Java Concurrency

SCOMP

1. Create an application that simulates a multithreaded ordering application in which multiple order takers, each running the application in a separate thread, generate orders (just a number) that are added to the queue that runs in the application's main thread. The orders are then handled by multiple order-handling threads, which remove orders from the queue and display them in the console. One suggestion for the classes you should use is:
    - Order class: Object that represents an order
    - OrderQueue class: Uses LinkedList to implement a list of orders
    - OrderTaker class: creates and adds orders to the queue
    - OrderHandler class: removes orders from the queue

2. Create an application that simulates a concurrent ticket selling. The number of tickets available is stored in a shared variable. There may exist N different sellers wishing to sell tickets, these are represented by threads. Implement a solution where the access to memory is protected using Semaphores and where it is possible to interrupt the sellers.

3. Create an application that simulates a bank transfer operation. Your application should have an Account object and a method for the transfer operation with the following signature:

    public boolean transferMoney(Account fromAcct,Account toAcct, EurosAmount amount)

    - Access to Account objects should be exclusive.
    - In your solution, can a deadlock occur? If yes, can you solve it by reordering the statements?

4. Admit we want to simulate how a taxi infrastructure works:
    - Taxi central receives requests for taxis and puts them on a waiting list.
    - Each request must include the number of passengers and the address were the taxi should pick them up.
    - Whenever a taxi is free it should look at the waiting list in order to satisfy the first request.
    - Each taxi has a maximum number of seats.
    - Whenever the number of passengers exceeds the number of seats available in the taxi that is available to pick them up, the taxi should pick up the maximum of passengers it can carry and allow the remaining ones for the next taxi.

5. Explore the Java framework and try using an Executor which allows to have a fixed number of threads and reuse those threads to accomplish a given task. Admit we want to sum a given vector of integers in parallel using an Executor:

- Create an object Sum which implements Runnable and that given an initial and final index of the vector of integers, sums the values in that interval and shows the result.
- Use a fixed number of threads (e.g. 5).
- Divide the array in a number of partitions higher than the number of available threads (e.g. 10).
- Create the same number of instances of Sum as the number of partitions and submit to the Executor.

6. In the previous exercise it is possible to compute the partial sums of the vector but it is not possible to return those values (since *Runnable*cannot return a value) to compute the global sum of all the partitions. Change your code to explore the use of *Callable* and *Future* instead of *Runnable* and be able to return values from tasks.