CONCURRENT PROGRAMMING TASK - DETAILS

Many banking systems allow multiple cardholders access a single account (e.g. a business account). The consequence of this is that is possible for more than one person to attempt to withdraw or deposit money from a single account simultaneously. If the access to the account is implemented naïvely (ignoring the principles of concurrent programming), this can lead to race conditions, and the potential for the following fraud to be successful:

> Multiple cardholders may collude and attempt to carry out a timing attack. In such an attack, multiple cardholders withdraw money from the account simultaneously, with the aim of only one deduction to the account balance being made.

Your task is to write a program (detailed specifications below) that will simulate the operation of an ATM-enabled bank account with multiple linked cards. In other words, there will be several individuals who can access an account, each with his/her own card (). You program should demonstrate the principles of concurrent programming, making it impossible for the above fraud to be successful.

DETAILED SPECIFICATIONS

Your program should implement each access card as a separate thread, and the account as a monitor.

You program should take *two* command line arguments: the first is the number of access cards, and the second should be the starting balance. If the user does not supply these two arguments, **your program should exit and give an error message**. Do not use default values for these variables, or prompt the user for these values once the program has started.

Within each thread, you should use the following code to simulate a series of accesses to the account:

```
for (int i = 0; i < 20; i++) {
	if (Math.random() > 0.5) {
		account.withdraw((int)(Math.random()*10));
	} else {
		account.deposit((int)(Math.random()*10));
	}
	sleep(200);
}
System.out.println("THREAD "+ getId() + " " + localBalance);
```

(This code is the basis for each thread. You are expected to build upon it in you program, but should maintain the same random generation of withdrawals and deposits. If you choose to implement a runnable instead of extending the Thread class, change the line saying `sleep(200);` to `Thread.sleep(200);`.)

Only one card may access the account balance at any time. Furthermore, the account must always have a positive balance; if a thread attempts to withdraw and there are insufficient funds, it must block until sufficient funds become available. Note that this means there is a

possibility of deadlock: if the account balance is 0 (or close to 0) and all cards request a withdrawal at this point, they may all enter the blocking state. You do not have to implement deadlock prevention, avoidance, detection or recovery; you should just accept that this may happen. (It can be avoided by starting with a small number of cards, such as 5, and a sufficiently high account balance, such as 1000.)

Program Execution

Your program should initialise the account (based on the balance specified on the command line), create and start the *n* threads (where *n* is the number of access cards, specified on the command line), keep a record of transactions, and when all threads are finished, print (to standard output) "**Complete**," followed by a statement of the transactions, in the format described below.

In addition, each thread, just before it terminates, should print out its net balance of transactions. (This is indicated by the `System.out.println("THREAD "+ getId() + " " + localBalance);` line of code overleaf.) The balance is the sum of all withdrawals, minus the sum of all deposits. Note that it acceptable for this to be negative; holders may deposit more than they withdraw.

Statement Format

The statement should be in the following format:

```
Transaction   Withdrawal   Deposit  Balance
                                     1000
1(1)          40                     960
2(3)                        60       1020
3(2)                        20       1040
4(4)          90                     950
…             …             …        …
…             …             …        …
```

The number appearing in parentheses after the transaction number shows the ID of the thread responsible for the transaction.

Hint: you may wish to use String.format to achieve neatly formatted output.