Singleton

Reference: Java Singleton Class - GeeksforGeeks

Singleton Pattern says that just **"define a class that has only one instance and provides a global point of access to it".**

In other words, a class must ensure that only single instance should be created, and single object can be used by all other classes.

The most popular approach is to implement a Singleton by creating a regular class and making sure it has:

- A private constructor
- A static field containing its only instance
- A static factory method for obtaining the instance

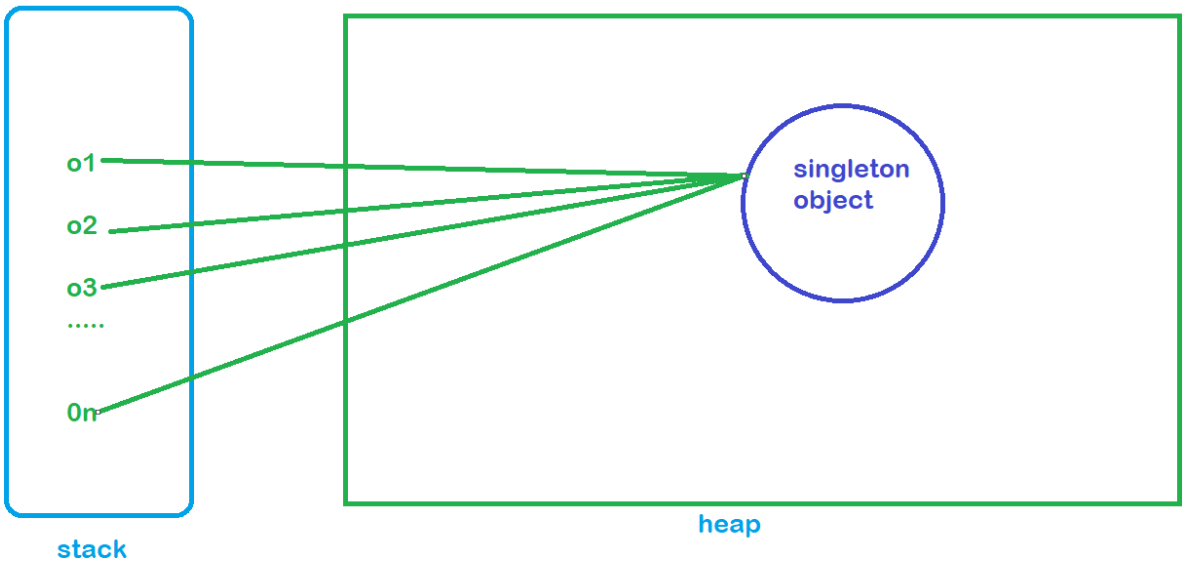
Eager initialization

Object created at the time of class loading, drawback is even client does not need it, it has been create by default. Thread safe. Reflection can break it.

```
public class EagerInitilization  {

    private static final EagerInitilization INSTANCE = new EagerInitilization();

    private EagerInitilization() {
        //logic
    }

    public static EagerInitilization getInstance() {
        return INSTANCE;
    }

}
```

```
package com.hdfc.singleton;

public class SingletonTest {
    public static void main(String[] args) {
        EagerInitilization object1 = EagerInitilization.getInstance();
        EagerInitilization object2 = EagerInitilization.getInstance();

        System.out.println(object1.hashCode());
        System.out.println(object2.hashCode());
        System.out.println(object1.equals(object2));
    }
}
```

 Contact: 8087883669



Lazy Initialization

When needed, it creates the object, but not thread safe.
Reflection can break it.

```
public class LazyInitializationSingleton   {

    private static LazyInitializationSingleton INSTANCE;

    private LazyInitializationSingleton() {
        super();
    }

    public static LazyInitializationSingleton getInstance() {
        if(null == INSTANCE) //thread -1 & thread-2
            INSTANCE = new LazyInitializationSingleton();
        return INSTANCE;
    }
}
```

```
//Runtime is singleton class
//Runtime object1 = Runtime.getRuntime();
//Runtime object2 = Runtime.getRuntime();
//Runtime object3 = Runtime.getRuntime();
```

Static Block Singleton

Provide place for exception handling, thread safe.

```
package com.hdfc.singleton;

public class StaticBlockSingleton {

    private static StaticBlockSingleton INSTANCE;


    static {
        try{

            //Logic to connect db
            //db server name
            //user name
            //pass word

            INSTANCE = new StaticBlockSingleton();
        }catch (Exception e){
            throw new RuntimeException("something went wrong");
        }finally {
            //close connection
        }
    }


    private StaticBlockSingleton() {
        //logic
    }

    public static StaticBlockSingleton getInstance() {
        return INSTANCE;
    }

}
```

Thread safe

Since synchronized on all method, not performance effective.

```
package com.hdfc.singleton;

public class ThreadSafeSingleton {

    private static ThreadSafeSingleton INSTANCE;

    private ThreadSafeSingleton() {
        //Logic
    }

    public synchronized static ThreadSafeSingleton getInstance() {
        //withouth thread logic
       //

        if(null == INSTANCE)
            INSTANCE = new ThreadSafeSingleton();
        return INSTANCE;

        //
```

```
        //withouth thread logic

        //1100 lines
    }
}
```

Thread safe double check locking singleton

Better performance the synchronized method.

```
package com.hdfc.singleton;

public class DoubleCheckSynchronizedBlock {

    private static DoubleCheckSynchronizedBlock INSTANCE;

    private DoubleCheckSynchronizedBlock() {
        //Logic
    }

    public static DoubleCheckSynchronizedBlock getInstance() {
        //withouth thread logic
        //

        if (null == INSTANCE) {
            synchronized (DoubleCheckSynchronizedBlock.class) {
                if (null == INSTANCE) {
                    INSTANCE = new DoubleCheckSynchronizedBlock();
                }
            }
        }
        return INSTANCE;

        //
        //withouth thread logic

        //1100 lines
    }
}
```

BillPughSingleton: thread safe.

```
package com.hdfc.singleton;

public class BillPughSingleton {

    private BillPughSingleton() {

    }

    public static BillPughSingleton getInstance() {
        return Helper.INSTANCE;
    }

    //Eager Initilization
    //static inner class
```

```
    private static class Helper { //static -> there should be only one copy
        private static final BillPughSingleton INSTANCE = new BillPughSingleton(); //private are accessible
from BillPughSingleton
    }


}
```

EnumSingleton

Java 5 introduced enum
Enum are by default singleton and thread safe.
Cannot be broken using reflection.

```
package com.hdfc.singleton;

public enum EnumSingleto {
    INSTANCE1;//cannot break using reflection
}
```

Break singleton using reflection.

Homework : try to break all singleton implementations using reflection.

```
package com.hdfc.singleton;

import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;

public class BreakSingletonUsingReflection {
    public static void main(String[] args) throws InvocationTargetException, InstantiationException, Ille-
galAccessException {


        EagerInitilization object1 = EagerInitilization.getInstance();
        EagerInitilization object2 = null;

        Constructor[] declaredConstructors = EagerInitilization.class.getDeclaredConstructors();
        for(Constructor constructor: declaredConstructors){
            constructor.setAccessible(true);
            object2 = (EagerInitilization)constructor.newInstance(null);
        }

        System.out.println(object1.hashCode());
        System.out.println(object2.hashCode());

        System.out.println(object1.equals(object2));
    }
}
```

Break Singleton using Serialization.

If parent implemented Serializable the child considered as Serializable.

```
package com.hdfc.singleton;

import java.io.Serializable;

class Parent implements Serializable {}

public class EagerInitilization extends Parent {

    private static final EagerInitilization INSTANCE = new EagerInitilization();

    private EagerInitilization() {
        //logic
    }

    public static EagerInitilization getInstance() {
        return INSTANCE;
    }


    //read the documentation : https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html
    //this method is used by serialztion/deserialziation process.
    //callback method
    protected Object readResolve(){
        return getInstance();
    }

}



package com.hdfc.singleton;

import java.io.*;

public class BreakUsingSerialiazation {
    public static void main(String[] args) throws IOException, ClassNotFoundException {

        EagerInitilization object = EagerInitilization.getInstance();
        serializeEagerInitilization(object);

        EagerInitilization object2 = deSerializeEagerInitilization();
        System.out.println(object.hashCode());
        System.out.println(object2.hashCode());
        System.out.println(object.equals(object2));
    }


    public static void serializeEagerInitilization(EagerInitilization object) throws IOException {
        FileOutputStream fos = new FileOutputStream("eager.serialized");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(object);
        oos.flush();
        oos.close();//Closes the stream. This method must be called to release any resources associated with
the stream.
    }

    public static EagerInitilization deSerializeEagerInitilization() throws IOException, ClassNotFoundExcep-
tion {
```

```
        FileInputStream fos = new FileInputStream("eager.serialized");
        ObjectInputStream oos = new ObjectInputStream(fos);
        EagerInitilization object = (EagerInitilization) oos.readObject();
        oos.close();//Closes the stream. This method must be called to release any resources associated with
the stream.
        return object;
    }
}
```

Break singleton using clone

```
package com.hdfc.singleton;

class Parent implements Cloneable{}
public class LazyInitializationSingleton extends  Parent {

    private static LazyInitializationSingleton INSTANCE;

    private LazyInitializationSingleton() {
        super();
    }

    public static LazyInitializationSingleton getInstance() {
        if(null == INSTANCE) //thread -1 & thread-2
            INSTANCE = new LazyInitializationSingleton();
        return INSTANCE;
    }

    @Override
    protected Object clone() throws CloneNotSupportedException {
       throw new RuntimeException("Clone not supported for singleton class");
    }
}
```