**How to Implement a Singleton?**

**There are multiple ways to implement Thread-Safe Singleton:**

**✅ 1. Eager Initialization (Thread-Safe but Not Lazy)**

* **The instance is created at the time of class loading.**
* **Not suitable if the instance is never used.**

**public class Singleton {**

**private static final Singleton INSTANCE = new Singleton(); // Instance created eagerly**

**private Singleton() {} // Private constructor**

**public static Singleton getInstance() {**

**return INSTANCE; // Return the single instance**

**}**

**}**

**✅ Thread-safe but not lazy (created even if not used).**

**✅ 2. Lazy Initialization (Not Thread-Safe)**

* **The instance is created only when required.**
* **Not thread-safe in a multi-threaded environment.**

**public class Singleton {**

**private static Singleton instance;**

**private Singleton() {}**

**public static Singleton getInstance() {**

**if (instance == null) { // Not thread-safe**

**instance = new Singleton();**

**}**

**return instance;**

**}**

**}**

**❌ Multiple threads might create multiple instances!**

**✅ 3. Synchronized Method (Thread-Safe but Slow)**

* **Uses synchronized to make it thread-safe.**
* **Slower because every call to getInstance() is synchronized.**

**public class Singleton {**

**private static Singleton instance;**

**private Singleton() {}**

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

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

**instance = new Singleton();**

**}**

**return instance;**

**}**

**}**

**❌ Performance issue due to synchronized method calls.**

**✅ 4. Double-Checked Locking (Best Performance)**

* **Thread-safe and avoids unnecessary synchronization.**
* **Uses volatile to ensure visibility of changes.**

**public class Singleton {**

**private static volatile Singleton instance; // Volatile ensures visibility**

**private Singleton() {}**

**public static Singleton getInstance() {**

**if (instance == null) { // First check**

**synchronized (Singleton.class) {**

**if (instance == null) { // Second check**

**instance = new Singleton();**

**}**

**}**

**}**

**return instance;**

**}**

**}**

**✅ Thread-safe & efficient! 🚀**

**-----------------**

**Ways to Break Singleton & How to Fix**

**1️⃣ Reflection (Bypassing Private Constructor)**

Java Reflection can break Singleton by making the private constructor accessible.

**📌 Code to Break Singleton using Reflection:**

import java.lang.reflect.Constructor;

public class ReflectionBreakSingleton {

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

Singleton instance1 = Singleton.getInstance();

Constructor<Singleton> constructor = Singleton.class.getDeclaredConstructor();

constructor.setAccessible(true); // Breaks the private constructor

Singleton instance2 = constructor.newInstance();

System.out.println("Instance 1: " + instance1.hashCode());

System.out.println("Instance 2: " + instance2.hashCode()); // Different hashcode = Singleton broken!

}

}

class Singleton {

private static final Singleton INSTANCE = new Singleton();

private Singleton() {}

public static Singleton getInstance() {

return INSTANCE;

}

}

**🚨 Output (Singleton Broken!)**

yaml

CopyEdit

Instance 1: 12345678

Instance 2: 87654321

✅ **Fix:** Prevent Reflection using **Enum** or throw an exception in the constructor.

1. class Singleton {

private static final Singleton INSTANCE = new Singleton();

private Singleton() {

if (INSTANCE != null) { // Prevent Reflection

throw new RuntimeException("Use getInstance() method!");

}

}

public static Singleton getInstance() {

return INSTANCE;

}

}

2.

Public enum Singelton{

INSTANCE; //Decalare variable

}

Singelton s = Singelton.INSTANCE;

System.out.println("Instance 1: " +s.hashCode());

**2️⃣ Serialization & Deserialization**

When a Singleton class implements Serializable, deserialization creates a new instance.

**📌 Code to Break Singleton using Serialization:**

import java.io.\*;

public class SerializationBreakSingleton {

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

Singleton instance1 = Singleton.getInstance();

// Serialize Singleton

ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("singleton.ser"));

oos.writeObject(instance1);

oos.close();

// Deserialize Singleton

ObjectInputStream ois = new ObjectInputStream(new FileInputStream("singleton.ser"));

Singleton instance2 = (Singleton) ois.readObject();

ois.close();

System.out.println("Instance 1: " + instance1.hashCode());

System.out.println("Instance 2: " + instance2.hashCode()); // Different hashcode = Singleton broken!

}

}

class Singleton implements Serializable {

private static final Singleton INSTANCE = new Singleton();

private Singleton() {}

public static Singleton getInstance() {

return INSTANCE;

}

}

**🚨 Output (Singleton Broken!)**

Instance 1: 12345678

Instance 2: 87654321

✅ **Fix:** Implement readResolve() to return the same instance.

class Singleton implements Serializable {

private static final Singleton INSTANCE = new Singleton();

private Singleton() {}

public static Singleton getInstance() {

return INSTANCE;

}

protected Object readResolve() { // Fix for Serialization

return INSTANCE;

}

}

**3️⃣ Cloning**

If the Singleton class implements Cloneable, cloning creates a new instance.

**📌 Code to Break Singleton using Cloning:**

class Singleton implements Cloneable {

private static final Singleton INSTANCE = new Singleton();

private Singleton() {}

public static Singleton getInstance() {

return INSTANCE;

}

@Override

protected Object clone() throws CloneNotSupportedException {

return super.clone(); // Creates a new instance!

}

}

public class CloneBreakSingleton {

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

Singleton instance1 = Singleton.getInstance();

Singleton instance2 = (Singleton) instance1.clone(); // Cloning breaks Singleton

System.out.println("Instance 1: " + instance1.hashCode());

System.out.println("Instance 2: " + instance2.hashCode()); // Different hashcode = Singleton broken!

}

}

**🚨 Output (Singleton Broken!)**

Instance 1: 12345678

Instance 2: 87654321

✅ **Fix:** Override clone() to prevent cloning.

@Override

protected Object clone() throws CloneNotSupportedException {

throw new CloneNotSupportedException("Cannot clone Singleton");

}

**4️⃣ Multithreading (Race Condition)**

If two threads access getInstance() at the same time, they may create two different instances.

**📌 Code to Break Singleton using Multithreading:**

public class MultithreadingBreakSingleton {

public static void main(String[] args) {

Runnable task = () -> {

Singleton instance = Singleton.getInstance();

System.out.println("Instance HashCode: " + instance.hashCode());

};

Thread t1 = new Thread(task);

Thread t2 = new Thread(task);

t1.start();

t2.start();

}

}

class Singleton {

private static Singleton instance;

private Singleton() {}

public static Singleton getInstance() {

if (instance == null) { // Not thread-safe!

instance = new Singleton();

}

return instance;

}

}

**🚨 Output (Singleton Broken!)**

Instance HashCode: 12345678

Instance HashCode: 87654321

✅ **Fix:** Use **Double-Checked Locking** or **Bill Pugh Singleton**.

class Singleton {

private static volatile Singleton instance;

private Singleton() {}

public static Singleton getInstance() {

if (instance == null) {

synchronized (Singleton.class) {

if (instance == null) {

instance = new Singleton();

}

}

}

return instance;

}

}

**5️⃣ Using Multiple Class Loaders**

Different class loaders may load separate instances of the same Singleton class.

**✅ Fix:** Use **Enum Singleton** (Best Approach).

public enum Singleton {

INSTANCE;

}

✅ **Enum is immune to reflection, serialization, and multithreading issues.**

**🔹 Summary: How to Protect Singleton?**

| **Breakage Type** | **Fix** |
| --- | --- |
| **Reflection** | Throw exception in the constructor or use Enum |
| **Serialization** | Implement readResolve() |
| **Cloning** | Override clone() and throw CloneNotSupportedException |
| **Multithreading** | Use **Double-Checked Locking** or **Bill Pugh Singleton** |
| **Class Loader Issues** | Use **Enum Singleton** |