**44-542 Object-Oriented Programming**

**Threads**

1. Create a project named **542BankAccount**. Then create the class **BankAccount** in package **my.accounts**. It has one private instance variable.

**private double balance;**

* 1. Constructor: There is one no-arg constructor that sets the account balance to 0.
  2. **getBalance**: There is a getter method for **balance**.
  3. **setBalance**: There is a setter method for **balance**.
  4. **deposit**: The code for the **deposit** method is shown below. Use *exactly* the code shown here. This code could be reduced to one line by just adding amount directly to balance, without introducing the intermediate variable **newBalance**. *Do not do this.* Leave the code exactly as shown below.

**public void deposit(double amount)**

**{**

**double newBalance = getBalance() + amount;**

**balance = newBalance;**

**}**

* 1. **withdraw**: The **withdraw** method is similar to the deposit method, except you subtract the amount instead of adding. Write the code for this method, again using the intermediate variable **newBalance**, rather than writing only a single line of code. Note that negative balances may occur and are not considered an error.

1. Create a new class in package **my.accounts** named **DepositRunnable**. This class implements **Runnable** and has the following constants and instance variables:

**private static final int DELAY = 10;**

**private BankAccount account;**

**private double amount;**

**private int count;**

* 1. Constructor: The constructor has three parameters for initializing the private instance variables listed above, in the same order as listed.
  2. **run**: This method implements the **run** method in the **Runnable** interface. It deposits the amount stored in the **amount** instance variable and then sleeps for the number of milliseconds stored in **DELAY**. These two actions are in a **for** loop that runs **count** times. Note that the **sleep** statement must be inside a **try-catch** block.

1. Create a new class in package **my.accounts** named **WithdrawRunnable**. This class implements **Runnable** and is identical to the class **DepositRunnable** except that the **run** method withdraws instead of depositing.
2. Create a new class in package **my.accounts** named **BankAccountThreadTester**. This class has a main method. The header of the method should contain the **throws InterruptedException** clause. Method **main** has the following local variables and constants declared:

**BankAccount account = new BankAccount();**

**final double AMOUNT = 100;**

**final int REPETITIONS = 1000;**

Following the variable declarations, the program does the following:

* 1. A new **DepositRunnable** object is created, using **account**, **AMOUNT**, and **REPETITIONS** as the arguments.
  2. A new **WithdrawRunnable** object is created, using **account**, **AMOUNT**, and **REPETITIONS** as the arguments.
  3. Two new **Thread** objects are created, using the **DepositRunnable** and **WithdrawRunnable** objects created in the previous steps.
  4. The two threads created in the previous step are started.
  5. Before testing this class, we need to add some print messages to the methods in the **BankAccount** class. Ordinarily print statements should not be included in a method, since we have no way of knowing where a user might want output directed. However, we can add print statements for testing, which we will do now.
  6. Change the code for the **deposit** method in the **BankAccount** class to the following:

**System.out.println("Depositing: " + amount);**

**double newBalance = getBalance() + amount;**

**System.out.println("New balance is " + newBalance);**

**balance = newBalance;**

* 1. Make similar changes to the **withdraw** method in the **BankAccount** class.
  2. Run the **BankAccountThreadTester** several times. Each thread runs for 1000 iterations, so there will be many lines of output. The output will vary from one run to another, but you will notice that the deposits and withdrawals are interleaved, that the balances can vary significantly, and that the final balance usually is not 0. It *should* be zero, because the initial balance is 0, and we are depositing the same amount that we are withdrawing, but there are concurrency problems which cause errors, resulting in an incorrect final balance.

1. We will now add a lock to ensure that a transaction (deposit or withdrawal) is completed before another transaction can take place. Alter the **BankAccount** class to include two additional private instance variables:

**private Lock balanceChangeLock;**

**private Condition sufficientFundsCondition;**

1. Change the constructor so that in addition to setting the account balance to 0, it also initializes the two instance variables that you just added:

**balanceChangeLock = new ReentrantLock();**

**sufficientFundsCondition = balanceChangeLock.newCondition();**

1. Alter the code for the **deposit** method so it looks like this.

**balanceChangeLock.lock();**

**try**

**{**

**System.out.println("Depositing: " + amount);**

**double newBalance = getBalance() + amount;**

**System.out.println("New balance is " + newBalance);**

**balance = newBalance;**

**}**

**finally**

**{**

**balanceChangeLock.unlock();**

**}**

1. Change the code for the **withdraw** method in a similar manner.
2. Run **BankAccountThreadTester** again. This time, the final balance should be 0. The lock has prevented the **withdraw** and **deposit** methods from interfering with one another.
3. There is still one more problem to deal with, because negative balances do occur. We will alter the program once again, to ensure that a withdrawal cannot be made unless there are sufficient funds present.
4. Alter the **withdraw** method to include the following statement as the *first* statement inside the **try** block. With this statement, we are forcing the **withdraw** method to wait until there are sufficient funds available Note that you will have to add the **throws InterruptedException** clause to the method header.

**while(balance < amount)**

**{**

**sufficientFundsCondition.await();**

**}**

1. Because you added the **throws InterruptedException** clause to the method header in the previous step, check that the statement **account.withdraw(amount);** in the **run** method of **WithdrawRunnable** is in the same **try-catch** block as the **sleep** statement.
2. Alter the **deposit** method to include the following statement as the *last* statement inside the **try** block. This statement will cause the deposit method to send a signal each time a deposit is completed, and the lock is released. Other methods that are waiting for this signal, such as the **withdraw** method, can them compete for the lock once again.

**sufficientFundsCondition.signalAll();**

1. Run **BankAccountThreadTester** again. The final balance should still be 0. However, there should no longer be negative balances.
2. For the last exercise we will create a new project that does the same thing as the project we just completed. However, we will use synchronized methods. Synchronized methods are simpler to use than locks and conditions, but are more limited in capabilities. Locks and conditions were added to JDK 1.5, but it is still alright to use synchronized methods when appropriate. For simple applications, requiring only one lock and one condition, it is probably the preferred method because of its simplicity.
3. Create a project named **542BankAccountSynchronized**. Then create the class **BankAccount** in package **my.accounts**. This class will look like the original **BankAccount** class, before we added the lock and condition variables:

**package my.accounts;**

**public class BankAccount**

**{**

**private double balance;**

**public BankAccount()**

**{**

**setBalance(0);**

**}**

**public void deposit(double amount)**

**{**

**System.out.println("Depositing: " + amount);**

**double newBalance = getBalance() + amount;**

**System.out.println("New balance is " + newBalance);**

**balance = newBalance;**

**}**

**public void withdraw(double amount) throws**

**InterruptedException**

**{**

**System.out.println("Withdrawing: " + amount);**

**double newBalance = getBalance() - amount;**

**System.out.println("New balance is " + newBalance);**

**balance = newBalance;**

**}**

**public double getBalance()**

**{**

**return balance;**

**}**

**public void setBalance(double balance)**

**{**

**this.balance = balance;**

**}**

**}**

1. Add the classes **DepositRunnable**, **WithdrawRunnable**, and **BankAccountThreadTester**. These classes will be identical to the classes that you created for the previous project.
2. Test your project with **BankAccountThreadTester**. It should run correctly, but may have negative balances and a final balance different from 0.
3. Add **synchronized** to the method header of both the **deposit** and the **withdraw** methods. (It goes right after the word **public**.)
4. Add code at the beginning of the **withdraw** method to have it wait for sufficient funds before withdrawing.

**while(balance < amount)**

**{**

**wait();**

**}**

1. Add the statement **notifyAll();** to the end of the **deposit** method.
2. Run **BankAccountThreadTester** again. It should work correctly. No negative balances should occur, and the final balance should be 0.