JAVA MULTITHREADING CHALLENGE: OPTIMIZED APPROACHES

PROBLEM STATEMENT:

We need to create two threads:
- Thread A: Prints numbers from 1 to 5
- Thread B: Prints letters from A to E

The output must alternate like this: 1 A 2 B 3 C 4 D 5 E

STEP-BY-STEP EXPLANATION:

1. Traditional multithreading methods like `synchronized`, `wait()`, and `notify()` can cause unnecessary blocking.

2. A more efficient approach uses `Lock` and `Condition`, which allow fine-grained control over thread execution.

3. Another alternative is using `Semaphore`, which efficiently controls access between threads.

SOLUTION 1: USING LOCK AND CONDITION

In this approach, we use a `ReentrantLock` and two conditions to control execution between threads. Thread A prints a number, signals Thread B, and waits. Thread B prints a letter, signals Thread A, and waits.

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


public class AlternatingThreads {
    private static final Lock lock = new ReentrantLock();
    private static final Condition numberTurn = lock.newCondition();
    private static final Condition letterTurn = lock.newCondition();
    private static boolean isNumberTurn = true;

    public static void main(String[] args) {
        Thread threadA = new Thread(() -> printNumbers());
        Thread threadB = new Thread(() -> printLetters());

        threadA.start();
        threadB.start();
    }
```

```
    static void printNumbers() {
        for (int i = 1; i <= 5; i++) {
            lock.lock();
            try {
                while (!isNumberTurn) {
                    numberTurn.await();
                }
                System.out.print(i + " ");
                isNumberTurn = false;
                letterTurn.signal();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            } finally {
                lock.unlock();
            }
        }
    }

    static void printLetters() {
        for (char ch = 'A'; ch <= 'E'; ch++) {
            lock.lock();
            try {
                while (isNumberTurn) {
                    letterTurn.await();
                }
                System.out.print(ch + " ");
                isNumberTurn = true;
                numberTurn.signal();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            } finally {
                lock.unlock();
            }
        }
    }
}
```

SOLUTION 2: USING SEMAPHORE

A `Semaphore` is used to enforce alternating execution. Thread A starts, prints a number, then releases the semaphore for Thread B to continue. Thread B waits for the semaphore, prints a letter, and releases control back to Thread A.

```
import java.util.concurrent.Semaphore;
```

```java
public class AlternatingThreadsSemaphore {
    private static final Semaphore numSemaphore = new Semaphore(1);
    private static final Semaphore letterSemaphore = new Semaphore(0);

    public static void main(String[] args) {
        Thread threadA = new Thread(() -> printNumbers());
        Thread threadB = new Thread(() -> printLetters());

        threadA.start();
        threadB.start();
    }

    static void printNumbers() {
        for (int i = 1; i <= 5; i++) {
            try {
                numSemaphore.acquire();
                System.out.print(i + " ");
                letterSemaphore.release();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    }

    static void printLetters() {
        for (char ch = 'A'; ch <= 'E'; ch++) {
            try {
                letterSemaphore.acquire();
                System.out.print(ch + " ");
                numSemaphore.release();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    }
}
```