Singleton design pattern-

It means define the class which has single instance that provide the global point of access to it called as singleton design pattern.

Why?

```
package com.test

public class SingletonTest {

        private static SingletonTest s;

        private static SingletonTest test() {

                s = new SingletonTest();

                return s;

        }


public static void main(String args[]) {

System.out.println("one instance=" + SingletonTest.s.test().hashCode());

System.out.println("two instance=" + SingletonTest.s.test().hashCode());

System.out.println("three instance=" + SingletonTest.s.test().hashCode());

  }

}
```

Output-

```
one instance=1159190947

two instance=925858445

three instance=798154996
```

In the above program, we have created the three instance of same class but I want to create only one instance of class then I should go for singleton design pattern.

When?

If you have business requirement in which only one object is created then you should go for singleton design pattern.

}

Steps to create the singleton design pattern as

Step-1 Create class singleton and static member of class.

```
package com.test
public class Singleton{
        private static Singleton singletonobject;
        }
```

Step- 2

Make constructor as private

```
public class Singleton {
private static Singleton singletonobject;
private singleton(){
}
}
```

Step- 3 Create the method for checking the references and use synchronized block instead of method.

```
public static Singleton getSingletonObject() {

                synchronized (Singleton.class) {

                        if (singletonobject == null) {
                                singletonobject = new Singleton();
                        } else {
                                return singletonobject;
                        }
                        return singletonobject;
                }
```

```
    }
```

Step-4 We can still able to create the copy of object by cloning it using object clone method. Override the object clone method to prevent cloning as below.

(If singleton class will implements clonable interface ->class Singleton implements Clonable and override object clone method into singleton class)

```
    public static void main(String[] args) throws CloneNotSupportedException {


        Singleton obj1 = Singleton.getSingletonObject();

        Singleton obj2 = (Singleton) obj1.clone();


        System.out.println("object 1>>"+obj1.hashCode());

        System.out.println("object 2>>"+obj2.hashCode());

    }
```

Output-

```
object 1>>2018699554

object 2>>1311053135
```

This again violates the design principle. We need to override the object clone method which throws CloneNotSupportedException.

```
@Override

    protected Object clone() throws CloneNotSupportedException {

        // TODO Auto-generated method stub

        return super.clone();

    }
```

Output-

```
object 1>>2018699554
object 2>>1311053135

package com.test;

import java.io.Serializable;

public class Singleton implements Cloneable {

        private static Singleton singletonobject;

        private Singleton() {

        }

        public static Singleton getSingletonObject() {

                synchronized (Singleton.class) {

                        if (singletonobject == null) {
                                singletonobject = new Singleton();
                        } else {
                                return singletonobject;
                        }
                        return singletonobject;
                }

        }

        @Override
        protected Object clone() throws CloneNotSupportedException {
                // TODO Auto-generated method stub
                return super.clone();
        }

        public static void main(String[] args) throws CloneNotSupportedException {

                Singleton obj1 = Singleton.getSingletonObject();
                Singleton obj2 = (Singleton) obj1.clone();

                System.out.println("object 1>>"+obj1.hashCode());
                System.out.println("object 2>>"+obj2.hashCode());
        }

}
```

Singleton pattern with serialization-

Some times in distributed system, we need to implement serializable interface in singleton class so that we can store it state in file system and retrieve it later point.

```
package com.test;

import java.io.Serializable;

public class Singleton implements Cloneable,Serializable {

        private static Singleton singletonobject;

        private Singleton() {
	}

	public static Singleton getSingletonObject() {

		synchronized (Singleton.class) {

			if (singletonobject == null) {
				singletonobject = new Singleton();
			} else {
				return singletonobject;
			}
			return singletonobject;
		}

	}

	@Override
	protected Object clone() throws CloneNotSupportedException {
		// TODO Auto-generated method stub
		return super.clone();
	}

}
```

Step- 2

Create class singleton with serialization

```
package com.test;
```

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInput;
import java.io.ObjectInputStream;
import java.io.ObjectOutput;
import java.io.ObjectOutputStream;
import java.io.Serializable;

public class SingletonSerializedTest implements Serializable {

	/**
	 *
	 */
	private static final long serialVersionUID = 1L;

	public static void main(String[] args) {

		try {
			Singleton instance1 = Singleton.getSingletonObject();
			ObjectOutput out = new ObjectOutputStream(new
FileOutputStream("E:\\test.txt"));
			out.writeObject(instance1);
			out.close();

			ObjectInput in = new ObjectInputStream(new
FileInputStream("E:\\test.txt"));
			Singleton instance2 = (Singleton) in.readObject();
			in.close();

			System.out.println("instance 1>>" + instance1.hashCode());
```

```
                        System.out.println("instance 2>>" + instance2.hashCode());

                } catch (Exception e) {
                        e.printStackTrace();
                }
        }
}
```

Step 3- create the input and output for reading and writing the object .

```
ObjectOutput out = new ObjectOutputStream(new
FileOutputStream("D:\\test.txt"));
                        out.writeObject(instance1);
                        out.close();

                        // deserailize from file to object
                        ObjectInput in = new ObjectInputStream(new
FileInputStream("D:\\test.txt"));

                        Singleton instance2 = (Singleton) in.readObject();
                        in.close();
```

Step- 4 run program to check output.

```
instance1 hashCode:- 625576447
instance2 hashCode:- 1494279232
```

It is prove that singleton object is not preserved during deserialization. Because both instance have different hashcode.

Step-5

To overcome this problem, we need to override the readResolve() method into singleton class.

```
protected Object readResolve() {

                return singletonobject;

        }
```

Step -5 after running SingletonSerializedTest program, you will get the output as

```
instance1 hashCode:- 625576447

instance2 hashCode:- 625576447
```