**Levels of Locking in Java**

In multi-threaded programming, **locking** is a way to control how multiple threads access shared resources, like variables or methods. This helps prevent problems like data corruption or incorrect results. Java has two main types of locks:

1. **Object-Level Locking (Instance Locking)**
2. **Class-Level Locking (Static Locking)**

Each of these locks is useful in different scenarios, depending on whether you want to control access to a single object or to a method that all instances of a class share.

Let’s begin with **Object-Level Locking (Instance Locking)**

**What is Object Level Locking:**This type of lock is used for non-static methods or code blocks. It ensures that only one thread can run a synchronized method or block on a specific instance of a class at a time.

Each object has its own lock, allowing different instances of the class to work independently.

**Real World Analogy:** Imagine two bathrooms in an office. Each bathroom is locked by the person using it. Even if someone is using Bathroom A, someone else can still use Bathroom B. Here, each bathroom (object) has its own lock, and each person (thread) waits only if their specific bathroom is occupied.

**Why Use Object-Level Locking?**

To prevent thread interference when working with an object’s data.

To allow multiple objects of the same class to operate independently in different threads.

**Example Code (Object-Level Locking) by Synchronized Block:**

package synchronization.test;

public class ObjectLocking implements Runnable {

public void print() {

// Synchronized block to achieve object-level locking

synchronized (this) { // Locks the current object instance (ol)

for (int i = 0; i < 5; i++) {

String name = Thread.currentThread().getName(); // Get the current thread's name

System.out.println(name); // Print the thread name

try {

Thread.sleep(500); // Pause execution for 0.5 seconds

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

@Override

public void run() {

// Call the print method in the run method

print();

}

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

ObjectLocking ol = new ObjectLocking(); // Create a single instance of ObjectLocking

// Create two threads that will use the same ObjectLocking instance

Thread t1 = new Thread(ol, "Thread 1"); // Thread t1 created with the name "Thread 1"

Thread t2 = new Thread(ol, "Thread 2"); // Thread t2 created with the name "Thread 2"

// Start both threads

t1.start();

t2.start();

}

}

### **Our Expected Output**



[Thread1]

[Thread1]

[Thread2]

[Thread2]

[Thread2]

[Thread2]

[Thread2]



**Let’s see What’s Happening in the Code**

**Object-Level Locking Using a Synchronized Block**:

Instead of using synchronized on the whole print() method, we use a **synchronized block** inside the print() method.

synchronized (this) means that only the code within this block will be locked, and the lock will apply to the current object instance (this), which is ol in this example.

**Single Instance, Multiple Threads**:

The main method creates a single instance of ObjectLocking, called ol.

Two threads, t1 and t2, are created using the same ol instance.

Both threads access the print method of this shared instance, ol.

**How the Synchronized Block Works**:

When a thread (say, t1) enters the synchronized block in print(), it locks the ol object, preventing any other thread (like t2) from entering the block on the same instance.

While t1 is inside the synchronized block, t2 has to wait.

Once t1 completes the block and releases the lock, t2 can then enter the block and execute print().

**Thread Sleep and Output**:

Inside print(), each thread prints its name five times, with a half-second pause between each print.

The pause allows us to see clearly that only one thread accesses the synchronized block at a time.

**Object-Level Locking by Locking a Method**

In this example, we use the synchronized keyword on the showMessage method to ensure only one thread can execute this method on a specific object instance at a time.

**Code Example (Object-Level Locking with Synchronized Method)**

package synchronization.test;

public class ObjectLocking implements Runnable {

// Synchronized method to achieve object-level locking

public synchronized void print() {

// Loop to print the current thread's name multiple times

for (int i = 0; i < 5; i++) {

String name = Thread.currentThread().getName(); // Get the current thread's name

System.out.println(name); // Print the thread name

try {

Thread.sleep(500); // Pause execution for 0.5 seconds

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

@Override

public void run() {

// Call the synchronized print method in the run method

print();

}

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

ObjectLocking ol = new ObjectLocking(); // Create a single instance of ObjectLocking

// Create two threads that will use the same ObjectLocking instance

Thread t1 = new Thread(ol, "Thread 1"); // Thread t1 created with the name "Thread 1"

Thread t2 = new Thread(ol, "Thread 2"); // Thread t2 created with the name "Thread 2"

// Start both threads

t1.start();

t2.start();

}

}



Expected output

[Thread1]

[Thread1]

[Thread1]

[Thread1]

[Thread1]

[Thread2]

[Thread2]

[Thread2]

[Thread2]

[Thread2]

### 

**What is Happening in the Above Code?**

**Object-Level Locking on print() Method**:

The print() method is marked synchronized, so it locks on the specific ObjectLocking instance (ol).

Only one thread can access print() at a time on this shared instance.

**What Happens**:

Thread t1 acquires the lock on ol and begins executing print().

While t1 is executing print(), t2 has to wait until t1 releases the lock.

Once t1 finishes and exits print(), it releases the lock, allowing t2 to access print().

**Result**:

This synchronization ensures that the threads do not interfere with each other’s execution on the same object.

Both threads print their names sequentially (one after the other) rather than concurrently, because only one thread can enter print() at a time on the ol instance.

Now let’s see the Class-level locking

**What is Class level locking ?**

**Answer: Class-Level Locking (Static Synchronization):** If we use a synchronized block or method on a static method, it locks the class instead of the object, ensuring that the lock is across all instances of that class.

**Class-Level Locking Using a Synchronized Block**

In this example, we lock on the StaticDisplay.class object (which represents the class itself) inside a synchronized block. This approach allows us to lock only the critical section of code instead of the entire method.

**Code Example (Class-Level Locking with Synchronized Block)**

class StaticDisplay {

// Static method that uses a synchronized block for class-level locking

public void showMessage(String message) {

synchronized (StaticDisplay.class) { // Locks the StaticDisplay class

for (int i = 0; i < 5; i++) {

System.out.print("[");

System.out.print(message);

System.out.println("]");

try {

Thread.sleep(500); // Sleep for half a second

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

}

class StaticMessageThread extends Thread {

private String message;

// Constructor to initialize the message

public StaticMessageThread(String message) {

this.message = message;

}

// Overridden run method to call the static showMessage method of StaticDisplay

@Override

public void run() {

StaticDisplay.showMessage(message);

}

}

public class ClassLevelLockDemo {

public static void main(String[] args) {

// Creating two threads that will use the same class-level lock

StaticMessageThread t1 = new StaticMessageThread("Thread1");

StaticMessageThread t2 = new StaticMessageThread("Thread2");

t1.start(); // Start thread t1

t2.start(); // Start thread t2

}

}

Expected Output

[Thread1]

[Thread1]

[Thread1]

[Thread1]

[Thread1]

[Thread2]

[Thread2]

[Thread2]

[Thread2]

[Thread2]

### 

**What is happening in the Code:**

By using synchronized (StaticDisplay.class), we lock the StaticDisplay class itself, rather than an instance.

This means only one thread can enter the synchronized block, even if different threads or different instances are used.

Thus, t1 and t2 must wait for each other, producing a sequential output for each thread.

Let’s use Synchronized Block

**Class-Level Locking using Synchronized Method**

In class-level locking, a single lock applies to all instances of the class. This means that when one thread accesses a static synchronized method, all other threads (even on different instances) must wait until the lock is released.

**Code Example (Class-Level Locking using Synchronized Method)**

class StaticDisplay {

// Static synchronized method for class-level locking

public static synchronized void showMessage(String message) {

for (int i = 0; i < 5; i++) {

System.out.print("[");

System.out.print(message);

System.out.println("]");

try {

Thread.sleep(500); // Sleep for half a second

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

class StaticMessageThread extends Thread {

private String message;

// Constructor to initialize the message

public StaticMessageThread(String message) {

this.message = message;

}

// Overridden run method to call the static showMessage method of StaticDisplay

@Override

public void run() {

StaticDisplay.showMessage(message);

}

}

public class ClassLevelLockDemo {

public static void main(String[] args) {

// Creating two threads that will use the same class-level lock

StaticMessageThread t1 = new StaticMessageThread("Thread1");

StaticMessageThread t2 = new StaticMessageThread("Thread2");

t1.start(); // Start thread t1

t2.start(); // Start thread t2

}

}



Expected Output

[Thread1]

[Thread1]

[Thread1]

[Thread1]

[Thread1]

[Thread2]

[Thread2]

[Thread2]

[Thread2]

[Thread2]



**What is happening in the Code:**

**What Happens:** Here, showMessage is a static synchronized method, meaning it locks at the class level (StaticDisplay class) rather than at the instance level. When t1 calls showMessage, it locks the entire class, so t2 must wait until t1 finishes before it can access showMessage.

**Result:** The output is sequential (first all messages from t1, then all from t2) because both threads are using a single lock that applies to the entire class, not individual instances.

**Comparison of Object-Level Locking vs Class-Level Locking**

| **Feature** | **Object-Level Locking** | **Class-Level Locking** |
| --- | --- | --- |
| **Lock Type** | Instance-specific lock | Class-wide lock |
| **Code Structure** | Uses synchronized(this) or synchronized block | Uses static synchronized on methods |
| **Concurrency** | Threads on different instances can run concurrently | All threads must wait regardless of instance |
| **Usage Scenario** | When each instance has independent data | When data is shared among all instances |
| **Example Analogy** | Two bathrooms in different parts of a building | One common power source in a room |

**Real-World Usage**

**Object-Level Locking** is ideal when:

You have multiple independent objects that should be accessed independently.

**Example**: Two different bank accounts (objects) where each account should be managed independently of the other.

**Class-Level Locking** is useful when:

You need to control access to shared resources across all instances.

**Example**: A logging system where every instance logs to the same file. Only one thread should access the file at a time to avoid mixing logs.

Our final conclusions

**Object-Level Locking** allows multiple threads to access different instances concurrently without interference.

**Class-Level Locking** ensures only one thread can access a static synchronized method, even if different instances are used, enforcing stricter control across the entire class.

**Advanced Concept: Reentrant Locks (Lock Interface)**

A **Reentrant Lock** is a more flexible way to control access to shared resources compared to synchronized blocks. It allows a thread to re-enter a lock it already holds, providing more control over resource access. Additionally, it offers advanced features like timeout-based locking, enabling threads to try to acquire a lock for a specific duration, making it a powerful tool for managing concurrent access to shared resources in complex applications.

**Real World Analogy:** Imagine a popular toy store. Many kids want to play with the same toy. To avoid fights, the store has a special system:

**The Special Key:** There's a special key that only one kid can hold at a time.

**Playing with the Toy:** When a kid gets the key, they can play with the toy.

**Leaving and Coming Back:** If the kid needs to leave the toy for a while, they can keep the key. When they come back, they can use the same key to continue playing.

**Waiting for the Key:** If another kid wants to play, they have to wait until the first kid returns the key.

**This special key system is like a Reentrant Lock.**

**Why is it useful?**

**Fair Play:** Ensures that only one kid plays at a time, preventing arguments.

**Flexible Rules:** The store can set rules like "You can only hold the key for 10 minutes" or "If you don't use the key for 5 minutes, someone else can take it."

**Efficient Play:** Kids don't have to wait for long periods, making the play more efficient.

**In computer terms:**

**Reentrant Lock:** A special tool to control access to a shared resource (like the toy).

**Thread:** A kid who wants to use the resource.

**Acquire the Lock:** Getting the key.

**Release the Lock:** Returning the key.

**Where to Find It:**

Reentrant Locks are part of the java.util.concurrent.locks package in the Java Development Kit (JDK).

**How to Use It:**

**Create a Reentrant Lock Objecct:**

ReentrantLock lock = new ReentrantLock();

**Acquiring the Lock by using lock() method**

lock.lock();

**Releasing the Lock using unlock() method**

lock.unlock();

**Acquiring the Lock with a Timeout using tryLock() method:**

boolean acquired = lock.tryLock(10, TimeUnit.SECONDS);

**Example Code (Reentrant Lock):**

package synchronization.test;

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;

// Class to demonstrate Reentrant Locking in Java

class ReentrantLocks {

private int count = 0; // Shared variable that will be incremented by threads

private final Lock lock = new ReentrantLock(); // ReentrantLock to control access to 'count'

// Method to increment 'count' with locking for thread safety

public void increment() {

lock.lock(); // Acquire the lock before entering the critical section

try {

count++; // Safely increment the count variable

} finally {

lock.unlock(); // Release the lock to allow other threads to access

}

}

// Method to get the current value of 'count'

public int getCount() {

return count;

}

}

// Thread class that uses ReentrantLocks to safely increment and print 'count'

class IncrementThread extends Thread {

private ReentrantLocks reentrantLocks; // Shared ReentrantLocks instance

// Constructor to pass the shared ReentrantLocks instance

public IncrementThread(ReentrantLocks reentrantLocks, String threadName) {

this.reentrantLocks = reentrantLocks;

this.setName(threadName); // Set the thread's name for easy identification

}

// Run method that performs the increment operation

@Override

public void run() {

reentrantLocks.increment(); // Call increment method

System.out.println(this.getName() + " Count: " + reentrantLocks.getCount()); // Print the result

}

}

// Main class to create and start threads

public class ReentrantLockDemo {

public static void main(String[] args) {

ReentrantLocks rl = new ReentrantLocks(); // Create one instance of ReentrantLocks

// Create three threads, each using the same ReentrantLocks instance

IncrementThread t1 = new IncrementThread(rl, "Thread 1");

IncrementThread t2 = new IncrementThread(rl, "Thread 2");

IncrementThread t3 = new IncrementThread(rl, "Thread 3");

// Start the threads

t1.start();

t2.start();

t3.start();

}

}

### 

**Code Explanation:**The ReentrantLock allows each thread to safely increment count without interference. If a thread holds the lock, no other thread can access increment until the lock is released. This ensures an orderly update to count without conflicts.