CSCI 340, Fall 2017
Instructor: Simina Fluture, PhD
Project 1 – **Due date: December Tuesday, 12**

Using Java programming, synchronize the threads, in the context of the problem. Closely follow the implementation requirements. The synchronization should be implemented through Java semaphores and operations on semaphores (acquire and release)

For Mutual Exclusion implementation use **Mutex semaphores**, no volatile variables.

For semaphore constructors, use ONLY: Semaphore(int permits)
Creates a Semaphore with the given number of permits and nonfair fairness setting.

As methods use ONLY: acquire(), release(); You can also use:

getQueueLength()
Returns an estimate of the number of threads waiting to acquire.

hasQueuedThreads()
Queries whether any threads are waiting to acquire.

DO NOT USE ANY OF THE OTHER METHODS of the semaphore's class, besides the ones mentioned above.

Any wait must be implemented using P(semaphores) (acquire), any shared variable must be protected by a mutex semaphore such that Mutual Exclusion is implemented.

Document your project and explain the purpose and the initialization of each semaphore.

DO NOT use synchronized methods (beside the operations on semaphores).

Do NOT use wait( ), notify( ) or notifyAll( ) as monitor methods. Whenever a synchronization issue can be resolved use semaphores and not a different type of implementation.

You should keep the concurrency of the threads as high as possible, however the access to shared structures has to be done in a Mutual Exclusive fashion, using a mutex semaphore.

Many of the activities can be simulated using the sleep(of a random time) method.

Use appropriate System.out.println( ) statements to reflect the time of each particular action done by a specific thread. This is necessary for us to observe how the synchronization is working.

Submission similar to project1. Upload it on Blackboard.

CSCI 340, Fall 2017
Instructor: Simina Fluture, PhD
Project 1 – **Due date: December Tuesday, 12**

Shopping at BALA

A customer walks into a BALA showroom and browses around the store, for a while, before (s)he finds an item (s)he likes. When a customer does so, (s)he must go to the floor-clerks and receive a slip with which (s)he can pay for the item and then take it home (implement this using semaphores)
Floor clerks wait (use semaphores) for customers to arrive; then, they help them with whatever information they need. However, floor clerks can only help one customer at a time; therefore, a customer must wait (use semaphores) for an available clerk to help him/her.

After all of the customers are assisted, the floor clerks will wait to be signaled when the closing time comes (use semaphores; if you also update any shared variable use mutex).

Once a customer gets the information needed from the floor clerk, (s)he will pay for it (simulated by sleep of random time).

Once the item is paid, if the item is a heavy item (**decide this by generating a random number; 30% of time the items should be light weight items**) the customer will have to pick-up the item from the storage room. Once arrived at the storage room, the customer waits (use semaphores) until it is helped.

Initially there are *n_storageClerks* available in the storage room. The items are very heavy so three clerks need to group in a group_size of three for help carrying the furniture of one customer (use semaphores).

Once all of the customers are assisted, all the storage clerks will wait for closing time (use semaphores).

Once the shopping is completed, each customer if s(he) is not the last customer to be done shopping, will just browse around waiting for the closing time (use semaphores). The very last customer to be done shopping will announce to the other customers that it is time to leave.

The very last customer to leave will also signal the clerks that the store can close.

Develop a multithreaded Java program simulating the Shopping at BALA operations.

Your program should have three types of threads:
- **Customer** threads
- **Floor clerk** threads
- **Storage clerk** threads

The number of customers, floor clerks and storage clerks should be read as command line arguments: *e.g.* –c <int> -f <int> -s <int> with default values of 10, 2 and 6 respectively.
In order to simulate different actions you must pick reasonable intervals of random time. Make sure that the execution of the entire program is somewhere between 40 seconds and 90 seconds.

CSCI 340, Fall 2017
Instructor: Simina Fluture, PhD
Project 1 – **Due date: December Tuesday, 12**

Guidelines

1. Do not submit any code that does not compile and run. If there are parts of the code that contain bugs, comment it out and leave the code in. A program that does not compile nor run will not be graded.

2. Closely follow all the requirements of the Project's description.

3. Main class is run by the main thread. The other threads must be manually specified by either implementing the Runnable interface or extending the Thread class. **Separate the classes into separate files. Do not leave all the classes in one file. Create a class for each type of thread. Don't create packages.**

4. The program asks you to create different types of threads. Since there will be more than one instance of the threads, no manual specification of each thread's activity is allowed (e.g. no Clerk1.checkCustomer()).

5. Add the following lines to all the threads you make:

```
public static long time = System.currentTimeMillis();

public void msg(String m) {
    System.out.println("["+(System.currentTimeMillis()-time)+"] "+getName()+": "+m);
}
```

It is recommended to initialize time at the beginning of the main method, so that it will be unique to all threads.

6. There should be printout messages indicating the execution interleaving. Whenever you want to print something from that thread use: msg("some message about what action is simulated");

7. NAME YOUR THREADS or the above lines that were added would mean nothing.
Here's how the constructors could look like (you may use any variant of this as long as each thread is unique and distinguishable):

```
// Default constructor
public RandomThread(int id) {
    setName("RandomThread-" + id);
}
```

8. Design an OOP program. All thread-related tasks must be specified in its respective classes, no class body should be empty.

9. **No** use of **wait( ), notify( ), notifyAll( )** or **synchronized** methods..

10. If a thread needs to block it should be done through wait on semaphores.

11. "Synchronized" is not a FCFS implementation. The “Synchronized” keyword in Java allows a lock on the method, any thread that accesses the lock first will control that block of code; it is used to enforce mutual exclusion on the critical section. **FCFS should be implemented in a queue or other data structure**.

12. DO NOT USE System.exit(0); the threads are supposed to terminate naturally by running to the end of their run methods.

13. Command line arguments must be implemented to allow changes to the **nCustomer, nFloorClerk** and **nStorageClerk** variables.

14. Javadoc is not required. Proper basic commenting explaining the flow of the program, self-explanatory variable names, correct whitespace and indentations are required.

Setting up project/Submission:

In Eclipse:

Name your project as follows: LASTNAME_FIRSTNAME_CSXXX_PY
where LASTNAME is your last name, FIRSTNAME is your first name, XXX is your course, and Y is the current project number.
For example: **Doe_John_CS340_p2**

To submit:

-Right click on your project and click export.
-Click on General (expand it)
-Select Archive File
-Select your project (make sure that .classpath and .project are also selected)
-Click Browse, select where you want to save it to and name it as LASTNAME_FIRSTNAME_CSXXX_PY
-Select Save in **zip format (.zip)**

-Press Finish

UPLOAD THE PROJECT ON BLACKBOARD.

The project must be done individually with no use of other sources including Internet. No plagiarism, No cheating.