Project 2: An Application Employing Synchronized/Cooperating Multiple Threads in Java Using Locks

--A Banking Simulator—

Objective: To develop an application which requires cooperating, synchronized multiple threads of execution.

Description:

* The project will simulate the deposit/withdrawal of money to a bank account
* Deposits and withdrawals will be made by user agents
  + Synchronized threads
* Synchronization is required for 2 reasons
  + Mutual exclusion (updates cannot be lost)
  + Withdrawals cannot occur if the amount of the withdrawal request is greater than the current balance in the account
* Access to the account (THE SHARED OBJECT) must be synchronized
  + This program requires cooperation and communication amongst the various agents (cooperating synchronized threads)
* IN OTHER WORDS
  + This problem is similar to the producer/consumer problem where there is more than one producer and more than one consumer process active simultaneously
* If a withdrawal agent attempts to withdraw an amount greater than the current balance in the account, then it must be blocked itself and wait until a depositing agent has added money to the account before it can try again
  + This requires that the depositing agents signal all waiting withdrawing agents whenever a deposit is completed

1. You should have 5 depositor agents (threads) and ten withdrawal agents (threads) and two auditor agents (threads) **SIMULATNEOUSLY EXECUTING**
   1. Use a FixedThreadPool() and an Executor object to control the threads
2. Assume that deposits are made in amounts ranging from $1 to $500 and withdrawals are made in amounts from $1 to $99 (**WHOLE DOLLARS ONLY**)
   1. Since there are more withdrawal threads than depositor threads, the account balance should constantly decrease over time
   2. Start the simulation with a balance of $0
3. Once a depositor agent (thread) has deposited into the account, put it to sleep for a few milliseconds (randomly generate this number – DON’T USE COMMON SLEEP TIME) or so (depends a little bit on the speed of your system as to how long you will want to sleep the threads – we basically want to ensure a lot more withdrawals than deposits) to allow other agents (threads) to execute
4. For withdrawal agents, things will be a bit different depending on whether you are working on a single or multi-core processor
   1. Single core: once a withdrawal has taken funds, have it yield the processor unit. Since the agent is giving up the processor voluntarily, it will be unlikely to run again (attempt a second withdrawal in a row) before another agent runs.
      1. Note that it does not prevent it from running again. If all other withdrawal agents are blocked and all depositors are sleeping, it will run again. Occasional back-to-back runs of withdrawal agents might occur
   2. Multi-core: once a withdrawal agent has executed, have it sleep for a random time period (a few milliseconds – same as depositor)
      1. Depending on which core a thread is executing, yielding the CPU won’t ensure that the same thread will not run again immediately
      2. While sleeping the thread will also not ensure that it will not run two or more times in succession, it is less likely to do so in the multi-core environment
   3. What we don’t want to happen is a single withdrawal agent gaining the CPU and then executing a long sequence of withdrawal operations
      1. Recall though, that withdrawal agents block if they attempt to withdraw more than the current balance in the account
   4. We don’t want depositor agents monopolizing the CPU either and causing the balance in the account to grow continuously
      1. This would most likely occur when the withdrawal agents are sleeping too long in comparison to average sleep time of depositing agents (Page 11 for pic)
5. Assume all depositor, withdrawal, and auditor agents have the same priority
   1. DO NOT give different priority to depositor, withdrawal, and auditor agents (threads)
   2. The auditor agents will also have normal priority and will simply run less frequently than the depositor and withdrawal threads (i.e. the auditor agents will sleep longer between runs than either depositor or withdrawal agents) See below
6. The output from the program must look reasonably similar to the output shown. The simulation output should show the action of each agent along with the account balance produced by the agent’s transaction and the transaction number
7. Do NOT put the threads into a **counted loop** for your simulation
   1. The *run ( )* method for all threads should be an **INFINITE LOOP**
   2. Just stop the simulation from your IDE after a few seconds
8. Do NOT use the Java synchronized statement
   1. Handle the locking and signaling on your own! No monitors!
9. You **must** utilize a reentrant lock from the *java.util.concurrent.locks* package for implementing your locking protocols
   1. No fairness policy for this application
   2. DO NOT **create your own lock** using a Boolean or any other variable
10. Banks are required to file currency transaction reports (CTR’s) with the federal government for any deposits of $10K or more
    1. Simulate this by flagging any deposit transaction with a deposit greater than $350 and any withdrawal amount greater than $75
    2. **Flag the transaction** in the normal output of the simulation as well as making an entry into a transaction log file (transactions.csv) which will keep track of all flagged transactions independently of the simulation
    3. Each entry in the file will contain transaction details, a timestamp (date and time of which the transaction occurred) and the transaction number
11. Every transaction made by a depositor or withdrawal agent will have a transaction number (**integer initialized to 1**)
    1. This number is printed out in the simulation with each completed transaction
12. The auditor agents (only 2) verify the current balance in the account at random intervals and indicate how many transactions have occurred since the last audit of its type
    1. One is called ***InternalBank*** and the other is ***TreasuryDept***
    2. The agents do not make transactions on the account and do not affect the transaction number sequence
    3. They simply **print the current account balance into the simulation run** and **keep track of the number of transactions that have executed since the last auditor execution** of that type
    4. All depositor and withdrawal agents **MUST WAIT** (**block**) for any auditor agents to complete before continuing execution
       1. Note that auditors should run much less frequently than either depositor or withdrawal agents

**REDIRECT CONSOLE OUTPUT TO AN OUTPUT FILE IN SUBMISSION!**