Homework #4: Threading

Due: Monday, November 4th 11:59 pm Eastern time

Grade: 15%

Last Modified: Wednesday, 10/22/19

Specification of Requirements

Overview

The purpose of this homework is to experiment with synchronizing access to objects shared by multiple threads. The provided test program will create the threads. You will create Bank and Account objects, and synchronize operations on them performed by the threads.

You will write code to simulate a **Bank**, which manages a set of **Account**s. Each **Account** has a balance. The only transaction processed by this **Bank** is the transfer of funds from one **Account** to another. Multiple threads will run these transfer operations simultaneously, so of course it is possible for the same **Account** to be accessed simultaneously by different threads. A correct implementation will find that the total of all **Account** balances is the same at the beginning and end of the simulation. Incorrect implementations are likely to fail either by having the beginning and ending balances disagree, or with some threads becoming stuck and never completing.

For example, suppose that there are three accounts, each with $100:

Account 1: 100
Account 2: 100
Account 3: 100

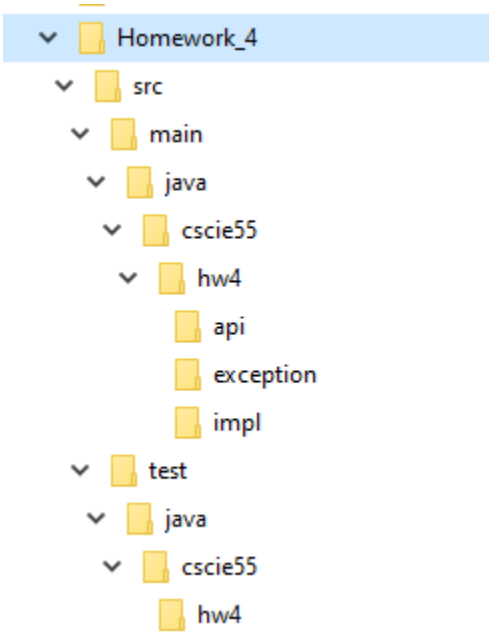
Suppose also that there are two threads, one transferring $25 from **Account** 1 to **Account** 2, and the other transferring $50 from **Account** 3 to **Account**1. Once both of these transactions are done, the accounts should have the following balances:

Account 1: 125
Account 2: 125
Account 3: 50

Note that the sum of balances is $300 before the transactions run, and after both are complete.

Both transactions update Account 1, and if synchronization is not done correctly, then it is possible to lose one of the modifications of **Account** 1's balance. This would leave the final total different from $300. (You should see this happen while working on this homework.)

Your submission will consist of the provided code, along with code that you write. The provided code contains two interfaces that you will implement, some exception classes, and a JUnit test. The code is organized as follows:



Many of the classes are provided. The list below shows what is provided and what is not.

src/

cscie55/

hw4/

api/

Account	Provided
Bank	Provided

exception/

DuplicateAccountException	Provided
InsufficientFundsException	Provided

impl/

AccountImpl	**To be written**
BankImpl	**To be written**

test/

cscie55/

hw4/

ThreadsTest	Provided

Tasks

The classes [some provided and some to be written] are described below.

Account

The provided interface **Account** contains the following:

```
package cscie55.hw4.api

public interface Account

{

   int getId();

   long getBalance();

   void deposit(long amount);

   void withdraw(long amount) throws InsufficientFundsException;

}
```

AccountImpl

Define a class implementing this interface named **AccountImpl** in the ***cscie55.hw4.impl*** package. Balances are of type long, (you can think of the balance as tracking pennies instead of dollars). Your implementation should throw InsufficientFundsException if withdraw attempts to withdraw an amount exceeding the Account's balance.

Bank

The provided interface **Bank** contains the following:

```
package cscie55.cscie55.hw4.api;

public interface Bank

{

  void addAccount(Account account);

 void transferWithoutLocking(int fromId, int toId, long amount) throws
InsufficientFundsException;

 void transferLockingBank(int fromId, int toId, long amount) throws InsufficientFundsException;

 void transferLockingAccounts(int fromId, int toId, long amount) throws
InsufficientFundsException;

 long getTotalBalances();
```

```
int  getNumberOfAccounts();

}
```

BankImpl

Define a class implementing the Bank interface named ***BankImpl*** in the ***cscie55.hw4.impl*** package***.*** Note that there are three methods doing transfers between accounts:

1. ***transferWithoutLocking*** just calls withdraw on one account, and deposit on the other, without doing any synchronization. In other words, your implementation of BankImpl.transferWithoutLocking should not use the synchronized keyword at all. This is completely wrong, but will give you some idea of what happens when synchronization is missing.
2. ***transferLockingBank*** does the transfer while synchronizing on the Bank object. This means that only one thread can do a transfer at any given moment. While this approach does not provide any concurrency, it should be correct.
3. ***transferLockingAccounts*** does the transfer, locking just the two affected Accounts. Your implementation should use the synchronized keyword twice, once on each Account. This should provide for greater concurrency, because threads not touching the two Accounts will not be blocked. Think very carefully about how to lock the two Accounts.

The supplied test code in the src/test/java directory (cscie55.cscie55.hw4.ThreadTest) will call these methods to try different locking strategies.

Define a class named AccountImpl, in package cscie55.cscie55.hw4.impl. This class implements Account, and as such must provide implementations for all methods defined in the Account interface.

addAccount adds an Account to the Bank. No two Accounts should have the same id. If an Account is added with a duplicate id, then *addAccount* must throw *DuplicateAccountException*.

You can assume that the methods *addAccount, totalBalances* and *numberOfAccounts* do not require synchronization.

Things to watch out for

If testPerformance never completes, it is possible that the test threads have become deadlocked. To get a start on this problem, figure out how to obtain and read a stack dump.

The Bank transfer methods throw InsufficientFundsException on an attempt to withdraw more than the balance. If this happens, (and it will almost certainly happen during testPerformance), then the balances of both accounts must be left unmodified. **You need to write code carefully to ensure that nothing has changed when InsufficientFundsException is thrown.** Another approach, which is more complicated (and therefore inferior) discovers the need to throw InsufficientFundsException after one or both balances have been updated, in which case you need to ensure that the balances are restored to their previous values.

How do you implement Bank.addAccount(Account account) in such a way that it throws the DuplicateAccountException if an attempt is made to add an Account with the same id? If your Bank has

a Collection of Accounts, you might ask if that Collection contains the Account in question. The Java Collection classes all offer a method 'contains()'. The question you need to answer is how does that method determine a match among the objects in the Collection. [Hint: consider the equals() method.]

Strongly recommended: Comment out *testPerformance* until the other tests are running successfully. [Simply comment out the @Test annotation. That will cause the JUnit runner to pass over the method.] Otherwise, you may waste a lot of time waiting for *testPerformance* to run when far more basic tests are failing.

Testing

You should not write a main() method. Instead, include cscie55.hw4.ThreadTester which contains JUnit tests. You can run the tests from your IDE, or from the command line as follows:

```
mvn test
```

Most of the tests within ThreadTest are single-threaded, and ensure that AccountImpl and BankImpl handle their inputs correctly.

testPerformance is quite different. It does not test correctness at all, i.e., it does not contain any JUnit assertions. This test has two purposes. First, unlike the other tests, it starts a number of threads, each submitting transfer requests to the Bank. If the Bank's getTotalBalances don't agree before and after the transfers, output will indicate this, although the test will not fail, (because we want to continue and gather output from the remaining scenarios.) Second, the test does timings, reporting the amount of time to do 5,000,000 transfers using 1, 2, 5, 10, and 20 threads, for each of the three locking strategies.

What to Submit

The pom.xml file included in your hw package contains the build targets to generate 3 jars, as we did for HW 3, ***excepting*** the JavaDoc jar. For this and future assignments, JavaDoc is not required. You should still comment your code well, because the Teaching Fellows will use it to understand what you have done. And you will soon forget why you have done what you have done, and it will thus help you too.

Revise the <artifactId> element and replace 'threading' with your lastName_firstName_hw4.
Test that they run from the command line.
Zip up the following:

- -sources.jar
- Classes jar [the regular generated jar]
- -tests.jar
- README

Submit via Canvas