Programming Assignment 1

Task 1: The inconsistent account balances occur because multiple threads (Depositor and Withdrawer objects) access a shared Account object concurrently. In the Account class, the deposit() and withdraw() methods modify the shared balance variable using operations that are **not atomic**. These operations require multiple CPU cycles to complete, since the corresponding assembly instructions involve several steps (loading values into registers, performing arithmetic, and writing the result back to memory). During this process, the operating system's scheduler may perform a **context switch**, interrupting a thread before it finishes all the instructions. As a result, another thread may execute before the previous one has completed its update, leading to lost or overwritten updates. For example, a Depositor thread might load the balance into a register but be preempted before performing the addition, allowing a Withdrawer thread to execute and modify the same balance value, thus causing data inconsistency.

Task 2: The starting order of threads in Java is **non-deterministic** and controlled by the **underlying operating system's scheduler** and kernel. Even if threadA.start() is invoked before threadB.start(), there is no guarantee that threadA will execute first. The actual execution order depends on the scheduling algorithm used by the operating system and factors such as CPU availability and system load. Moreover, the observed inconsistency in account balances is **not caused by the starting order of threads**, but rather by a **lack of proper synchronization**. The critical section of the code, where the account balance is updated, is not atomic and can be **preempted** during execution. Therefore, even if a Depositor thread starts before a Withdrawer thread, the Depositor may be interrupted before completing its update, allowing the Withdrawer to run and modify the shared balance concurrently, leading to inconsistent results.

Task 3 The critical section of the code are within the following functions that I have pasted. The entire function is not the critical section, the critical section is only the lines

balance = balance + amount;

balance = balance - amount;

within these functions

```
    /**
     * A method that allows a customer to deposit money into this account
```

```
 * @param amount A double that represents a deposit amount
 */
public void debosit(double amount){

        // Waste some time doing fake computations
        // do not remove or modify any of the following 3 statements
        double k = 999999999;
        for(int i=0;i<100;i++)
                k = k / 2;

                balance = balance + amount;

        // Waste some time doing fake computations
        // do not remove or modify any of the following 3 statements
        k = 999999999;
        for(int i=0;i<100;i++)
                k = k / 2;

}

/**
 * A method that allows a customer to withdraw money from this account
 * @param amount A double that represents a withdrawal amount
 */
public void withdraw(double amount){
        // Waste some time doing fake computations

        // do not remove or modify any of the following 3 statements

        double k = 999999999;

        for(int i=0;i<100;i++)

            k = k / 2;


            balance = balance - amount;


        // Waste some time doing fake computations

        // do not remove or modify any of the following 3 statements

        k = 999999999;

        for(int i=0;i<100;i++)

            k = k / 2;

    }
```

Task 6: While running the compiled code for **Task 4** and **Task 5**, the results consistently show that the total elapsed time is lower in **Task 4** (method-level synchronization) than in **Task 5** (block-level synchronization). This can be explained by how the Java Virtual Machine (JVM) and the underlying operating system handle synchronization internally.

Using **method-level synchronization** allows the JVM to apply several internal optimizations such as instead of acquiring a monitor, the thread might spin lock which reduces the overhead of acquiring and releasing locks when contention is low.

On the other hand, **synchronized blocks** provide finer-grained control but will introduce additional **locking overhead**. Each entry into a synchronized block requires the JVM to acquire a monitor, and this leads to the thread being parked by the kernel, which introduces overhead as the kernel has to perform a kernel routine to park the thread, and then, when the lock is released, the kernel has to put the threads waiting on that monitor into the ready queue.

Therefore, for **simple scenarios**—where each method performs a small, atomic operation on shared data—method-level synchronization tends to perform better because of its simplicity and JVM-level optimizations. However, in **more complex or fine-grained cases**, synchronized blocks are preferable. For example, if a method performs intensive computations but only a small portion of it actually accesses shared data, synchronizing just that critical section allows multiple threads to execute the CPU-heavy part concurrently without waiting for each other. In contrast, synchronizing the entire method would block all threads from entering, reducing parallelism and overall performance.