9 Creational: Singleton Pattern for a Centralised Logger

**Objective** demonstrate the three common Singleton implementations (Level 1 = eager, Level 2 = lazy non-thread-safe, Level 3 = lazy thread-safe), migrate client code so it always re-uses the same instance, and reflect on the trade-offs.

#### **Starter code (src/main/java/legacy/)**

package legacy;

public class Logger {

public void log(String msg){ System.out.println(msg); }

}

package legacy;

public class Demo {

public static void main(String[] args){

new Logger().log("first"); // each call instantiates a new logger

new Logger().log("second");

}

}

#### **Tasks**

1 – analysis/singleton\_problems.md explain why uncontrolled instantiation wastes resources and hinders log aggregation.  
 2 – Create three singleton variants in src/main/java/clean/  
 • **Level 1 EagerLogger** – eager initialisation, inherently thread-safe.  
 • **Level 2 LazyLogger** – lazy, not thread-safe.  
 • **Level 3 ThreadSafeLogger** – double-checked locking with volatile.  
 3 – Refactor clean.App so it retrieves a logger via the chosen singleton and logs two messages, proving that hashCode() is identical on every call.  
 4 – Add a multi-threaded JUnit test (ThreadSafetyTest) that shows Level 2 can create more than one instance while Level 3 never does.  
 5 – reflection.md answer  
 • When would you prefer eager vs. lazy vs. thread-safe?  
 • What design smells can over-use of singletons introduce?  
 • How could dependency-injection improve testability here?

#### **Deliverables**

analysis/singleton\_problems.md

src/main/java/clean/EagerLogger.java

src/main/java/clean/LazyLogger.java

src/main/java/clean/ThreadSafeLogger.java

src/main/java/clean/App.java

src/test/java/clean/ThreadSafetyTest.java

reflection.md

README.md

### **Solution reference**

#### **Level 1 – Eager Singleton**

package clean;

/\*\* Level 1 – eager initialisation (class-loading time, thread-safe). \*/

public final class EagerLogger {

private static final EagerLogger INSTANCE = new EagerLogger();

private EagerLogger() {}

public static EagerLogger getInstance(){ return INSTANCE; }

public void log(String msg){ System.out.println("[Eager] "+msg); }

}

#### **Level 2 – Lazy Singleton (non-thread-safe)**

package clean;

/\*\* Level 2 – lazy initialisation, NOT thread-safe. \*/

public final class LazyLogger {

private static LazyLogger instance;

private LazyLogger() {}

public static LazyLogger getInstance(){

if(instance == null){ instance = new LazyLogger(); }

return instance;

}

public void log(String msg){ System.out.println("[Lazy] "+msg); }

}

#### **Level 3 – Lazy Thread-Safe Singleton**

package clean;

/\*\* Level 3 – lazy + double-checked locking, thread-safe. \*/

public final class ThreadSafeLogger {

private static volatile ThreadSafeLogger instance;

private ThreadSafeLogger() {}

public static ThreadSafeLogger getInstance(){

if(instance == null){ // 1st check

synchronized(ThreadSafeLogger.class){

if(instance == null){ // 2nd check

instance = new ThreadSafeLogger();

}

}

}

return instance;

}

public void log(String msg){ System.out.println("[TS] "+msg); }

}

#### **Client**

package clean;

public class App {

public static void main(String[] args){

ThreadSafeLogger logger1 = ThreadSafeLogger.getInstance();

ThreadSafeLogger logger2 = ThreadSafeLogger.getInstance();

logger1.log("hello singleton");

logger2.log("same instance? "+(logger1==logger2));

System.out.println("hashCodes: "+logger1.hashCode()+" "+logger2.hashCode());

}

}

#### **Thread-Safety Test (excerpt)**

@Test

public void lazyLogger\_canCreateMultipleInstances() throws Exception {

Set<Integer> hashes = ConcurrentHashMap.newKeySet();

IntStream.range(0,100).parallel().forEach(i ->

hashes.add( LazyLogger.getInstance().hashCode() ));

assertTrue(hashes.size() > 1); // failure proves non-thread-safe

}

@Test

public void threadSafeLogger\_singleInstance\_always() throws Exception {

Set<Integer> hashes = ConcurrentHashMap.newKeySet();

IntStream.range(0,100).parallel().forEach(i ->

hashes.add( ThreadSafeLogger.getInstance().hashCode() ));

assertEquals(1, hashes.size()); // always one instance

}

#### **Solution levels at a glance**

| **Level** | **Technique** | **Thread-safe?** | **Memory cost** | **Typical use-case** |
| --- | --- | --- | --- | --- |
| **1** | static final instance | Yes (class-loader) | instance always allocated | cheap, immutable config |
| **2** | lazy with null check | No | only when first used | single-thread apps or demos |
| **3** | double-checked locking + volatile | Yes | only when first used | production, multi-thread |

These three levels illustrate the evolutionary path of a Singleton: from simplest (Level 1) to safest (Level 3).