**Design Patterns**

**What is Design Pattern?**

* A design pattern is a reusable and generalized solution to a recurring problem that occurs in software design.
* Design patterns provide proven solutions to common design challenges, allowing developers to solve problems efficiently while following best practices and maintaining code quality.
* These patterns are not specific pieces of code but rather high-level concepts that guide the overall structure and organization of software systems.
* Design patterns help address common issues in software development, such as flexibility, modularity, maintainability, and scalability.
* They provide a shared vocabulary for developers to communicate about design decisions and help establish a common approach to solving problems.

**History of Design Patterns**

As A Pattern Language explains “Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in a way that you can use this solution a million times over, without ever doing it the same way twice.”

* **Christopher Alexander** was the first person who invented the Design Patterns in 1977.
* But later the **Gang of Four - Design patterns, elements of reusable object-oriented software** book was written by a group of four persons named as Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides in 1995.
* That's why all the 23 Design Patterns are known as **Gang of Four (GoF) Design Patterns.**

**Advantage of design pattern:**

1. They are reusable in multiple projects.
2. They provide the solutions that help to define the system architecture.
3. They capture the software engineering experiences.
4. They provide transparency to the design of an application.
5. They are well-proved and testified solutions since they have been built upon the knowledge and experience of expert software developers.
6. Design patterns don’t guarantee an absolute solution to a problem. They provide clarity to the system architecture and the possibility of building a better system.

### **When should we use the design patterns?**

We must use the design patterns **during the analysis and requirement phase of SDLC**(Software Development Life Cycle).

Design patterns ease the analysis and requirement phase of SDLC by providing information based on prior hands-on experiences.

**Classification of design patterns**

In core java, there are mainly three types of design patterns, which are further divided into their sub-parts:

**1. Creational Design Pattern**

This type deals with the object creation and initialization. This pattern gives the program more flexibility in deciding which objects need to be created for a given case

1. Factory Pattern
2. Abstract Factory Pattern
3. Singleton Pattern
4. Prototype Pattern
5. Builder Pattern.

**2. Structural Design Pattern**

This type deals with class and object composition. This pattern focusses on decoupling interface and implementation of classes and its objects.

1. Adapter Pattern
2. Bridge Pattern
3. Composite Pattern
4. Decorator Pattern
5. Facade Pattern
6. Flyweight Pattern
7. Proxy Pattern

**3. Behavioural Design Pattern**

This type deals with the communication between objects.

1. Chain Of Responsibility Pattern
2. Command Pattern
3. Interpreter Pattern
4. Iterator Pattern
5. Mediator Pattern
6. Memento Pattern
7. Observer Pattern
8. State Pattern
9. Strategy Pattern
10. Template Pattern
11. Visitor Pattern

**Singleton design pattern:**

The Singleton design pattern is a creational design pattern that ensures a class has only one instance and provides a global point of access to that instance. In other words, it restricts the instantiation of a class to a single object and provides a mechanism to access that object throughout the application's lifecycle.

**Key features of the Singleton pattern:**

**Single Instance:** The Singleton class guarantees that only one instance of itself will be created.

**Global Access:** The Singleton instance is typically accessible globally within the application, allowing any part of the code to use it.

**Lazy Initialization:** The Singleton instance is created only when it's first requested, not at the time the class is loaded.

**Thread Safety:** A well-implemented Singleton ensures that the instance is created in a thread-safe manner, preventing multiple threads from creating duplicate instances.

**Advantage of singleton**

* Saves memory because object is not created at each request. Only single instance is reused again and again.

**Disadvantage of singleton**

* Overusing Singleton can lead to tight coupling and difficulties in testing. Depending on the use case, other patterns or dependency injection might be more suitable.

#### **Usage of Singleton design pattern**

* Singleton pattern is mostly used in multi-threaded and database applications. It is used in logging, caching, thread pools, configuration settings etc.

**Implementation:**

There are two forms of singleton design pattern

* **Eager Instantiation:** creation of instance at load time.
* **Lazy Instantiation:** creation of instance when required.

## [**Singleton Pattern Principles**](https://www.digitalocean.com/community/tutorials/java-singleton-design-pattern-best-practices-examples#singleton-pattern-principles)

* Singleton pattern restricts the instantiation of a class and ensures that only one instance of the class exists in the Java Virtual Machine.
* The singleton class must provide a global access point to get the instance of the class.
* Singleton pattern is used for [logging](https://www.digitalocean.com/community/tutorials/logger-in-java-logging-example), drivers objects, caching, and [thread pool](https://www.digitalocean.com/community/tutorials/threadpoolexecutor-java-thread-pool-example-executorservice).
* Singleton design pattern is also used in other design patterns like [Abstract Factory](https://www.digitalocean.com/community/tutorials/abstract-factory-design-pattern-in-java), [Builder](https://www.digitalocean.com/community/tutorials/builder-design-pattern-in-java), [Prototype](https://www.digitalocean.com/community/tutorials/prototype-design-pattern-in-java), [Facade](https://www.digitalocean.com/community/tutorials/facade-design-pattern-in-java), etc.
* Singleton design pattern is used in core Java classes also (for example, java.lang.Runtime, java.awt.Desktop).

## [**Java Singleton Pattern Implementation**](https://www.digitalocean.com/community/tutorials/java-singleton-design-pattern-best-practices-examples#java-singleton-pattern-implementation)

To implement a singleton pattern, we have different approaches, but all of them have the following common concepts.

* Private constructor to restrict instantiation of the class from other classes.
* Private static variable of the same class that is the only instance of the class.
* Public static method that returns the instance of the class, this is the global access point for the outer world to get the instance of the singleton class.

**Code for eager initialization**

In eager initialization, the instance of the singleton class is created at the time of class loading. The drawback to eager initialization is that the method is created even though the client application might not be using it. Here is the implementation of the static initialization singleton class:

**1. Public constructor**

**package** com.stg;

**public** **class** EagerInitializedSingleton {

**private** **static** **final** EagerInitializedSingleton ***instance*** = **new** EagerInitializedSingleton();

**public** EagerInitializedSingleton() {

}

**public** **static** EagerInitializedSingleton getInstance() {

**return** ***instance***;

}

}

**TestSingleton.java**

**package** com.stg;

**public** **class** TestSingleton {

**public** **static** **void** main(String[] args) {

EagerInitializedSingleton instance01 = **new** EagerInitializedSingleton();

EagerInitializedSingleton instance02 = **new** EagerInitializedSingleton();

EagerInitializedSingleton instance03 = **new** EagerInitializedSingleton();

System.***out***.println(instance01);

System.***out***.println(instance02);

System.***out***.println(instance03);

}

}

Output: -

com.stg.EagerInitializedSingleton@6504e3b2

com.stg.EagerInitializedSingleton@379619aa

[com.stg.EagerInitializedSingleton@cac736f](mailto:com.stg.EagerInitializedSingleton@cac736f)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**2. Private constructor**

**package** com.stg;

**public** **class** EagerInitializedSingleton {

**private** **static** **final** EagerInitializedSingleton ***instance*** = **new** EagerInitializedSingleton();

**private** EagerInitializedSingleton() {

}

**public** **static** EagerInitializedSingleton getInstance() {

**return** ***instance***;

}

}

**TestSingleton.java**

**package** com.stg;

**public** **class** TestSingleton {

**public** **static** **void** main(String[] args) {

EagerInitializedSingleton instance01 = EagerInitializedSingleton.*getInstance*();

EagerInitializedSingleton instance02 = EagerInitializedSingleton.*getInstance*();

EagerInitializedSingleton instance03 = EagerInitializedSingleton.*getInstance*();

System.***out***.println(instance01);

System.***out***.println(instance02);

System.***out***.println(instance03);

}

}

**Output:**

com.stg.EagerInitializedSingleton@6504e3b2

com.stg.EagerInitializedSingleton@6504e3b2

com.stg.EagerInitializedSingleton@6504e3b2

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. We should avoid the instantiation unless the client calls the getInstance method. Also, this method doesn’t provide any options for exception handling.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

## [**3. Static block initialization**](https://www.digitalocean.com/community/tutorials/java-singleton-design-pattern-best-practices-examples#2-static-block-initialization)

[Static block](https://www.digitalocean.com/community/tutorials/static-keyword-in-java) initialization implementation is similar to eager initialization, except that instance of the class is created in the static block that provides the option for [exception handling](https://www.digitalocean.com/community/tutorials/exception-handling-in-java).

StaticBlockSingleton class

**package** com.stg;

**public** **class** StaticBlockSingleton {

**private** **static** StaticBlockSingleton *instance*;

**private** StaticBlockSingleton() {

}

**static** {

**try** {

*instance* = **new** StaticBlockSingleton();

} **catch** (Exception e) {

**throw** **new** RuntimeException("Exception occurred in creating singleton instance");

}

}

**public** **static** StaticBlockSingleton getInstance() {

**return** *instance*;

}

}

TestSingleton.java

**package** com.stg;

**public** **class** TestSingleton {

**public** **static** **void** main(String[] args) {

StaticBlockSingleton instance01 = StaticBlockSingleton.*getInstance*();

StaticBlockSingleton instance02 = StaticBlockSingleton.*getInstance*();

StaticBlockSingleton instance03 = StaticBlockSingleton.*getInstance*();

System.***out***.println(instance01);

System.***out***.println(instance02);

System.***out***.println(instance03);

}

}

**Output:-**

com.stg.StaticBlockSingleton@6504e3b2

com.stg.StaticBlockSingleton@6504e3b2

com.stg.StaticBlockSingleton@6504e3b2

**4. LazyInitializedSingleton Class**

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

**package** com.stg;

**public** **class** LazyInitializedSingleton {

**private** **static** LazyInitializedSingleton *instance*;

**private** LazyInitializedSingleton() {

}

**public** **static** LazyInitializedSingleton getInstance() {

**if** (*instance* == **null**) {

*instance* = **new** LazyInitializedSingleton();

}

**return** *instance*;

}

}

TestSingleton.java

**package** com.stg;

**public** **class** TestSingleton {

**public** **static** **void** main(String[] args) {

LazyInitializedSingleton instance01 = LazyInitializedSingleton.*getInstance*();

LazyInitializedSingleton instance02 = LazyInitializedSingleton.*getInstance*();

LazyInitializedSingleton instance03 = LazyInitializedSingleton.*getInstance*();

System.***out***.println(instance01);

System.***out***.println(instance02);

System.***out***.println(instance03);

}

}

Output:

com.stg.LazyInitializedSingleton@6504e3b2

com.stg.LazyInitializedSingleton@6504e3b2

[com.stg.LazyInitializedSingleton@6504e3b2](mailto:com.stg.LazyInitializedSingleton@6504e3b2)

The preceding implementation works fine in the case of the single-threaded environment, but when it comes to multi-threaded systems, it can cause issues if multiple threads are inside the if condition at the same time. It will destroy the singleton pattern and both threads will get different instances of the singleton class. In the next section, we will see different ways to create a [thread-safe](https://www.digitalocean.com/community/tutorials/thread-safety-in-java) singleton class.

**5.ThreadSafeSingleton class**

1. Synchronized at method level

**package** com.stg;

**public** **class** ThreadSafeSingleton {

**private** **static** ThreadSafeSingleton *instance*;

**private** ThreadSafeSingleton() {

}

**public** **static** **synchronized** ThreadSafeSingleton getInstance() {

**if** (*instance* == **null**) {

*instance* = **new** ThreadSafeSingleton();

}

**return** *instance*;

}

}

**TestSingleton.java**

**package** com.stg;

**public** **class** TestSingleton {

**public** **static** **void** main(String[] args) {

ThreadSafeSingleton instance01 = ThreadSafeSingleton.*getInstance*();

ThreadSafeSingleton instance02 = ThreadSafeSingleton.*getInstance*();

System.***out***.println(instance01);

System.***out***.println(instance02);

}

}

**Output:**

com.stg.ThreadSafeSingleton@6504e3b2

[com.stg.ThreadSafeSingleton@6504e3b2](mailto:com.stg.ThreadSafeSingleton@6504e3b2)

2. Synchronized Block

**package** com.stg;

**public** **class** ThreadSafeSingleton {

**private** **static** ThreadSafeSingleton *instance*;

**private** ThreadSafeSingleton() {

}

**public** **static** ThreadSafeSingleton getInstanceUsingDoubleLocking() {

**if** (*instance* == **null**) {

**synchronized** (ThreadSafeSingleton.**class**) {

**if** (*instance* == **null**) {

*instance* = **new** ThreadSafeSingleton();

}

}

}

**return** *instance*;

}

}

TestSingleton.java

**package** com.stg;

**public** **class** TestSingleton {

**public** **static** **void** main(String[] args) {

ThreadSafeSingleton instance01 = ThreadSafeSingleton.*getInstanceUsingDoubleLocking*();

ThreadSafeSingleton instance02 = ThreadSafeSingleton.*getInstanceUsingDoubleLocking*();

System.***out***.println(instance01);

System.***out***.println(instance02);

}

}

Output:

com.stg.ThreadSafeSingleton@6504e3b2

com.stg.ThreadSafeSingleton@6504e3b2

## [**Using Reflection to destroy Singleton Pattern**](https://www.digitalocean.com/community/tutorials/java-singleton-design-pattern-best-practices-examples#6-using-reflection-to-destroy-singleton-pattern)

**package** com.stg;

**public** **class** EagerInitializedSingleton {

**private** **static** **final** EagerInitializedSingleton ***instance*** = **new** EagerInitializedSingleton();

**private** EagerInitializedSingleton() {

}

**public** **static** EagerInitializedSingleton getInstance() {

**return** ***instance***;

}

}

## **ReflectionSingletonTest class**

**package** com.stg;

**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) {

// This code will destroy the singleton pattern

constructor.setAccessible(**true**);

instanceTwo = (EagerInitializedSingleton) constructor.newInstance();

**break**;

}

} **catch** (Exception e) {

e.printStackTrace();

}

System.***out***.println(instanceOne);

System.***out***.println(instanceTwo);

}

}

**Output:-**

com.stg.EagerInitializedSingleton@6504e3b2

com.stg.EagerInitializedSingleton@379619aa

**If we comment constructor.setAccessible(true); this one then it should throw an exception**

java.lang.IllegalAccessException: class com.stg.ReflectionSingletonTest cannot access a member of class com.stg.EagerInitializedSingleton with modifiers "private"

at java.base/jdk.internal.reflect.Reflection.newIllegalAccessException(Reflection.java:392)

at java.base/java.lang.reflect.AccessibleObject.checkAccess(AccessibleObject.java:674)

at java.base/java.lang.reflect.Constructor.newInstanceWithCaller(Constructor.java:489)

at java.base/java.lang.reflect.Constructor.newInstance(Constructor.java:480)

at com.stg.ReflectionSingletonTest.main(ReflectionSingletonTest.java:14)

com.stg.EagerInitializedSingleton@53bd815b

null

**Enum**

To overcome issues raised by reflection, [enums](https://www.geeksforgeeks.org/enum-in-java/) are used because java ensures internally that the enum value is instantiated only once. Since java Enums are globally accessible, they can be used for singletons. Its only drawback is that it is not flexible i.e it does not allow lazy initialization.

As the constructor for an enum is **package-private** or **private access**, It automatically creates the constants that are defined at the beginning of the enum body. You cannot invoke an enum constructor yourself, so it is not possible for Reflection to utilize it. Hence, reflection can’t break singleton property in the case of enums.

**SingletonEnum.java**

**package** com.stg;

**public** **enum** SingletonEnum {

***INSTANCE***;

**private** **int** value;

**public** **int** getValue() {

**return** value;

}

**public** **void** setValue(**int** value) {

**this**.value = value;

}

}

**EnumDemo**

**package** com.stg;

**public** **class** EnumDemo {

**public** **static** **void** main(String[] args) {

SingletonEnum instance01 = SingletonEnum.***INSTANCE***;

SingletonEnum instance02 = SingletonEnum.***INSTANCE***;

System.***out***.println(instance01);

System.***out***.println(instance02);

}

}

**Output:**

INSTANCE

INSTANCE

* **Now go to reflectionsingletontest class again**

**SingletonEnum class**

**package** com.stg;

**public** **enum** SingletonEnum {

***INSTANCE***;

**private** **int** value;

**public** **int** getValue() {

**return** value;

}

**public** **void** setValue(**int** value) {

**this**.value = value;

}

}

**ReflectionSingletonTest class**

**package** com.stg;

**import** java.lang.reflect.Constructor;

**public** **class** ReflectionSingletonTest {

**public** **static** **void** main(String[] args) {

SingletonEnum instanceOne = SingletonEnum.***INSTANCE***;

SingletonEnum instanceTwo = **null**;

**try** {

Constructor[] constructors = SingletonEnum.**class**.getDeclaredConstructors();

**for** (Constructor constructor : constructors) {

// This code will destroy the singleton pattern

constructor.setAccessible(**true**);

instanceTwo = (SingletonEnum) constructor.newInstance();

**break**;

}

} **catch** (Exception e) {

e.printStackTrace();

}

System.***out***.println(instanceOne);

System.***out***.println(instanceTwo);

}

}

**Output:**

java.lang.IllegalArgumentException: Cannot reflectively create enum objects

at java.base/java.lang.reflect.Constructor.newInstanceWithCaller(Constructor.java:492)

at java.base/java.lang.reflect.Constructor.newInstance(Constructor.java:480)

at com.stg.ReflectionSingletonTest.main(ReflectionSingletonTest.java:14)

INSTANCE

null

**Serialization: -**  [Serialization](https://www.geeksforgeeks.org/serialization-in-java/) can also cause breakage of singleton property of singleton classes. Serialization is used to convert an object of a byte stream and save in a file or send it over a network. Suppose you serialize an object of a singleton class. Then if you de-serialize that object it will create a new instance and hence break the singleton pattern.

**SerializedSingleton class**

**package** com.stg;

**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***;

}

}

**SingletonSerializedTest**

**package** com.stg;

**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();

// deserialize 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);

System.***out***.println("instanceTwo hashCode = "+instanceTwo);

}

}

**Overcome serialization issue:-** To overcome this issue, we have to implement the readResolve() method.

**package** com.stg;

**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***;

}

**protected** Object readResolve() {

**return** *getInstance*();

}

}

**Output:**

instanceOne hashCode = com.stg.SerializedSingleton@c39f790

instanceTwo hashCode = com.stg.SerializedSingleton@c39f790

**Cloning:** [Cloning](https://www.geeksforgeeks.org/clone-method-in-java-2/) is the concept to create duplicate objects. Using clone we can create copy of object. Suppose, we create clone of a singleton object, then it will create a copy that is there are two instances of a singleton class, hence the class is no more singleton.

Shallow copy and deep copy are two different approaches to cloning or copying objects in programming, and they have distinct characteristics:

**Shallow Copy:**

A shallow copy of an object creates a new object, but it does not duplicate the entire object hierarchy.

In a shallow copy, the new object is a copy of the original object itself, and it contains references to the same objects as the original.

Changes made to the objects within the copied object will be reflected in the original object, and vice versa.

Shallow copying is often faster and requires less memory compared to deep copying.

In Java, you can perform a shallow copy of an object by implementing the Cloneable interface and overriding the clone() method to create a new instance of the object.

**Deep Copy:**

A deep copy of an object creates a new object and duplicates the entire object hierarchy, including all objects referenced by the original.

In a deep copy, all objects referenced by the original object are recursively copied to create new instances, ensuring that changes made to the copied object do not affect the original, and vice versa.

Deep copying is often used when you need to create entirely independent copies of complex objects or object graphs.

Deep copying can be more complex and memory-intensive compared to shallow copying, especially for objects with complex hierarchies or circular references.

**If we override super class with clone() then singleton will break**

**EagerInitializedSingleton**: Below code is shallow copy

**package** com.stg;

**public** **class** EagerInitializedSingleton **implements** Cloneable {

**private** **static** **final** EagerInitializedSingleton ***instance*** = **new** EagerInitializedSingleton();

**private** EagerInitializedSingleton() {

}

**public** **static** EagerInitializedSingleton getInstance() {

**return** ***instance***;

}

@Override

**public** Object clone() **throws** CloneNotSupportedException {

**return** **super**.clone();

}

}

**TestSingleton Class**

**package** com.stg;

**public** **class** TestSingleton {

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

EagerInitializedSingleton instance01 = EagerInitializedSingleton.*getInstance*();

EagerInitializedSingleton instance02 = EagerInitializedSingleton.*getInstance*();

EagerInitializedSingleton instance03 = EagerInitializedSingleton.*getInstance*();

EagerInitializedSingleton instance04 = (EagerInitializedSingleton) instance01.clone();

System.***out***.println(instance01);

System.***out***.println(instance02);

System.***out***.println(instance03);

System.***out***.println(instance04);

}

}

**Output:**

com.stg.EagerInitializedSingleton@6504e3b2

com.stg.EagerInitializedSingleton@6504e3b2

com.stg.EagerInitializedSingleton@6504e3b2

[com.stg.EagerInitializedSingleton@379619aa](mailto:com.stg.EagerInitializedSingleton@379619aa)

Two different hashCode means there are 2 different objects of the singleton class.

**Overcome Cloning issue:**To overcome this issue, override clone() method and throw an exception from clone method that is CloneNotSupportedException. Now, whenever user will try to create clone of singleton object, it will throw an exception and hence our class remains singleton.

**package** com.stg;

**public** **class** EagerInitializedSingleton **implements** Cloneable {

**private** **static** **final** EagerInitializedSingleton ***instance*** = **new** EagerInitializedSingleton();

**private** EagerInitializedSingleton() {

}

**public** **static** EagerInitializedSingleton getInstance() {

**return** ***instance***;

}

@Override

**public** Object clone() **throws** CloneNotSupportedException {

**throw** **new** CloneNotSupportedException();

}

}

**TestSingleton Class**

**package** com.stg;

**public** **class** TestSingleton {

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

EagerInitializedSingleton instance01 = EagerInitializedSingleton.*getInstance*();

EagerInitializedSingleton instance02 = EagerInitializedSingleton.*getInstance*();

EagerInitializedSingleton instance03 = EagerInitializedSingleton.*getInstance*();

EagerInitializedSingleton instance04 = (EagerInitializedSingleton) instance01.clone();

System.***out***.println(instance01);

System.***out***.println(instance02);

System.***out***.println(instance03);

System.***out***.println(instance04);

}

}

**Output:**

Exception in thread "main" java.lang.CloneNotSupportedException

at com.stg.EagerInitializedSingleton.clone(EagerInitializedSingleton.java:15)

at com.stg.TestSingleton.main(TestSingleton.java:9)

Now we have stopped users to create clones of the singleton class. If you don’t want to throw an exception you can also return the same instance from the clone method.

**package** com.stg;

**public** **class** EagerInitializedSingleton **implements** Cloneable {

**private** **static** **final** EagerInitializedSingleton ***instance*** = **new** EagerInitializedSingleton();

**private** EagerInitializedSingleton() {

}

**public** **static** EagerInitializedSingleton getInstance() {

**return** ***instance***;

}

@Override

**public** Object clone() **throws** CloneNotSupportedException {

**return** ***instance***;

}

}

**TestSingleton Class**

**package** com.stg;

**public** **class** TestSingleton {

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

EagerInitializedSingleton instance01 = EagerInitializedSingleton.*getInstance*();

EagerInitializedSingleton instance02 = EagerInitializedSingleton.*getInstance*();

EagerInitializedSingleton instance03 = EagerInitializedSingleton.*getInstance*();

EagerInitializedSingleton instance04 = (EagerInitializedSingleton) instance01.clone();

System.***out***.println(instance01);

System.***out***.println(instance02);

System.***out***.println(instance03);

System.***out***.println(instance04);

}

}

**Output:**

com.stg.EagerInitializedSingleton@6504e3b2

com.stg.EagerInitializedSingleton@6504e3b2

com.stg.EagerInitializedSingleton@6504e3b2

com.stg.EagerInitializedSingleton@6504e3b2

Now, as hashcode of both instances is the same that means they represent a single instance.