SYNCHRONIZATION IN JAVA: CONCEPTS AND BEST PRACTICES

Sandra Kumi

REVIEWER Mr. Thomas Darko

Synchronization in Java: Concepts and Best Practices

Introduction to Synchronization

Synchronization is a fundamental concept in multithreading that ensures the safe execution of code when multiple threads interact with shared resources. Without synchronization, two or more threads could access and modify the same data concurrently, leading to race conditions and inconsistent results.

Java provides various mechanisms to synchronize the access of shared resources to maintain data integrity and ensure thread safety.

Synchronized Methods

A synchronized method in Java ensures that only one thread at a time can execute that method for a given object. This is useful for ensuring exclusive access to critical sections that modify shared resources.

Example: Synchronized Method

```
class Counter {

    private int count = 0;



    public synchronized void increment() {

        count++;

    }
    public int getCount() {
        return count;
    }
}
```

In the above example, the increment method is synchronized, which means that if one thread is executing the method, no other thread can enter it until the first thread exits.

Use Case:

Synchronized methods are used when the entire method needs to be thread-safe, especially when dealing with critical data like counters, balances, or collections that multiple threads modify.

Synchronized Blocks

Sometimes, synchronizing the entire method can be overkill, especially when only a part of the method modifies the shared resource. Synchronized blocks allow us to synchronize only the critical section of the code, improving performance by minimizing the time the lock is held.

Example: Synchronized Block

```
class Counter {
    private int count = 0;

    public void increment() {
        synchronized (this) {

            count++;

        }

    }



    public int getCount() {

        return count;

    }

}
```

In this example, only the block that modifies the count variable is synchronized. This allows the rest of the method to be executed by multiple threads simultaneously without restriction.

Use Case:

Synchronized blocks are more efficient than synchronized methods when only a portion of the code needs protection.

Reentrant Locks

While synchronized provides a basic level of synchronization, Java also offers more advanced mechanisms such as ReentrantLock from the java.util.concurrent.locks package. A ReentrantLock provides more flexibility, allowing features like:

Fairness policies (first-come, first-served lock acquisition).

Explicit lock acquisition and release.

Ability to check if the lock is available (tryLock).

Example: Reentrant Lock

```
import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;



class SharedResource {

    private final Lock lock = new ReentrantLock();



    public void safeMethod() {

        lock.lock();

        try {

            // critical section

        } finally {
            lock.unlock();

        }

    }

}
```

Use Case:

ReentrantLocks are used when you need more control over the locking mechanism, such as the ability to interrupt threads waiting on a lock or to attempt lock acquisition with timeouts.

Condition Variables

Condition variables allow threads to wait for certain conditions to become true. They are used in conjunction with locks to coordinate thread execution more precisely than with synchronized.

Example: Condition Variable

```
import java.util.concurrent.locks.Condition;

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;



class SharedResource {

    private Lock lock = new ReentrantLock();
    private Condition condition = lock.newCondition();


    public void waitForCondition() throws InterruptedException {

        lock.lock();

        try {

            condition.await();  // wait for a signal

        } finally {

            lock.unlock();

        }

    }


    public void signalCondition() {

        lock.lock();

        try {

            condition.signal();  // notify waiting threads
        } finally {
            lock.unlock();
        }
    }
}
```

Use Case:

Condition variables are used for more complex thread coordination, such as implementing producer-consumer models or signaling between threads based on certain conditions.

Deadlock and Prevention

A deadlock occurs when two or more threads are blocked forever, each waiting on the other to release a resource. To avoid deadlock, several strategies can be employed:

Acquiring locks in a consistent order across all threads.

Using tryLock() to acquire locks with timeouts, avoiding waiting indefinitely.

Reducing the scope of locks, ensuring that threads hold locks for only as long as necessary.

Example: Deadlock Prevention Using tryLock

```
import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;
```

```
class DeadlockSolution {

    private Lock lock1 = new ReentrantLock();

    private Lock lock2 = new ReentrantLock();



    public void acquireResources() throws InterruptedException {

        while (true) {

            if (lock1.tryLock()) {

                try {

                    if (lock2.tryLock()) {

                        try {

                            // critical section

                            break;

                        } finally {

                            lock2.unlock();
                        }
                    }
                } finally {
                    lock1.unlock();
                }
            }
            Thread.sleep(10);  // Wait before retrying
        }
    }
}
```

Use Case:

Deadlock prevention is critical when designing systems where multiple locks are required. Using tryLock and acquiring locks in a specific order can prevent circular dependencies between threads.

Best Practices for Synchronization

Minimize the scope of synchronization: Synchronize only the parts of the code that require it to reduce contention and improve performance.

Avoid unnecessary synchronization: Only synchronize shared resources. If a resource is thread-local, it does not need synchronization.

Prefer higher-level concurrency utilities: Whenever possible, use higher-level abstractions like java.util.concurrent classes (e.g., ConcurrentHashMap, BlockingQueue) instead of managing synchronization manually.

Use immutable objects: Immutable objects are inherently thread-safe and can reduce the need for synchronization.

Always release locks in a finally block: Ensure locks are always released by placing unlock() in a finally block to avoid lock leaks in the case of exceptions.

Avoid holding locks for long durations: Keep the locked section of code as short as possible to minimize blocking other threads.