1. **Start by compiling and running the program as is. The result at the very end should show a total of 200, but concurrency problems may make it lower or higher, meaning that money is either being lost or created erroneously. You may need to run the program multiple times in order to see incorrect results. (That's the difficulty of concurrent applications: bugs may not show for a long time... until they do!)**

Run 1: works fine.

Run 2: works fine.

Acc1:116 Acc2:84 Total:200

Run 3: works fine.

Acc1:116 Acc2:84 Total:200

Run 4: works fine.

Acc1:84 Acc2:116 Total:200

Run 5: incorrect

Acc1:78 Acc2:151 Total:229

Deadlocks: doesn’t even exit maybe when locks don’t work correctly.

Run 6: incorrect

Acc1:387 Acc2:44 Total:431

2. **The method transferMoney already has commented code that tries to fix the race conditions. Uncomment that code, compile and run the program again. This time, you should run into another problem: the dreadful deadlocks, which, again, may or may not happen. You may need to run the program several times. When a deadlock happens, the program will block, and won't ever exit.**

synchronized (fromAccount) {

synchronized (toAccount) {

Run 1: deadlocks b/c it get stuck and not moving. This is similar to listing 10.1 in chapter 10 of the book, Lock-ordering deadlocks.

A race condition arises in software when a computer program, to operate properly, depends on the sequence or timing of the program's processes or threads.

Race condition in Java is a type of concurrency bug or issue that is introduced in your program because of parallel execution of your program by multiple threads at the same time.

Solution:

To avoid deadlocks, we need to lock the threads in the same order from every thread to avoid deadlocks.

To do so, we find the account with the lowest account number and always lock that one first