projects and it’s easy to understand and implement also.

Using Reflection to destroy Singleton Pattern

Reflection can be used to destroy all the above singleton implementation approaches. Let's see this with an example class.

```
import java.lang.reflect.Constructor;
public class ReflectionSingletonTest {
   public static void main(String[] args) {
      EagerInitializedSingleton instanceOne = EagerInitializedSingleton.getInstance();
      EagerInitializedSingleton instanceTwo = null;
      try {
         Constructor[] constructors = EagerInitializedSingleton.class.getDeclaredConstructors();
         for (Constructor constructor : constructors) {
            //Below code will destroy the singleton pattern
            constructor.setAccessible(true);
            instanceTwo = (EagerInitializedSingleton) constructor.newInstance();
            break;
         }
      } catch (Exception e) {
         e.printStackTrace();
      }
      System.out.println(instanceOne.hashCode());
      System.out.println(instanceTwo.hashCode());
   }
}
```

When you run the above test class, you will notice that hashCode of both the instances are not same that destroys the singleton pattern. Reflection is very powerful and used in a lot of frameworks like Spring and Hibernate, do check out Java Reflection Tutorial.

Enum Singleton

To overcome this situation with Reflection, Joshua Bloch suggests the use of Enum to implement Singleton design pattern as Java ensures that any enum value is instantiated only once in a Java program. Since Java Enum values are globally accessible, so is the singleton. The drawback is that the enum type is somewhat inflexible; for example, it does not allow lazy initialization.

```
public enum EnumSingleton {
   INSTANCE;
   public static void doSomething(){
      //do something
   }
}
```

Serialization and Singleton

Sometimes in distributed systems, we need to implement Serializable interface in Singleton class so that we can store it's state in file system and retrieve it at later point of time. Here is a small singleton class that implements Serializable interface also.

```
import java.io.Serializable;
 public class SerializedSingleton implements Serializable{
    private static final long serialVersionUID = -7604766932017737115L;
    private SerializedSingleton(){}
    private static class SingletonHelper{
        private static final SerializedSingleton instance = new SerializedSingleton();
    }
    public static SerializedSingleton getInstance(){
        return SingletonHelper.instance;
    }
}
```

The problem with above serialized singleton class is that whenever we deserialize it, it will create a new instance of the class. Let's see it with a simple program.

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectInputStream;
import java.io.ObjectOutput;
import java.io.ObjectOutputStream;
public class SingletonSerializedTest {
    public static void main(String[] args) throws FileNotFoundException, IOException, ClassNotFoundException {
        SerializedSingleton instanceOne = SerializedSingleton.getInstance();
        ObjectOutput out = new ObjectOutputStream(new FileOutputStream(
                "filename.ser"));
        out.writeObject(instanceOne);
        out.close();
        //deserailize from file to object
        ObjectInput in = new ObjectInputStream(new FileInputStream(
                "filename.ser"));
        SerializedSingleton instanceTwo = (SerializedSingleton) in.readObject();
        in.close();
        System.out.println("instanceOne hashCode="+instanceOne.hashCode());
        System.out.println("instanceTwo hashCode="+instanceTwo.hashCode());
```

```
    }
}
```

Output of the above program is;

```
instanceOne hashCode=2011117821
instanceTwo hashCode=109647522
```

So it destroys the singleton pattern, to overcome this scenario all we need to do it provide the implementation of readResolve() method.

```
protected Object readResolve() {
    return getInstance();
}
```

After this you will notice that hashCode of both the instances are same in test program.