IST 411 Exercise s 1&2 – A Bank Account Transaction System – 2 Parts Serial, 5 Parts Concurrent and Threaded

Exercise 1a: A more procedural approach to refresh basic Java skills

1. First, write a class to represent a bank account, Account. You can see by these instructions that this is not a strict entity class, since it mixes the true entity of a bank account with transactions on it.

   Give Account the fields int accountNumber and double accountBalance; a 2-argument constructor Account(int n, double balance) which initialize the fields to the arguments supplied; and the methods int getAccountNumber(), double getAccountBalance(), void makeDeposit(double amount), and int makeWithdrawal(double amount), each of which implements the obvious behavior that its name implies. If a withdrawal brings the account balance below 0, void the transaction and print at the command line "There are insufficient funds to complete this transaction – transaction voided.", and return 1. If the withdrawal is successful, return 0. Write these methods.

2. Now write a class AccountTest with a main method which instantiates an array of 5 accounts with account numbers 1, 2, 3, 4, 5 with initial balances given below – accept these initial balances as command-line arguments and parse them into the correct type. Also give AccountTest a method void makeTransfer(Account fromAccount, Account toAccount, double amount), which first withdraws amount from fromAccount and deposits into toAccount.

3. Then in AccountTest, execute one transaction of teach type (getAccountNumber, getAccountBalance, makeDeposit, makeWithdrawal, and makeTransfer) for each account. Use the following data to implement these transactions in the order indicated:

   Account 1: initial balance 100.00; deposit amount 350.00; withdrawal amount 200.00; transfer toAccount 2 for 100.00
   Account 2: initial balance 300.00; deposit amount 200.00; withdrawal amount 150.00; transfer toAccount 3 for 200.00
   Account 3: initial balance 200.00; deposit amount 100.00; withdrawal amount 150.00; transfer toAccount 4 for 200.00
   Account 4: initial balance 400.00; deposit amount 100.00; withdrawal amount 200.00; transfer toAccount 5 for 300.00
   Account 5: initial balance 150.00; deposit amount 100.00; withdrawal amount 150.00; transfer toAccount 1 for 200.00

   Give main() a return type double [], and return a 5-element array containing the final balances of Account 1-5 in that order.

4. Write a class TestProgram which calls the main method of AccountTest and compares the computed values returned against the correct ones – see a largely completed TestProgram.java in the directory with these instructions.

In Part b, you'll modify this system to be object-oriented, using inheritance, polymorphism, virtual methods, abstract classes and methods, and interfaces. But for now – please critique this approach to handling transactions like this, both good and bad aspects.

Exercise 1b: Object-Oriented Version of the Serial Bank Account Transaction System

In this exercise, write a strictly object-oriented version of the bank account transaction system from Part a. Strictly follow MVC, encapsulate classes as much as possible, and use inheritance and polymorphism when appropriate.

1. Account should be a strict entity class, don't put transactions here. Only put universally-applicable fields (universally applicable means that the field must be present for all sub-classes) plus accessors (getters) and mutators (setters) in Account.

2. Write a Transaction class and subclasses for the different types of transactions (e.g., BalanceInquiry, Withdrawal, Deposit, Transfer). Clearly, Transaction should be abstract, and the behavior of any Transaction should be contained in a method makeTransaction(), which should override Transaction's abstract version.

3. In AccountTest, declare Transactions as a heterogeneous array, and polymorphically instantiate different types of Transaction objects using their respective sub-class types. Use inheritance and polymorphism as much as possible, using virtual method invocation - do not use *instanceof* and casting. Any submission which uses instanceof and/or casting to decide which method to invoke will be returned to you with a request to fix it.

4. Use one or more interfaces to enforce uniform behavior on the classes.

5. Test as in Part a.

Exercise 2a – Non-Thread-Safe Bank Account Transaction System

Part a is to make a basic, threaded, object-oriented, but non-thread-safe bank account transaction system.

1. Modify your Transaction class to extend Thread. Define your Transaction subclasses: BalanceInquiry, Withdrawal, Deposit, Transfer to extend Transaction – thus each of these subclasses also extends Thread. For each of these, override the run() method much as done in the BankTeller class on page 527 of Raposa – for each class, the run() method should print out what it's doing at the command-line :

   System.out.println("Account number: " + account.getAccountNumber() + "Depositing: $" + amount);

   and then call makeTransaction() for its Transaction type. Reminder: make sure that no transaction can drop any Account balance below $0.00.
2. Now first run the same group of tests as in Exercise 1b using AccountTest and TestProgram, but now the system is fully threaded. But you will need to continue with **many** more testing examples until you come up with plenty of thread safety problems. On your first tests, I strongly recommend that you back up to an earlier version of the Java JDK to see the differences in behavior. As you can more predictably generate thread safety problems, move up to later JDK versions. As you move up to later versions, you should still have thread safety problems, albeit perhaps not as many.
3. I can't emphasize enough throughout this exercise that because of the Java thread scheduler, you should not expect deterministic (non-random) behavior in thread execution. This is one of the paramount features of concurrent processing - random-looking behavior. In fact, I believe that due to the fact that the timing of thread execution is governed by both a job-scheduler and a physical clock, that the behavior is, at least to some extent, physically stochastic (i.e., random), as opposed to a process that is pseudo-randomly generated using a pseudo-random number generator.

Exercise 2b - Thread-Safe Bank Account Transaction System

1. For Part b, fix the thread safety problems from your system in Part a. I suggest you start this by getting acquainted with the various thread-synchronization and other methods of the Thread class, and then apply them to fix your thread safety issues.
2. Again, I recommend initially backing up to an earlier version of the Java JDK to make sure that you can effectively test for thread safety problems. The more you 'fix' problems, the more you need to be concerned that you've just lowered the probability of that problem to the point where you can't detect it if and when it does happen. Therefore, the more unlikely a problem is to occur, the more extensive testing you need to do to effectively rule out that problem. So if you start with an earlier JDK, you should be able to test effectively with smaller numbers of tests. Then move to later versions and blast it with lots of tests to confirm your results.

Exercise 2c - Demonstrate Deadlock Behavior in Thread-Safe Bank Account Transaction System

1. For Part c, experiment until you can demonstrate reasonably consistent deadlock behavior using your system from Part b. Again, you may have to experiment with various Thread methods to get the timing right. Thread timing is especially critical to get deadlocks to occur - just think about the prototype deadlock sequence we discussed in class.
2. Based on previous student experience with this type of exercise, you may get frustrated if you start out with JDK 1.7 or even 1.6. So here I think you should definitely roll back to earlier JDK versions. When you're able to get deadlocks to occur reasonably regularly, then move up to later JDK versions. Note that you should focus **only** on transaction type(s) that can generate a deadlock - carefully think through what types of transactions can generate a deadlock.

Exercise 2d - Deadlock-Free Bank Account Transaction System

1. For part d, fix the demonstrated deadlock behavior you created in Part c.
2. Here is where extensive testing is going to be especially critical. Getting deadlocks to occur is tough enough - the probability of a deadlock is not that high in any case. Once you have implemented some 'fixes', you should expect the probability of a deadlock to be very low indeed. So definitely start testing on earlier JDK versions, and do extensive testing in that environment before moving on to later JDK versions. In your testing, again focus **only** on transaction type(s) that can generate a deadlock - again, carefully think through what types of transactions can generate a deadlock..

Exercise 2e - Simulation of Large, Threaded Bank Account Transaction System

The general objective for this part is to subject your thread-safe and deadlock-free Bank Account Transaction system to a lot of transactions, with the objectives to: 1) try to break the system; and 2) get a practical sense of execution performance of different approaches.

The dominant form of transaction should be the short BalanceInquiry, Withdrawal, Deposit, Transfer threads you've done so far. But you should periodically put in one or more long-running transaction(s) that take a long time to execute and try to tie up the system resources. For example, in real life systems, users may run extensive reports, do database updates, conduct database searches, perform extensive analysis of data, or do other tasks that take a long time to run. The objective of this exercise is to simulate this kind of thing using the system you have developed.

First - apply this test to **two different versions** of your Exercise 2d system:

Version 1. The thread-safe, deadlock-free system as you developed it for Part d, but instantiate at least 10 Account objects.

Version 2. Modify the system from Version 1 to force time-ordering of all your transactions. In other words, you should force all transactions to be time-sequentially processed in strict First-In/First-Out order.

Second – in a separate Java program, write a loop to generate random transactions and write them to a test file - a simple text file is appropriate. Your transactions should be of 5 types – first the 4 types you have already been using: BalanceInquiry, Withdrawal, Deposit, and Transfer threads on all 10 Accounts. The 5th transaction type should be of type Batch, which does any operation you like for a random amount of time from 30 seconds to 1 minute.

The Batch transactions must actively run for this amount of time – don't just put the thread to sleep. You can do ANY type of calculation you like – compute successive digits of PI for a random number of seconds, attempt to factorize the product of to very large prime numbers for a random number of seconds, run an infinite while-loop for a random number of seconds, whatever you like.

Put enough transactions in your test file to keep your Version 1 program busy for at least 20 minutes, but in no case use less than 100,000 transactions. I think 1,000,000 transactions would be more realistic, but I don't want this to take hours to run – yes, our time is expensive, both yours and mine.

Specifics: Different transaction types should occur at random times. Pick a mix of transaction types as follows:

1. Randomly choose between BalanceInquiry, Withdrawal, Deposit threads on all 10 Account objects for 89.97% (probability 0.8997) of your transactions. Randomly choose to which account you're applying the transaction, and make amounts (when applicable) uniformly randomly distributed between $20 and $200.
2. Choose Transfer transactions 10% (probability 0.10) of the time – again, randomly choose to which account you're applying the transaction, and make amounts (when applicable) uniformly randomly distributed between $20 and $200.
3. Choose 0.03% (probability 0.0003) of your transactions to be Batch transactions. As stated earlier, each Batch transaction should run a random amount of time between 30 and 60 seconds. If you use the 100,000 transaction number, this should generate 30 Batch transactions with an average running time of 45 seconds each.

Third - in your AccountTest main method, read in the Transactions from your Test file and execute them for both Versions 1 and 2 of your system. In your code,

- Measure how long each transaction takes to run. I suggest the java.lang.System.nanoTime() method to measure elapsed time in nanoseconds (ns) from the start of execution (1 ns = 10^-9 seconds, or 1 Billion ns/second)- http://docs.oracle.com/javase/6/docs/api/java/lang/System.html#nanoTime()
- For each of the 5 transaction types, keep track of the mean transaction execution time and its standard deviation about the mean. (By mean, I mean arithmetic mean, or simple average), and print these , as well as the cumulative running time, at the end.
- For each version of the program, start each Account object's balance at $1000. For each Account object, print the computed starting and ending balance, and also print the theoretical ending balance – that is, the ending balance if all the transactions were run sequentially in a single thread. These 3 sets of ending balances should all be identical for each