Dating Game – cs340 Project1, Fall 2015

Due: Monday, Nov. 30th

Through its implementation, this project will familiarize you with the creation and execution of threads, and with the use of the Thread class methods. In order to synchronize the threads you will have to use (when necessary), run( ), start( ), currentThread( ), getName( ), join( ), yield( ), sleep(time), isAlive( ), getPriority( ), setPriority( ), interrupt( ), synchronized methods.

Dating Game

OBH has a reality game show called the Dating Game where the host, SmartPants, instructs a group of novices on the finer points of socializing. On one particular episode, SmartPants brings the group into a club. The goal of each Contestant is to get phone numbers from potential Dates at the club. He has **num_rounds** to do so.

Three types of threads:

SmartPants
Contestant
Date

Each Contestant arrives at the club on his own (simulated by **sleep(random time)**). When the Contestant arrives, he has to meet SmartPants. Contestant will do **busy waiting** until SmartPants lets him know that he is available to meet. SmartPants will talk to each Contestant in FCFS order (you can use a Boolean array/vector, queue like structure in order to enforce the order).

After the initiation, Contestants enter the club to meet a Date. If there is no Date available, the Contestant will kill some time at the Juice Bar (use **yield( )** and after **sleep(random time)**). If there are available Dates they approach the next available Date. They will use busy waiting for the Date's decision.

Dates wait to be approached by **busy waiting** (you can use a Boolean array/vector). Once approached, she will feel rushed to take a decision. Simulate this by increasing the priority of the Date. After a brief talk (simulated by **sleep(randomtime)**), the Date randomly decides if she is interested enough in the contestant to provide her contact info. Next, her priority will be set back to its default value (use **getPriority( )**, **setPriority( )**).
The Date becomes available again for another contestant.

Make sure that the Contestant will not approach twice the same Date.

The Contestant will have the right to **num_rounds** attempts. After the num_rounds attempts, he will update a shared variable **contestant_done**. This is a shared variable, have it accessed from inside of a **synchronized method**.

The Contestant will go back to SmartPants's table (**sleep(random time)**) to get a pat on the back by interrupting him (use **isInterrupted( )** and **interrupt( )**).

SmartPants will congratulate the Contestant and if all contestants are done, he will announce the end of the show (use a shared variable **show_ends,** and have this check done from inside a **synchronized method).**

Note: After the initiation process, throughout most of the show, SmartPants will sleep for a very long time. If it is interrupted it will check if it is the time to end the show and if not, it will go back to sleep, again, for a long time.

Dates and SmartPants terminate when the show ends.

Contestants linger outside the club to brag (**sleep(random_time)**).
Once a Contestant leaves, he will check to see if the next Contestant (i.e. the Contestant with ID +1) is still in the club (use **isAlive( )** and **join()**). If yes, he will wait to join that Contestant. If the next Contestant is already gone, then he will simply go home.

Before terminating its execution, the Contestant will print an outline containing:

Contestant's name followed by the names of the Dates who provided him with the contact information.

In your implementation in order to refer to a specific thread use **getName()** and **currentThread( )** methods.

The program should take as command line arguments:

Argument	**Default Value**
num_contestant	**9**
num_date	**6**
num_rounds	**3**

Using Java programming, synchronize the two types of threads, Contestant, SmartPants, Dates in the context of the problem. ***Closely follow the implementation requirements****.*

Choose appropriate amount of sleep time(s) that will agree with the content of the story. Note: I didn't write yet the code for this project but from the experience of grading previous semester projects a project should take somewhere between 40 second and at most 1 minute and ½.

Academic integrity must be honored at all times and no code sharing or cheating is permitted. Only submit code that you have written.

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.

4. The program asks you to create different types of threads. For Contestant thread type and Date thread type, there is more than one instance of the thread. No manual specification of each thread's activity is allowed (e.g. no Contestant1.doErrands()).

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);
}
```

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

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** implementation of semaphores or use of **wait( ), notify( ) or notifyAll( )** are allowed.

10. Thread.sleep() is not busy wait. while (expr) {..} is busy wait.

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 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 num_contestant, num_rounds, num_dates.

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

Tips:
-If you run into some synchronization issue, and don't know which thread or threads are causing it, press F11 which will run the program in debug mode. You will clearly see the thread names in the debug perspective.

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: Fluture_Simina_CS340_p1

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**, Create directory structure for files and also Compress

the contents of the file should be checked.
-Press Finish

Email the archive with the specific heading: **CS340 Project # Submission: Last Name, First Name** to simina.fluture@qc.cuny.edu.

You should receive an acknowledgement within 24 hours.

The project must be done individually, not any other sources. No plagiarism, No cheating. Read the academic integrity list one more time.