Issue date : September 20 2016 **Due date** : October 4 2016
Instructor : Kerly Titus

Objectives

This is an introductory assignment to operating systems and Java. You will simply use Java while grasping concurrency and atomicity concepts.

Tools

You will use Java 1.5 or later for this assignment. You can work on the assignment on any machine you wish (i.e. personal pc or laptop) as long as you use Java 1.5 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 the course web site.

File Checklist
Account.java
AccountManager.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 hold 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 same X amount from the same account. 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 thread deposits into account #1234 the amount 10 CAD * 10000000 iterations. On the other side, one withdrawer thread withdraws from account #1234 the same amount 10 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
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.

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 task3 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 task4 then copy the java files from task 3 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 3 and task 4, what is the advantage of synchronized block over synchronized method?

Deliverables
Submit your assignment electronically via https://fis.encs.concordia.ca/eas/. 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 EAS.** 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

1. API: http://java.sun.com/j2se/1.3/docs/api/
2. Object: http://java.sun.com/j2se/1.3/docs/api/java/lang/Object.html
3. wait(): http://java.sun.com/j2se/1.3/docs/api/java/lang/Object.html#wait()
4. notify(): http://java.sun.com/j2se/1.3/docs/api/java/lang/Object.html#notify()
5. notifyAll(): http://java.sun.com/j2se/1.3/docs/api/java/lang/Object.html#notifyAll()
6. Thread: http://java.sun.com/j2se/1.3/docs/api/java/lang/Thread.html
7. start(): http://java.sun.com/j2se/1.3/docs/api/java/lang/Thread.html#start()
8. run(): http://java.sun.com/j2se/1.3/docs/api/java/lang/Thread.html#run()
9. join() http://java.sun.com/j2se/1.3/docs/api/java/lang/Thread.html#join()