Introduction

This lab will give you practice working with Java threads and concurrency, which are covered in the handout. You will take a set of classes that simulate the use of a bank account by multiple users and modify it to give the simulation realistic, multi-threaded behavior.

Note: This lab is based on material by Gary Shute for his CS2511 class at University of Minnesota.

Basic Requirements

The program allows control of the following aspects of a bank account:

- **Starting balance**.
- **Number of users** having access to the account.
- **Number of transactions** allowed per user.
- **Transaction amount limit**.

We imagine a scenario in which a number of siblings are given access to a bank account by a parent. For each simulation run:

- A bank account’s balance is set to the indicated starting amount.
- The indicated number of bank account users — siblings — are created.
- For each sibling, a list of the indicated number of transactions, randomly created as deposits or withdrawals, of random amounts in the range governed by the indicated limit, is created.
- Each sibling carries out its list of transactions, with the constraint that the balance cannot go below zero.
- The result of each transaction is displayed in a transaction log.

Current Behavior

In the version of the program you are given, the siblings’ transaction lists are processed in series — a sibling carries out all of the transactions on its list before another sibling can carry out its transactions. This creates two deficiencies:

- The simulation is not realistic — the siblings should carry out their transactions in parallel
- If a sibling happens to attempt a withdrawal causing a negative balance, the simulation halts, and remaining siblings are denied their transactions

In the Current Program:

- A high starting balance will ensure that all transactions are carried out.
- A lower starting balance increases the chance of an early exit.
Desired Behavior

You will correct these deficiencies by making each sibling’s transaction processing occur in a separate thread, making sure to avoid potential race conditions resulting in an incorrect balance.

Additionally, to avoid the **deadlock** situation that occurs when all siblings attempt illegal transactions, you will add a parent thread that “rescues” the siblings from deadlock by making deposits when appropriate. In the Desired Program:

- A high starting balance will make a parent unnecessary, but the siblings’ transactions will be carried out in parallel, with it possibly happening that one sibling ”rescues” another from an illegal withdrawal.
- A lower starting balance increases the chance of a parent rescue.
- The parent rescues with (possibly repeated) deposits of $100.
- The parent is only involved when the siblings are deadlocked and all siblings have completed their transactions, in which case the parent thread terminates by attempting to withdraw a zero amount.
- No situation results in any sibling’s not carrying out all of its transactions.

How to proceed

Create a Lab09 Java Project in Eclipse and within it create a package named threads. Copy the contents of /common/cs/cs062/labs/lab09 into the src directory in your new Eclipse project and select File/Refresh.

The lab will proceed in three steps:

- Thread the program
- Synchronize the threads
- Rescue the threads from deadlocks

Step 1: Add Threads

The BankAccountUser class has code in its run method that we want to run in a separate thread. The BankAccountControl class constructs a RUN button whose action listener loops through the BankAccountUser objects and calls their run methods in succession. In this step, you will simply thread the program (without synchronizing) and observe the results.

Here is the simplest way to make the BankAccountUser class run methods execute in separate threads:

- Declare BankAccountUser to extend the Thread class.
- Since the Thread class already has a getName method, remove this method from BankAccountUser and, in the constructor, replace the statement this.name = name; to super(name);
- At the bottom of BankAccountUser’s run method’s while loop, just before repeating, sleep for, say, 100 milliseconds — this will require handling a possible InterruptedException.
- In BankAccountControl, find where the button listener loops through the users and change the call run() to start().

Now try running the simulation.

- For one sibling, the behavior will be fine unless an illegal withdrawal is attempted, in which case an exception is thrown and the simulation halts.
- For multiple siblings, one or more of the following scenarios will likely occur:
  - Output will be corrupted as messages are not able to complete.
  - Runtime exceptions occur due to illegal withdrawals.
Assertion errors occur due to race conditions as deposits and withdrawals are interrupted.

To solve these problems, the threads must be synchronized.

---

**Step 2: Synchronize the Threads**

In this step, you will eliminate the runtime exceptions and assertion errors by synchronizing the threads using the Object class’s `wait()` and `notifyAll()` methods, and observe the results. Runtime exceptions and assertion errors originate in the withdraw and deposit methods of the BankAccount class, so:

- For each of these methods add the `synchronized` modifier to the method’s declaration.
- In `withdraw`, instead of throwing an exception when an illegal withdrawal is attempted, make a call to `wait()` while the withdrawal amount is greater than the balance.
- Declare the `withdraw` method to throw `InterruptedException` (which is handled by the BankAccountUser’s `run` method).
- At the end of the `deposit` method, just before the assertion, make a call to `notifyAll()`.

Results: Assertion errors from race conditions and illegal withdrawals are avoided, but deadlocks can result, with one or more siblings not finishing their transactions. Here is an example of a run with an initial balance of 100 dollars, 3 siblings, and a 100 dollar amount limit. In this run:

- Sibling 1 tries to make a withdrawal ($97) greater than the balance ($59), so it waits.
- Sibling 2 makes a deposit ($57), lifting the balance sufficiently, so sibling 1’s withdrawal completes.
- Siblings 3 and 2 both try to withdraw ($95 and $99) more than the balance, so they wait.
- Sibling 1 makes a deposit ($8), but not enough for siblings 2 and 3.
- Sibling 1 tries to withdraw more than the balance ($78), so deadlock results.

---

**Step 3: Rescue Sibling Threads from Deadlocks**

In this step, you will add a thread whose sole purpose is to rescue the sibling threads from deadlock. This thread, analogous to the siblings’ parent, will be executed by the `run` method of a new BankAccountRescuer class, which will extend BankAccountUser.

The BankAccountRescuer thread will:

- Detect when all siblings are waiting to make a withdrawal
- Deposit $100 when all siblings are waiting
- Deposit $0 and terminate when all siblings are finished (the dummy deposit is so the BankAccount object can log a message when the parent thread has finished)

Note how the “finished” status of a BankAccountUser object is managed:

- BankAccountUser has boolean fields `oneMore` and `finished` with accompanying getters and setters.
- The BankAccountUser constructor initializes `oneMore` and `finished` to `false`.
- The BankAccountUser `run` method sets `oneMore` to true when it is about to make its last transaction.
- The BankAccount class’s `deposit` and `withdraw` methods check the `oneMore` status of the user, setting the user’s `finished` field to `true` after the last transaction is complete.

You can manage the waiting status of a user in a similar way:
• Add a boolean `waiting` field with setter and getter to `BankAccountUser`.

• Initialize `waiting` to `false`.

• In the `BankAccount` class’s `withdraw` method:
  – When the user tries to withdraw more than the balance, the user’s `waiting` field is set to `true` before entering the `wait()` loop.
  – When the loop is finished, the `waiting` field is set to `false`.

Now you are ready to create a new class called `BankAccountRescuer` that extends `BankAccountUser`.

• Create a `BankAccountRescuer` constructor that:
  – Takes a name, a `BankAccount`, and an array of `BankAccountUsers` as parameters.
  – Appropriately calls `super` (can send a `null` list of transactions) and stores the user array.

• Suggestion: write private `allFinished()` and `allWaiting()` methods that loop through the user array and return the appropriate boolean value.

• In order for the `BankAccount` object to be available to `BankAccountRescuer`, you will need to use the `getAccount` method of `BankAccountUser`.

• Override the `run` method to loop indefinitely until all users (siblings) have finished:
  – Inside the loop, check if all non-finished users are waiting, and deposit $100 if they are.
  – Like any thread loop, it should occasionally sleep.

• When all siblings have finished their transactions, the parent thread should terminate, after:
  – Setting its `oneMore` status to `true`.
  – Depositing $0 in the account.

Your last step is to modify the `BankAccountControl` class which creates and starts all the sibling threads. To add the parent thread:

• Just after the loop that creates the `BankAccountUser` array of siblings, create a new `BankAccountRescuer` object with name "Parent”.

• Just after the loop that starts each of the sibling threads, start the parent thread.

• In the `allFinished` method, also check that the parent is finished (so the parent’s `run` method can exit).

The program should now display the desired behavior.

---

**What to Hand In**

Export your Eclipse code as usual and submit `answers.txt` along with your `bin/` and `src/` directories as usual. Don’t forget to put both your name and your partner’s name (if any) in the `.json` file.