Concordia University
Computer Science & Software Engineering
Faculty of Engineering and Computer Science

COMP 346 – Fall 2017
Programming Assignment 1

Due Date **By 11:59pm Friday October 13, 2017**
Format Programming assignments must be typed and submitted online to Moodle system. **Scanned Hand-written assignments will be discarded**.
Late Submission: none accepted
Purpose: The purpose of this assignment is to give you the opportunity to implement some concepts of operating systems with Java. You will simply use Java while grasping concurrency and atomicity concepts.

CEAB/CIPS Attributes: Design/Problem analysis/Communication Skills

Question # 1

Tools
You will use Java 1.8 or later for this assignment. You can work on the assignment on any machine you wish (i.e. personal laptop) as long as you use Java 1.8 or a later version, and the assignment works in the lab as expected. You can also use any source code editing & compiling tools, such as TextPad, vim, Emacs, Eclipse, JBuilder, NetBeans, etc. All these tools have easy configurable convenient syntax highlighting. You can use whichever you prefer but you need to make sure that your programs run as expected at the Concordia lab before you submit them.

Setting Up Your Environment
To work in the labs you will need to set up your environment first, which will be explained in the first tutorial. If you require help, don't hesitate to ask your lab instructor for it!

Source Code
There are four files provided with the assignment. You will need these files to work on the assignment. A soft copy of the needed code is available for download from moodle.

File Checklist:
- Account.java
- AccountManagder.java
- Depositor.java
- Withdrawer.java

Background
To begin, realize that this assignment is "a bit" artificial in a sense, and should only be considered for learning purposes. In this assignment, we deal with concurrent execution of multiple threads operating on a shared data structure (Accounts). You will be learning more of Java along the way.

An array of accounts holds 10 account information (imagine like OS resources of the same

type). The account array has a valid state once it is initialized. We will employ 10 threads for deposit and 10 threads for withdrawal; each thread is bound to a specific account. There will be one depositor thread and one withdrawer thread that are bound to one specific account. The depositor is responsible for depositing X amount to the account and the withdrawer is responsible for withdrawing the Y amount from the same account (Y=X). As a result of running 20 threads the final accounts' balance/state should be the same as initial accounts' balance/state. The main goal is to maintain the accounts' state valid regardless of the number of the concurrent threads accessing them, and regardless of their order.

E.g.: of consistent run

```
Print initial          account balances
Account: 1234          Name: Mike          Balance:    1000.0
Account: 2345          Name: Adam          Balance:    2000.0
Account: 3456          Name: Linda         Balance:    3000.0
Account: 4567          Name: John          Balance:    4000.0
Account: 5678          Name: Rami          Balance:    5000.0
Account: 6789          Name: Lee           Balance:    6000.0
Account: 7890          Name: Tom           Balance:    7000.0
Account: 8901          Name: Lisa          Balance:    8000.0
Account: 9012          Name: Sam           Balance:    9000.0
Account: 4321          Name: Ted           Balance:    10000.0

Depositor and Withdrawal threads have been created
Print final account balances after all the child thread terminated...

Account:    1234       Name:  Mike         Balance:    1000.0
Account:    2345       Name:  Adam         Balance:    2000.0
Account:    3456       Name:  Linda        Balance:    3000.0
Account:    4567       Name:  John         Balance:    4000.0
Account:    5678       Name:  Rami         Balance:    5000.0
Account:    6789       Name:  Lee          Balance:    6000.0
Account:    7890       Name:  Tom          Balance:    7000.0
Account:    8901       Name:  Lisa         Balance:    8000.0
Account:    9012       Name:  Sam          Balance:    9000.0
Account:    4321       Name:  Ted          Balance:    10000.0
Elapsed time in milliseconds 13900 Elapsed
time in seconds is 13.9
```

E.g.: of non-consistent run

```
Print initial          account balances
Account: 1234          Name: Mike          Balance:    1000.0
Account: 2345          Name: Adam          Balance:    2000.0
Account: 3456          Name: Linda         Balance:    3000.0
Account: 4567          Name: John          Balance:    4000.0
Account: 5678          Name: Rami          Balance:    5000.0
Account: 6789          Name: Lee           Balance:    6000.0
Account: 7890          Name: Tom           Balance:    7000.0
Account: 8901          Name: Lisa          Balance:    8000.0
Account: 9012          Name: Sam           Balance:    9000.0
Account: 4321          Name: Ted           Balance:    10000.0

Depositor and Withdrawal threads have been created
Print final account balances after all the child thread terminated...

Account:    1234       Name:  Mike         Balance:    -9180.0
Account:    2345       Name:  Adam         Balance:    2000.0
Account:    3456       Name:  Linda        Balance:    -9.953971E7
Account:    4567       Name:  John         Balance:    1.00004E8
Account:    5678       Name:  Rami         Balance:    -3.682011E7
Account:    6789       Name:  Lee          Balance:    6000.0
```

```
Account:    7890    Name:  Tom    Balance:    -6.951315E7
Account:    8901    Name:  Lisa   Balance:    -9.087615E7
Account:    9012    Name:  Sam    Balance:    1.00009E8
Account:    4321    Name:  Ted    Balance:    -9.631527E7
Elapsed time in milliseconds 139 Elapsed
time in seconds is 0.139
```

There are 20 threads accessing the accounts concurrently. One depositor threads deposits into account # 1234 the amount 10 CAD * 10000000 iterations. On the other side, one withdrawer thread withdraws from account # 1234 the same amount 100 CAD * 10000000 iterations. The rest 9 depositor threads and 9 withdrawer threads perform the same operations over the other 9 accounts.

Tasks

Task 1: Atomicity Bug Hunt

At the beginning assume that the Y and X are equal. Compile and run the Java app given to you as it is. Explain why the main requirement above (i.e. consistent state of the account array) is not met. What atomicity problem does it pose? Find the bug that causes it. In no more than three sentences, explain what went wrong. Is there any way that two operations can write at the same time? What is happened if you put X=10 CAD and Y=20 CAD. Modify the code, run it and see the results. Explain the reason no more than three sentences. What is the equivalent situation in operating system?

Task 2: Starting Order

Explain, in about one sentence, what determines the start order of the threads. Also, very briefly, explain the lifetime of a thread: its creation, execution, and termination. Experiment with the start order of any of the threads. Is the consistency of the accounts preserved?

Task 3: Method level synchronization

- Create a package and name it task3A and copy the provided java files into that new package. Use synchronized technology (method level synchronization) in order to introduce a solution to the problem at hand.
- We put the X=10, and Y=20. We assume that the account should not go negative at any time (we ignore the role of consistency on the first section). On the other hand, you cannot expend more than your recourses. Having said, create a package and name it task3B and copy the provided java files into that new package. Use synchronized technology (method level synchronization) in order to introduce a solution to the problem at hand.

Task 4: Block level synchronization

- Create a package and name it task4A then copy the java files from task 3A into that new package. Use synchronized technology (block level synchronization) in order to improve the solution to the problem at hand.
- We put the X=10, and Y=20. We assume that the account should not go negative. Having this assumption, Create a package and name it task4B

then copy the java files from task 3B into that new package. Use synchronized technology (block level synchronization) in order to improve the solution to the problem at hand.

Task 5: synchronized block vs synchronized method

Considering the results of task 3A vs task 4A, what is the advantage of synchronized block over synchronized method?

Deliverables

IMPORTANT: You are allowed to work on a team of 2 students at most (including yourself!). Any teams of 3 or more students will result in 0 marks for all team members. If your work on a team, ONLY one copy of the assignment is to be submitted for both members. You must make sure that you upload the assignment to the correct directory of **Lab Assignment 1** using moodle. Assignments uploaded to the wrong directory will be discarded and no resubmission will be allowed.

Naming convention for uploaded file: Create one zip file, containing all needed files for your assignment using the following naming convention:
The zip file should be called *a#_studentID*, where # is the number of the assignment *studentID* is your student ID(s) number. For example, for the first assignment, student 1234567 would submit

a zip file named *a1_1234567.zip.* If you work on a team and your IDs are 1234567 and 4567890, you would submit a zip file named *a1_1234567_4567890.zip.*

Submit your assignment electronically via moodle. **Please see course outline for submission rules and format, as well as for the required demo of the assignment.** A working copy of the code and a sample output should be submitted for the tasks that require them. A text file with answers to tasks 1 to 3 should be provided. Put it all in a file layout as explained below, archive it with any archiving and compressing utility, such as WinZip, WinRAR, tar, gzip, bzip2, or others. **You must keep a record of your submission confirmation from moodle.** This is your proof of submission, which you may need should a submission problem arises.

Possible file layout:

```
pa1.txt                 -- Tasks 1, 2 and 5 theory components (can be .doc as
well) task3/
        *.java          -- Fixed Java code
        before.out      -- Buggy output
        after.out       -- Output after the code has been fixed

task4
/       *.java          -- Fixed Java code
        before.out      -- Output after the code has been fixed in task 3
        after.out       -- Output after the code has been improved for
        exceptions
```

Zip all files into a file called pa1.zip, and submit that file electronically to EAS.

Grading Scheme

Grading Scheme:

```
T#      MX      MK
-----------------------
1         /1
2         /1
3         /3
4         /3
5         /1
General /1
-----------------------
Total:
```

(T# - task number, MX - max (out of), MK - your mark)

References

API: http://java.sun.com/j2se/1.3/docs/api/