Names: Nadege Gaju

Synchronization in Java

1. Introduction

Synchronization is the process of controlling access to shared resources by multiple threads. Without proper synchronization, race conditions may occur, leading to unpredictable results. Java provides several mechanisms to handle synchronization, including synchronized methods, synchronized blocks, Lock and Condition classes, and atomic variables.

Below, we will explore these synchronization techniques, discuss potential deadlock scenarios, and demonstrate how to use locks and atomic variables for thread-safe operations.

2. Synchronized Methods and Blocks

2.1 Synchronized Methods

In a synchronized method, only one thread can execute the method at any given time. It ensures that shared resources are not accessed concurrently by multiple threads, thereby preventing race conditions.

Code Example: Synchronized Method

```
class Counter {
private int count = 0;
public synchronized void increment() {
count++;
}
public int getCount() {
return count;
}
}
public class SynchronizedMethodDemo {
public static void main(String[] args) throws InterruptedException {
Counter counter = new Counter();
Runnable task = () -> {
        for (int i = 0; i < 1000; i++) {
            counter.increment();
```

```
            }
        };
 Thread t1 = new Thread(task);
 Thread t2 = new Thread(task);
 t1.start();
 t2.start();
 t1.join();
 t2.join();
 System.out.println("Final count: " + counter.getCount());
    }
}
```

In this example, the increment() method is synchronized, ensuring that only one thread can execute the method at a time.

2.2 Synchronized Blocks

Synchronized blocks allow more granular control by synchronizing only a specific section of the code rather than an entire method. This can help improve performance while still preventing race conditions.

Code Example: Synchronized Block

```
class Counter {
    private int count = 0;
    public void increment() {
        synchronized (this) {
            count++;
        }
    }
 public int getCount() {
        return count;
    }
}
```

```
public class SynchronizedBlockDemo {
public static void main(String[] args) throws InterruptedException {
Counter counter = new Counter();
Runnable task = () -> {
for (int i = 0; i < 1000; i++) {
counter.increment();
}
};
Thread t1 = new Thread(task);
Thread t2 = new Thread(task);
t1.start();
        t2.start();

        t1.join();
        t2.join();

        System.out.println("Final count: " + counter.getCount());
    }
}
```

3. Deadlock Scenarios and Prevention

3.1 Deadlock Scenario

A deadlock occurs when two or more threads are blocked forever, waiting for each other to release a lock. Deadlocks can happen when multiple threads acquire locks in different orders.

Code Example: Deadlock Scenario

```
public class DeadlockExample {
private final Object lock1 = new Object();
private final Object lock2 = new Object();
```

```
public void method1() {
synchronized (lock1) {
System.out.println("Thread 1: Holding lock 1...");
try { Thread.sleep(100); } catch (InterruptedException e) {}
synchronized (lock2) {
System.out.println("Thread 1: Holding lock 1 & 2...");
}
}
}

public void method2() {
synchronized (lock2) {
System.out.println("Thread 2: Holding lock 2...");
try { Thread.sleep(100); } catch (InterruptedException e) {}
synchronized (lock1) {
System.out.println("Thread 2: Holding lock 2 & 1...");
}
}
}

public static void main(String[] args) {
DeadlockExample deadlock = new DeadlockExample();
Thread t1 = new Thread(deadlock::method1);
Thread t2 = new Thread(deadlock::method2);
t1.start();
t2.start();
    }
}
```

In this scenario, both threads are waiting for each other, causing a deadlock.

3.2 Deadlock Prevention

To prevent deadlock, always acquire locks in a consistent order. This eliminates circular wait conditions.

Code Example: Deadlock Prevention

```
public class DeadlockPrevention {
private final Object lock1 = new Object();
private final Object lock2 = new Object();
public void method1() {
synchronized (lock1) {
System.out.println("Thread 1: Holding lock 1...");
synchronized (lock2) {
System.out.println("Thread 1: Holding lock 1 & 2...");
}
}
}

public void method2() {
synchronized (lock1) {
System.out.println("Thread 2: Holding lock 1...");
synchronized (lock2) {
System.out.println("Thread 2: Holding lock 1 & 2...");
}
}
}
public static void main(String[] args) {
DeadlockPrevention deadlockPrevention = new DeadlockPrevention();
Thread t1 = new Thread(deadlockPrevention::method1);
Thread t2 = new Thread(deadlockPrevention::method2);
```

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

4. Locks and Condition Variables

Java's Lock and Condition classes provide a more flexible alternative to synchronized methods and blocks. They allow you to coordinate thread execution more explicitly.

4.1 Using Locks and Conditions

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

class SharedResource {
private int count = 0;
private final Lock lock = new ReentrantLock();
private final Condition condition = lock.newCondition();
public void increment() {
lock.lock();
try {
count++;
condition.signalAll();
} finally {
lock.unlock();
}
   }
public void waitForCount(int target) throws InterruptedException {
       lock.lock();
```

```
try {
while (count < target) {
condition.await();
}
System.out.println("Reached target: " + count);
} finally {
lock.unlock();
}
}
}

public class LockConditionDemo {
public static void main(String[] args) {
SharedResource sharedResource = new SharedResource();
Thread incrementer = new Thread(() -> {
for (int i = 0; i < 5; i++) {
sharedResource.increment();
}
});

Thread waiter = new Thread(() -> {
try {
sharedResource.waitForCount(5);
} catch (InterruptedException e) {
e.printStackTrace();
}
});
```

```
waiter.start();
incrementer.start();
}
}
```

5. Atomic Variables

Atomic variables, such as AtomicInteger, provide a lock-free thread-safe mechanism for incrementing and updating variables.

5.1 Example: Using Atomic Variables

```
import java.util.concurrent.atomic.AtomicInteger;
class AtomicCounter {
private final AtomicInteger count = new AtomicInteger(0);
public void increment() {
count.incrementAndGet();
}
public int getCount() {
return count.get();
}
}
public class AtomicVariableDemo {
public static void main(String[] args) throws InterruptedException {
AtomicCounter counter = new AtomicCounter();
Runnable task = () -> {
for (int i = 0; i < 1000; i++) {
counter.increment();
}
};
Thread t1 = new Thread(task);
        Thread t2 = new Thread(task);
```

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

        t1.join();
        t2.join();

        System.out.println("Final count: " + counter.getCount());
    }
}
```

6. Best Practices for Synchronization

- ✓ **Use Synchronized Methods/Blocks Appropriately:**
  - ❖ Synchronized methods are simple, but sometimes synchronized blocks can be more efficient if only part of the method needs to be synchronized.
- ✓ **Avoid Deadlock:**
  - ❖ Always acquire locks in a consistent order to prevent deadlocks.
- ✓ **Use Locks and Conditions for Complex Scenarios:**
  - ❖ For complex thread synchronization, prefer Lock and Condition classes over synchronized.
- ✓ **Atomic Variables for Simple Lock-Free Synchronization:**
  - ❖ For simple operations, use atomic variables like AtomicInteger to avoid the overhead of explicit synchronization.

7. Conclusion

Synchronization is essential in a multithreaded environment to avoid race conditions and ensure data integrity. Java provides various mechanisms such as synchronized methods, blocks, locks, condition variables, and atomic variables to handle synchronization efficiently. It's important to understand these mechanisms and use them appropriately based on the complexity of the thread interaction.