

CSC-226 Name: **Kyle Evangelisto**

Programming Languages



Chapter #13 -- Concurrency Lab

Download the ***ConcurrencyEg*** zipped project from Piazza.

We will use this sample program to examine some of the pitfalls when programs run concurrent processes (using Java).

The program will “deposit” and “withdraw” money from the same bank account object. It performs the same # (and same amount) of deposits and withdraws, so the ‘final’ balance should be 0.

1. Take a few minutes to examine the program – get a basic understanding of its intent and how it works. Then run the program and examine the output three times.

Output files stored in with zip contents.

1. Was the final balance always the same? No. Was it zero? No.

Final Balances:

. , new balance is 500.0

. , new balance is -600.0

. , new balance is 800.0

1. Record the last few lines of output (cut and paste here).

.Test1:

Depositing 100.0, new balance is 500.0

, new balance is 300.0

, new balance is 400.0

, new balance is 500.0

.Test 2:

Depositing 100.0Withdrawing 100.0Depositing 100.0, new balance is -800.0

, new balance is -600.0

, new balance is -600.0

Withdrawing 100.0Depositing 100.0, new balance is -700.0

Depositing 100.0, new balance is -500.0

, new balance is -600.0

.Test 3:

Withdrawing 100.0Withdrawing 100.0Depositing 100.0Depositing 100.0Depositing 100.0, new balance is 1100.0

, new balance is 1100.0

, new balance is 1100.0

, new balance is 900.0

, new balance is 900.0

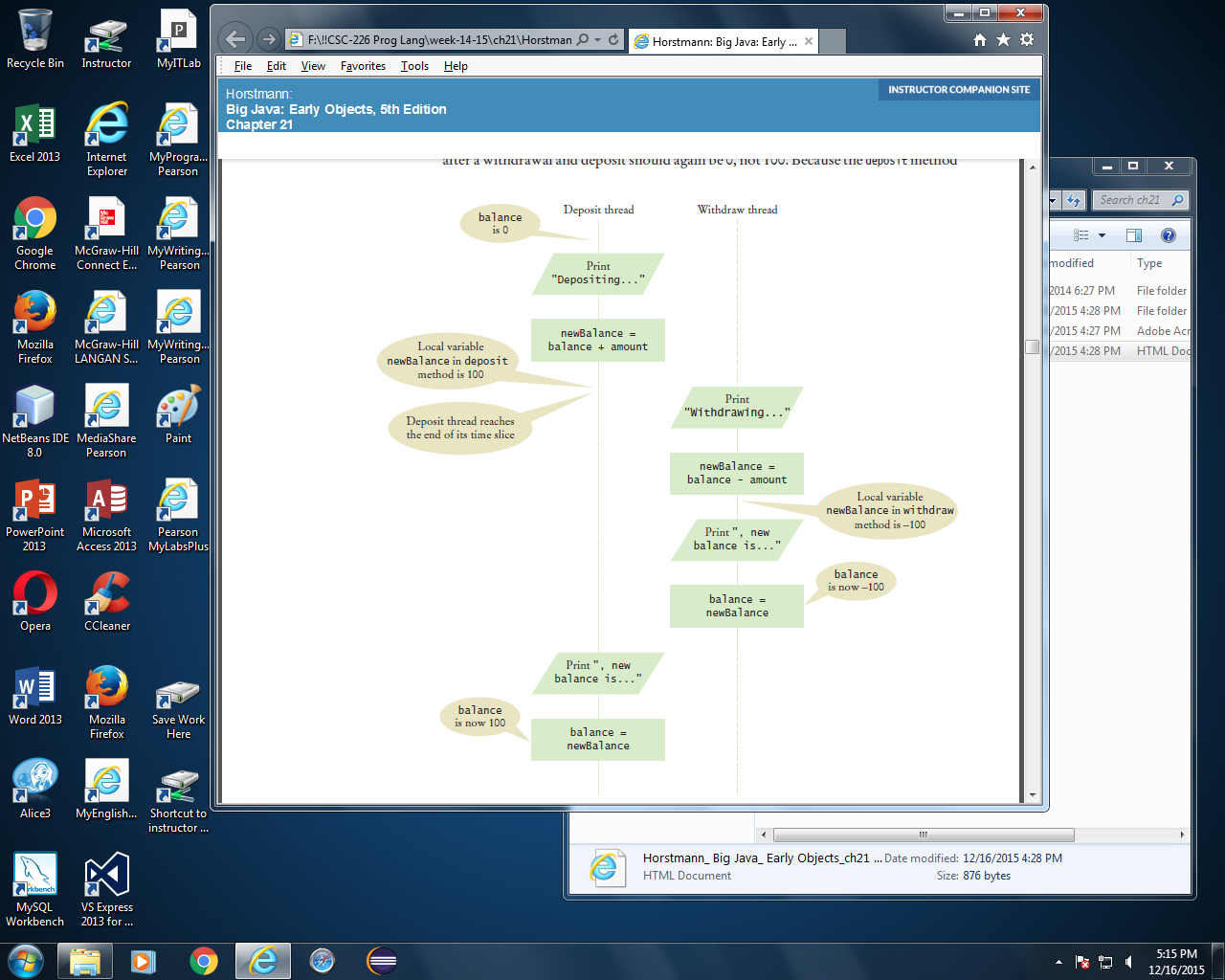
Withdrawing 100.0, new balance is 800.0

1. Since the withdrawal thread and the deposit thread are sharing the same account, this is an example of cooperation synchronization.
2. Explain how Java (VM) manages its threads? Hint: lightweight tasks/time slicing

Inside the JVM, each thread has its own stack, which contains data that no other thread can access, including local variables, parameters, and return values of each method the thread has invoked. The data on the stack is limited to primitive types.

This is an example of a ***race condition.*** Although each thread has its own local variables, they share access to and mutate the balance instance variable.

Here is a picture of what is happening to cause the problem



(Figure 21.1 from Big Java, by Cay Horstmann)

Perhaps we can “speed up” each deposit/withdraw process. Alter the code:

* so that it doesn’t use a local variable inside the two methods and
* mutates the balance FIRST followed by one System.out.println() method to display the info.

Cut and paste your new code here….

public BankAccount()

{

balance = 0;

}

/\*\*

Deposits money into the bank account.

//@param amount the amount to deposit

\*/

public void deposit(double amount) {

balance += amount;

System.out.print("Depositing " + amount + " , new balance is " + balance + "\n");

}

/\*\*

Withdraws money from the bank account.

//@param amount the amount to withdraw

\*/

public void withdraw(double amount) {

balance -= amount;

System.out.print("Withdrawing " + amount + ", new balance is " + balance + "\n");

}

/\*\*

Gets the current balance of the bank account.

@return the current balance

\*/

public double getBalance()

{

return balance;

}

Re-run the program a few times. Did this fix the problem? Cut/paste (some) output here

Depositing 100.0 , new balance is 0.0

Withdrawing 100.0, new balance is -100.0

Withdrawing 100.0, new balance is -300.0

Depositing 100.0 , new balance is 0.0

Withdrawing 100.0, new balance is 100.0

Regardless, we still have a problem – even if this now seems to work, we still have the same inherent problem (it just may occur less frequently). It is still possible for the deposit (or withdrawal) time slice to run out of time before it completes the balance assignment and therefore puts a wrong value into balance…. :0

Part 2. We need to “synchronize” these competing methods by using Java *Lock objects* (essentially an implementation of a semaphore or guard)

Add the following to the Account class:

1. Add these two import statements

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;

1. Add a class instance variable for the Lock

private Lock balanceChangeLock;

1. Initialize the Lock object inside the constructor as follows:

balanceChangeLock = new ReentrantLock();

1. Add the “guard” to the **deposit and withdraw** methods, e.g.

public void deposit(double amount)

{

**balanceChangeLock.lock();**

balance += amount;

…

**balanceChangeLock.unlock();**

}

When a thread calls the Account’s balanceChangeLock object’s lock method, it owns the lock until it invokes the unlock method. The Java thread scheduler will put any other thread that attempts to call the *balanceChangeLock.lock()* into a (queued) wait state and periodically reactivates it and lets it try again to attempt to lock the shared resource.

1. Run the code 3-4 times. Did this fix the problem?

Yes.

1. Remove the unlock at the end of the withdraw method….explain what happens? Show sample output.

This is my output when I remove the unlock

The lock and unlock statements used in the deposit and withdraw methods keep control of what money is going in and out of the bank account. When deposit is called it is locked and unlocked. When withdrawal is called the lock is never unlocked. And there are only withdrawals after the first deposit.

Depositing 100.0 , new balance is 100.0

Withdrawing 100.0, new balance is 0.0

Withdrawing 100.0, new balance is -100.0

Withdrawing 100.0, new balance is -200.0

Withdrawing 100.0, new balance is -300.0

Withdrawing 100.0, new balance is -400.0

Withdrawing 100.0, new balance is -500.0

Withdrawing 100.0, new balance is -600.0

Withdrawing 100.0, new balance is -700.0

Withdrawing 100.0, new balance is -800.0

Withdrawing 100.0, new balance is -900.0

Let’s bullet-proof this a bit more. If one of the methods threw an exception before the unlock, it would not release its lock on the shared resource. So to make this as safe as possible, we should add try and finally blocks:

public void deposit(double amount)

{

**balanceChangeLock.lock();**

**try{**

balance += amount;

…

**} finally{**

**balanceChangeLock.unlock();**

**}**

}

Now, even if an exception is thrown the shared resource will be freed up.

Fix your code to include this change to deposit () and withdraw() methods of class Account.

Submit this paper with typed answers/pasted code

Submit a digital copy of your completed concurrency lab as a zipped NetBeans project.