Assignment 4: Bank Simulation

In this assignment, you'll be building a banking simulation that will involve gathering statistics, building tellers, receptionists, clients, and using event queues. This assignment in reality is a 3-in-one exercise :

Programming with the Java Collection classes (a priority queue and an ordinary FIFO (First In First Out) queue will be required)

Design of classes

Introduction to a pattern (Event queue) used very often in Discrete Event Simulation

The general pattern of these types of simulations is this: we have a **Clock..** which is simply an integer counter, and an **Event queue** in which we keep *a list of **events** that will happen in the future*. An event is something that changes the state of the system.. for example an arrival of a client, or a departure of a client from a particular server. The kinds of events we will have depend upon the specific problem.
In this simulation we can identify initially five types of Events.

Arrival of a client to the Bank (or the receptionist, it could be useful to separate these two events though)

Departure of a client from the Receptionist.

Arrival of a client to the Teller and

Departure of a client from the Teller.

Each event is has the following attributes:
The **time** it will occur
What is to be done when the event is processed. This can coded in a method ***process***( ), which all events should have. Obviously each event type has different code in it's ***process***( ).

The basic idea of the *event list pattern* is to use a list of events to control the simulation:
We keep a list of events that will happen, *ordered by increasing time of occurrence*, (and that's why a priority queue is the ideal data structure to use (with smallest time = highest priority))

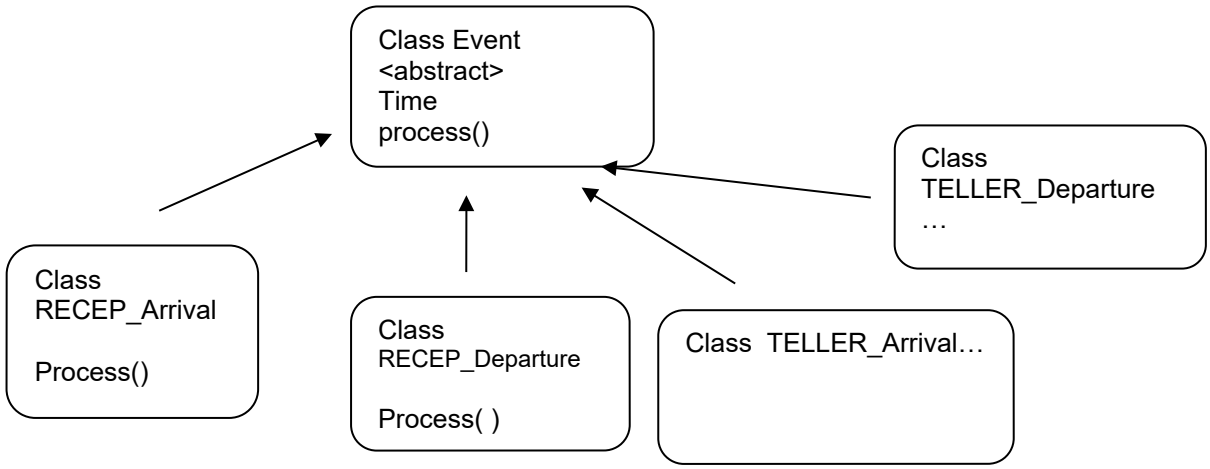
The algorithm now is simple and very general.

```
While the Clock is less than the end of simulation time DO
    Remove an event from the event queue,
    Process it. (That is.. if it is an Arrival, do what must be done at an arrival, if it is a
            Departure… do what must be done at a departure.)
```

Now is the time for some Object Oriented thinking.

Instead of having a separate method that asks each event removed from the Event queue "What are you? If you are an Arrival I will do this, if you are a Departure I will do that.." it is much more useful to have each Event class handle its own processing in a process() method. Now we can just tell each event X to process itself by calling its own process method X.process( ).

This immediately suggests an architecture for our Event classes.



All four classes override the inherited ***process***( ) method so for example, the RECEP _Arrival process( ) does what must be done at an arrival of a client at the Gas pump and the PUMP _Departure process( ) does what must be done at a departure.

Don't forget that Events are objects that tell us *What* will happen *When*, and in addition, know how to process themselves. (through their method **process**( ))

The Event queue will now be a priority queue of *Event objects*, which means that we can store both arrival and departure events in this queue.. (plus any event you might want to add later!)
Since when we are inserting these events in the priority queue we have to compare them with each other, ***class Event must implement interface Comparator*** (i.e. must implement a **compare**( ) method that works the same as the compare for Strings.. look it up if you don't remember how this works).

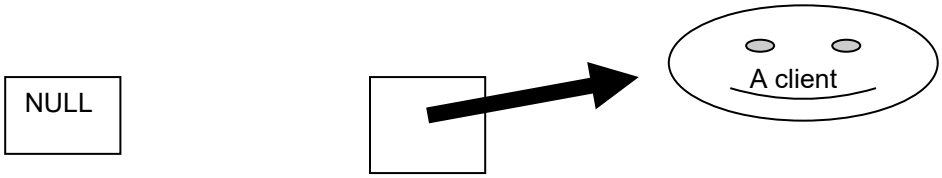
In general a.compare(b) returns:

0 if a equals b
-1 if a < b
+1 if a > b).

The RECEPTIONIST and TELLER servers

We need a representation of the servers that are serving the clients. In this particular example a simple variable of type **Client** (see description further down) is enough.

When there is nobody being served, it's value is null. When a client is being served, we show that by *assigning the Client reference* to it.



An empty desk A desk serving a client

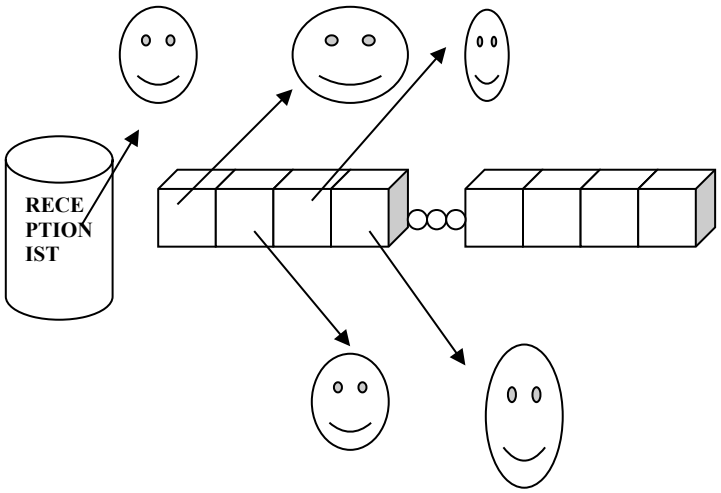
THE CLIENT CLASS

Since we will be dealing with clients, we need a class **Client**. This class will have a data field that stores the **time** the client arrived (remember that this is simply an integer…we are dealing with simulated time not real time) and a data field that holds the number of **transactions** (randomly determined by the client constructor).

When we create a client object we call a constructor with a single parameter, the time of arrival. The constructor also initializes the number of **transactions** by using a random number generator to produce a number between 1 and 100.

THE RECEPTIONIST and TELLER QUEUES

When a desk is busy, and an arrival occurs, the client who just arrived is *made to wait in a queue*.
This is the Client Queue, (a separate queue for each server) and it is a simple FIFO queue of Client



The client queue for each server holds the clients that are waiting for that particular server to become empty (through a departure)

It should be a simple First In First Out queue with insertions at the tail and removals from the head

THE STATISTICS CLASS

The whole purpose of the exercise is to gather statistics. So, why not have a class that has the appropriate methods to do this and return the average waiting time at the end. Ensure you gather at least 2 statistics here.

The RANDBOX class

Many times during the simulation we require a number drawn from a certain distribution. It is neater to put these methods that return these numbers within a class called RandBox instead of having them as static methods in our main class.

RandBox will contain a static method ***expo(rmean)*** that, when called, will returns a random value from a (negative) exponential distribution with average *rmean*. (i.e. if you call expo(100.0) it will return some value.. if you call it a million times with the same parameter and calculate the average it will approach 100.0 and if you plot the values they will be distributed negatively exponentially...MAGIC!!)

We are making a major assumption that inter-arrival times are distributed like that…it makes sense though that we will have a small number of large inter-arrival times and a relatively large number of small inter-arrival times. The actual real curve is usually determined by empirical observation. The code for expo() within the class RandBox is the following:

```
public class RandBox extends Random
{
   public static double expo( double mean)
    {
      double x = Math.random();
      x = -mean * Math.log(x);
      return x;
    }
}
```

Now, whenever you need an **interarrival** time you can call **RandBox.expo(120.0)**
Or a service time **RandBox.expo(transactions * 60)**

Of course, we don't want to use numbers like the above in our program. To make it easier to modify we assign these to constants.

private final double INTERARRIVAL_TIME = 120.0;
private final double AVG_TIME_PER_TRNSACTION = 60;

So far so good…now what about the functionality of the specific event classes? What kind of services would we want from them?

As far as the queue classes go, you need a **priority** queue and an **ordinary** queue..either use whatever you can find in the Java Collection classes, or make your own.

WHAT HAPPENS AT AN ARRIVAL?

The process( ) method of an Arrival event should do the following (Teller is similar though it doesn't schedule a new Receptionist Arrival):

1. ***Move the clock to the time of the arrival event***
2. Create a client object with the current time of the clock as arrival time
3. If the receptionist is busy then *insert the client in the client queue for the receptionist*
4. ELSE (the receptionist is free i.e. it is null) place the client in the receptionist and, very important ***schedule a new RECEP_departure.***
   To schedule a new ***RECEP*** _Departure you need to make one by
   a) calling the constructor of the RECEP _Departure class and
   b) to insert it in the event queue. The time when the new Departure will happen is determined by calling the function expo with a parameter which should be AVG_TIME_PER_TRANSACTION * number of courses to get a service time, and *adding that to the current value of the clock*. (i.e. the departure will happen at time: NOW + a random service time)
5. ***SCHEDULE a new Receptionist Arrival at time NOW + RandBox.expo (INTERARRIVAL_TIME)***

WHAT HAPPENS AT A RECEP_DEPARTURE?

The process( ) method of a Departure event should do the following:

1. ***Move the clock to the time of the Departure event***
2. Set the RECEPTIONIST to null
3. Schedule a TELLER_arrival of the client that is leaving the Receptionist
4. If the client queue is NOT empty then remove the first client , set him to the RECEPTIONIST and *schedule a RECEP_departure* (as described above)
   Calculate the time the client has been in the queue (CLOCK - the client arrival time) and give that time to the statistics object

WHAT HAPPENS AT A TELLER_DEPARTURE ?

Think about it

HOW DO WE PUT ALL THAT TOGETHER ?

Here's one way. Have one class (let's call it ***BankSim***) which contains the Event queue, the client queue, a clock, and two variables (of type Client) RECEPTIONIST and TELLER. This class will have a constructor with parameters for the constants INTERARRIVAL_TIME and AVG_TIME_PER_TRANSACTION.

Class BankSim will have a method ***run(simulation_time)*** that will "run" the simulation. The algorithm of this method is simple:

- Schedule a first arrival
- While the clock is less than simulation_time do:
    - Remove an event from the Eventqueue
    - Call it's process( ) method.
- Call methods that will give you the accumulated statistics
- Print them out

PROBLEMS YOU WILL HAVE TO THINK ABOUT (...AND SOLVE)

The process() methods of the Arrival and Departure classes need to be able to access the client Queue the Desk and the Event queue.

WHAT GOES IN THE main( ) METHOD ?

Very simple..

```
public static void main(String[] args)
    {
          BankSim  s = new BankSim (120.0 , 60.0);
           s.run(5 * 8 * 60 * 60); // seconds in 5  8-hr workdays

    } // end main()
```

Grading Criteria:

Style/submission guidelines: https://gmierzwinski.github.io/bishops/cs321/style_guidelines.html
The grading criteria for this assignment is subject to change.

Comments, Formatting, & Readability	**10 Marks**
Submission Guidelines	**10 Marks**
Events	**20 Marks**
Event Queues	**10 Marks**
Servers (Reception, and Tellers)	**20 Marks**
Bank Simulator	**10 Marks**
Client	**5 Marks**
Statistics	**5 Marks**
Total	**90 Marks**