

Singleton in Java Level 1



Singleton Method

Now we go level by level growth like a game on this Java topic, provide coding example with its explanation, real-world use case example, necessary hints, points to remember, usual mistakes, etc.

start lvl1

Alright Divakar! Let's start **Singleton in Java** from **Level 1** and go step by step, like a game. I'll give **code examples, explanations, real-world use cases, hints, common mistakes**, etc.

Singleton - Level 1 (Basic Concept)

Definition:

A Singleton is a design pattern that ensures **only one instance of a class exists** in the application and provides a **global point of access** to it.

Think of it as a **single control room** in a building. You don't need multiple control rooms; one is enough, and everyone refers to it.

Step 1: Basic Singleton (Lazy Initialization)

```
java

class Singleton {
    // Step 1: private static instance (initially null)
    private static Singleton instance;

    // Step 2: private constructor (prevents external instantiation)
    private Singleton() {
        System.out.println("Singleton instance created!");
    }

    // Step 3: public static method to provide access
    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton(); // create instance only if null
        }
        return instance;
    }
}

public class Main {
    public static void main(String[] args) {
        Singleton s1 = Singleton.getInstance(); // Singleton instance created!
        Singleton s2 = Singleton.getInstance();

        System.out.println(s1 == s2); // true → both references point to same object
    }
}
```

Explanation (Level 1)

1. `private static Singleton instance;`
 - Holds the single instance of the class.
 2. `private Singleton()`
 - Prevents **external instantiation** using `new`.
 3. `public static Singleton getInstance()`
 - Provides **global access**.
 - Only creates a new object if one doesn't already exist (**lazy initialization**).
 4. `s1 == s2`
 - Verifies that both references point to the **same object**.
-

Real-world Use Case (Level 1)

- **Logger class** → You only want **one logger object** writing logs to a file.
 - **Database connection pool manager** → Only one instance manages the connection pool.
 - **Configuration manager** → Single point to read app configs.
-

Hints / Points to Remember

- Always make the constructor **private**.
 - Provide **static method** to access the instance.
 - **Lazy initialization** creates the instance **only when needed**.
-

Usual Mistakes (Level 1)

1. Making the constructor public → breaks singleton principle.
 2. Creating multiple instances using `new` elsewhere.
 3. Forgetting **static** for the instance → won't be global.
-

Level 1 Summary:

- Singleton ensures **one object only**.
 - Access via **static method**.
 - Lazy initialization creates object **when first used**.
-

If you want, I can take you to **Level 2 – Thread-safe Singleton**, which is the next step for real-world multi-threaded applications.

Do you want to go to **Level 2**?



next

Perfect! Let's move to **Level 2 – Thread-Safe Singleton**. This is essential because in real-world applications, **multiple threads may try to access `getInstance()` simultaneously**, which can break the singleton principle if not handled properly.

Singleton – Level 2 (Thread-Safe Singleton)

Problem in Level 1

In **Level 1**, the lazy initialization is **not thread-safe**:

```
java
Singleton s = Singleton.getInstance();
```

- If **Thread A** and **Thread B** call `getInstance()` at the same time, **two instances may be created**.
- We need **thread synchronization**.

Solution 1: Synchronized Method (Simple Thread-Safety)

```
java
class Singleton {
    private static Singleton instance;

    private Singleton() {
        System.out.println("Singleton instance created!");
    }

    public static synchronized Singleton getInstance() { // synchronized ensures thread-safety
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}

public class Main {
    public static void main(String[] args) {
        Runnable task = () -> {
            Singleton s = Singleton.getInstance();
            System.out.println(s);
        };

        Thread t1 = new Thread(task);
        Thread t2 = new Thread(task);

        t1.start();
        t2.start();
    }
}
```

Output Example:

```
graphql
```

```
Singleton instance created!
Singleton@1a2b3c
Singleton@1a2b3c
```

- **Only one instance** is created, even with multiple threads.

Solution 2: Double-Checked Locking (Efficient)

- Synchronizing the whole method is **slow**.
- Instead, we can **synchronize only when the instance is null**.

```
java

class Singleton {
    private static volatile Singleton instance; // volatile ensures visibility across threads

    private Singleton() {}

    public static Singleton getInstance() {
        if (instance == null) { // 1st check
            synchronized (Singleton.class) {
                if (instance == null) { // 2nd check
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

Why **volatile**?

- Prevents **instruction reordering** issues in multithreaded environments.

Real-world Use Case (Level 2)

- **Database Connection Pool** → multiple threads may request connections simultaneously.
- **Logger** → multi-threaded applications need a single logger object.
- **Cache Manager** → shared cache access by multiple threads.

Hints / Points to Remember

1. **Synchronized method** → simple, but slow for high concurrency.
2. **Double-checked locking** → efficient for performance.
3. **volatile** keyword is mandatory for double-checked locking in Java 1.5+.
4. Singleton should **prevent multiple instantiations** in multi-threaded environments.

Usual Mistakes

- Forgetting **volatile** → may still create multiple instances.
- Synchronizing only the block but missing the first null check → less efficient.
- Using singleton for objects that **must have multiple instances** → design error.

Level 2 Summary:

- Level 1 singleton is **not thread-safe**.
- Level 2 fixes it using **synchronized** or **double-checked locking**.
- Critical for **multi-threaded applications** like DB, logging, caching.

Divakar, the next step is **Level 3 – Eager Initialization & Static Block Singleton**, which is simpler and doesn't need synchronization.

Do you want to go **Level 3**?



diff between sync method and block

Absolutely, Divakar! Let's break this down **carefully**, because this is a common point of confusion in Java multithreading.

Synchronized Method vs Synchronized Block

1. Synchronized Method

```
java

public synchronized void doSomething() {
    // critical section
    System.out.println("Thread-safe code");
}
```

Characteristics:

- Locks the **entire method** for the current object (or class if static).
- **Any thread** trying to enter the method must wait if another thread is already executing it.
- **Simpler to write**, but can be **slower**, because it locks **everything in the method**, even code that doesn't need synchronization.

Example:

```
java

class Counter {
    private int count = 0;

    public synchronized void increment() {
```

```

        count++; // critical section
    }

    public int getCount() {
        return count; // not synchronized
    }
}

```

2. Synchronized Block

```

java

public void doSomething() {
    // non-critical code

    synchronized(this) {
        // critical section
        System.out.println("Thread-safe code");
    }

    // non-critical code
}

```

Characteristics:

- Locks **only the block of code** inside `synchronized`.
- More **efficient**, because you don't lock the **entire method** unnecessarily.
- You can also lock on **specific objects**, not just `this`.

Example:

```

java

class Counter {
    private int count = 0;
    private final Object lock = new Object();

    public void increment() {
        // some non-critical operations

        synchronized(lock) {
            count++; // critical section
        }

        // some more non-critical operations
    }
}

```

Key Differences

Feature	Synchronized Method	Synchronized Block
Scope	Entire method	Only specified block
Performance	Slower if method has non-critical code	Faster, locks only what is needed
Flexibility	Cannot choose object to lock	Can lock on any object
Syntax	Simple to write	Slightly more code

Real-world Analogy

- **Method** → Lock the **whole room** even if you need only the desk.
- **Block** → Lock **just the desk**, others can still use the room.

Hint for Singleton

- **Synchronized method** → simplest thread-safe singleton.
- **Synchronized block (double-checked locking)** → efficient for high-performance multithreading.

If you want, I can **draw a diagram showing the method vs block locking and threads waiting**, which makes it super clear visually.

Do you want me to do that?



next

Great! Let's move to **Level 3 – Eager Initialization & Static Block Singleton**. This is simpler than thread-safe lazy initialization and works well when you **know the singleton will always be used**.

Singleton – Level 3 (Eager Initialization)

1. Eager Initialization

```
java

class Singleton {
    // Instance is created at the time of class loading
    private static final Singleton instance = new Singleton();

    private Singleton() {
        System.out.println("Singleton instance created!");
    }

    public static Singleton getInstance() {
        return instance;
    }
}

public class Main {
    public static void main(String[] args) {
        Singleton s1 = Singleton.getInstance();
        Singleton s2 = Singleton.getInstance();

        System.out.println(s1 == s2); // true
    }
}
```

Explanation:

- The instance is created **when the class is loaded**, so it's **thread-safe** by default.

- No need for **synchronized**, because **class loading itself is thread-safe** in Java.

2. Static Block Initialization

- Similar to eager, but allows **exception handling** during instance creation.

```
java

class Singleton {
    private static final Singleton instance;

    static {
        Singleton temp = null;
        try {
            temp = new Singleton();
        } catch (Exception e) {
            System.out.println("Exception during Singleton creation");
        }
        instance = temp;
    }

    private Singleton() {
        System.out.println("Singleton instance created!");
    }

    public static Singleton getInstance() {
        return instance;
    }
}
```

Use case:

- When **constructor might throw exceptions**, static block allows **handling them safely**.

Real-world Use Cases (Level 3)

- **Configuration Manager** → always required, load at startup.
- **Application Logger** → always used, no lazy initialization needed.
- **Cache Loader** → pre-load cache at application start.

Hints / Points to Remember

1. Eager initialization is **simple and thread-safe**.
2. Static block allows **exception handling** during creation.
3. Not suitable if object creation is **heavy** and may not be used immediately.

Usual Mistakes

- Forgetting **private constructor** → breaks singleton.
- Using eager initialization for **resource-heavy objects** → may slow startup unnecessarily.
- Ignoring exception handling if the constructor can fail.

✓ Level 3 Summary:

- Eager initialization → thread-safe **without synchronization**.
- Static block → same as eager but allows **exceptions**.
- Best for **lightweight, always-needed singletons**.

Divakar, the next step is **Level 4 – Singleton using Bill Pugh (Inner Static Helper Class)**, which is the **best of lazy + thread-safe**.

Do you want to go **Level 4?**



next

Perfect! Let's go to **Level 4 – Singleton using Bill Pugh (Inner Static Helper Class)**. This is considered the **best approach** for lazy-loaded, thread-safe singletons in Java.

Singleton – Level 4 (Bill Pugh / Inner Static Helper Class)

Concept

- Uses a **static inner class** to hold the singleton instance.
- The instance is **created only when the inner class is loaded**.
- Thread-safety is ensured by **Java class loading mechanism**.

Code Example

```
java

class Singleton {

    private Singleton() {
        System.out.println("Singleton instance created!");
    }

    // Inner static helper class
    private static class SingletonHelper {
        private static final Singleton INSTANCE = new Singleton();
    }

    public static Singleton getInstance() {
        return SingletonHelper.INSTANCE; // loaded only when this method is called
    }
}

public class Main {
    public static void main(String[] args) {
        System.out.println("Program started");

        Singleton s1 = Singleton.getInstance();
        Singleton s2 = Singleton.getInstance();

        System.out.println(s1 == s2); // true
    }
}
```

```

    }
}
```

Explanation

1. **SingletonHelper** is **not loaded** until `getInstance()` is called.
2. JVM ensures **thread-safety** when loading static classes.
3. Provides **lazy initialization without synchronization overhead**.

Real-world Use Case

- **Database Connection Pool** → initialized only when needed.
- **Logger** → create instance only when logging first occurs.
- **Configuration Manager** → lazy-loaded, thread-safe, efficient.

Advantages

- Lazy-loaded → created **only when needed**.
- Thread-safe → no need for **synchronized**.
- Efficient → no synchronization overhead like double-checked locking.
- Clean and readable → avoids complex syntax.

Points to Remember / Hints

1. Inner static class is loaded **only when referenced**.
2. Guarantees **one instance** in multithreaded environment.
3. Best practice for most real-world Java singleton use cases.

Usual Mistakes

- Trying to access **SingletonHelper** directly → it should be **private**.
- Making constructor **public** → breaks singleton.
- Ignoring lazy vs eager choice → for heavy objects, lazy is better.

Level 4 Summary:

- Best combination of **lazy initialization + thread-safety + performance**.
- No **synchronized** needed.
- Clean, safe, and widely used in production.

Divakar, the next level is **Level 5 – Singleton with Enum**, which is the **most concise and secure approach** (prevents serialization & reflection attacks).

Do you want to go **Level 5**?



next

Awesome! Let's move to **Level 5 – Singleton using Enum**, which is considered the **most robust way** to implement Singleton in Java.

Singleton – Level 5 (Enum Singleton)

Concept

- Java **enums** are **implicitly final and serializable**.
- JVM ensures **only one instance per enum value**.
- Handles **serialization** and **reflection attacks** automatically.

Code Example

```
java

enum Singleton {
    INSTANCE; // single instance

    public void showMessage() {
        System.out.println("Hello from Enum Singleton!");
    }
}

public class Main {
    public static void main(String[] args) {
        Singleton s1 = Singleton.INSTANCE;
        Singleton s2 = Singleton.INSTANCE;

        s1.showMessage();

        System.out.println(s1 == s2); // true → both references point to the same instance
    }
}
```

Explanation

- INSTANCE** is **the single instance** of the enum.
- JVM guarantees **thread-safety** for enum creation.
- Prevents **multiple instances** via serialization or reflection.

Why Enum Singleton is Safe

- **Serialization Safe:**

Normally, serialization can create **new instances**, breaking singleton. Enum prevents this.

- **Reflection Safe:**

Reflection cannot instantiate enums.

- **Thread-Safe:**

Enum instances are created **when JVM loads enum**, so inherently thread-safe.

Real-world Use Cases

- **Java Logging frameworks** → single logger instance.
 - **Configuration Managers** → global access point.
 - **State Machines** → enums already represent states, can also ensure singleton.
-

Advantages

1. **Simple syntax** → fewer lines, very readable.
 2. **Thread-safe** → no synchronization needed.
 3. **Serialization-safe** → no extra handling required.
 4. **Reflection-safe** → cannot create extra instances.
-

Points to Remember / Hints

- Enum singleton is the **strongest way** to implement singleton in Java.
 - Good for cases where **security & serialization** are concerns.
 - Can define **methods and variables** inside enum like a normal class.
-

Usual Mistakes

- Trying to create a **new instance using new** → not allowed.
 - Forgetting that enums are **implicitly final**, so cannot extend.
 - Treating enum like a regular class → don't use multiple instances, always use **INSTANCE**.
-

Level 5 Summary:

- Enum singleton is **thread-safe, serialization-safe, reflection-safe**.
 - Simplest and most robust singleton implementation.
 - Recommended by **Effective Java** (Joshua Bloch).
-

Divakar, if you want, I can make a **single summary table of all 5 levels of Singleton**, showing **lazy vs eager, thread-safety, pros, cons, and usage**, which will make it very easy to remember for interviews.

Do you want me to do that?



real world code example

Absolutely, Divakar! Let's take a **real-world scenario** where singleton is very practical. We'll use a **Logger** as an example. This is one of the most common use cases in enterprise applications.

Real-World Example: Logger Singleton

Scenario

You want **all parts of your application** to use a **single logger instance** that writes logs to a file.

- Multiple threads may try to log at the same time → needs **thread-safety**.
- Only **one file writer** should exist → singleton ensures single point of access.

Code Example: Logger using Bill Pugh Singleton

```
java

import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.time.LocalDateTime;

// Singleton Logger
class Logger {

    private Logger() {
        System.out.println("Logger initialized!");
    }

    // Inner static helper class ensures lazy and thread-safe initialization
    private static class LoggerHelper {
        private static final Logger INSTANCE = new Logger();
    }

    public static Logger getInstance() {
        return LoggerHelper.INSTANCE;
    }

    // Method to log messages
    public synchronized void log(String message) { // synchronized to prevent garbled logs
        try (FileWriter fw = new FileWriter("app.log", true);
             PrintWriter pw = new PrintWriter(fw)) {
            pw.println(LocalDateTime.now() + " : " + message);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

// Main application
public class Main {
    public static void main(String[] args) {
        Runnable task = () -> {
            Logger logger = Logger.getInstance();
            logger.log(Thread.currentThread().getName() + " logging message");
        };
    }
}
```

```

};

Thread t1 = new Thread(task, "Thread-1");
Thread t2 = new Thread(task, "Thread-2");

t1.start();
t2.start();
}
}

```

Explanation

1. **Singleton ensures single logger instance** across the app.
2. **Inner static helper class** makes it **lazy-loaded and thread-safe**.
3. **synchronized log() method** ensures multiple threads writing simultaneously **don't corrupt the file**.
4. Each thread uses `Logger.getInstance()` → always the **same instance**.

Output (app.log)

```

makefile

2025-10-03T12:10:01 : Thread-1 logging message
2025-10-03T12:10:01 : Thread-2 logging message

```

Why Singleton is perfect here

- **Only one file writer object** → prevents multiple open files.
- **Global access point** → anywhere in the app, you just call `Logger.getInstance()`.
- **Thread-safe** → prevents concurrent issues in multithreaded applications.

Real-world Applications

1. **Logging frameworks** → Log4j, SLF4J, etc., internally use singleton for loggers.
2. **Database connection pools** → single manager controlling connections.
3. **Cache manager** → centralized cache for the whole application.
4. **Configuration manager** → read configs once, shared globally.

If you want, I can make an **even bigger real-world example** like a **Database Connection Pool using Singleton** which combines **multi-threading + lazy initialization + resource management**, just like production-level code.

Do you want me to do that?

