Lab 2

Submission deadline: 2359, Sunday, September 10, 2017.

Prerequisites

Assume that students are already familiar with:

- the CS2030 laboratory environment [../unix/index.html]
- how to compile and run Java programs
- familiar with standard I/O and I/O redirection
- comfortable with Java syntax
- looking up Java API documentation
- understand the concepts of encapsulation and using encapsulated objects

Learning Objectives

After completing this lab, students should:

- be more comfortable with looking at a complex problem and identify data and procedures that should be kept within an abstraction barrier. In other words, be more comfortable with creating own encapsulated class.
- be more comfortable with basic Java syntax and semantics, particularly when creating classes from scratch.

Setup

Login to `cs2030-i`, copy the files from `~cs2030/lab02` to your local directory under your home `~/lab02`. You should see one java file (`LabTwo.java`) and a few data files (`TESTDATA1.txt`, `TESTDATA2.txt`, ..., `TESTDATA5.txt`)

Task

`LabTwo.java` implements a working discrete event simulator that simulates customers being served by a server. It is written in procedural style with no encapsulation. Your task, in this lab, is to rewrite this simulator with OO style, by properly using encapsulation to create abstraction barriers to the various variables and methods.

Grading

This lab contributes 4 marks to your final grade (100 marks). You will get 0-3 marks for your design and encapsulation of the classes. You will get 1 mark if you code works correctly (but only if you score at least 1 mark for your design and use of encapsulation [1] [#fn:1])

Discrete Event Simulator

A discrete event simulator is a software that simulates a system (often modeled after the real world) with events and states. An event occurs at a particular time, and each event alter the states of the system, and may generate more events. A discrete event simulator can be used to study many complex real world systems. The term *discrete* refers to the fact that, the states remain unchanged between two events, and therefore, the simulator can *jump* from the time of one event to another, instead of following the clock in real time. The simulator typically keeps track of some statistics to measure the performance of the system.

In this lab, we start with simulating a specific situation:

- We have a shop with a *server* (a person providing service to customer).
- The server can serve one customer at a time.

- We assume for now that the server takes constant time to serve a customer. The time taken to serve is called *service time*.
- When a customer arrives:
  - if the server is idle (not serving any customer), then the server serves the customer immediately (no waiting).
  - if the server is serving another customer, then the customer that just arrives waits.
  - if the server is serving one customer, and another customer is waiting, then the customer that just arrives just leave (no waiting) and go elsewhere. In other words, there is at most one waiting customer.
- When the server is done serving a customer:
  - the served customer leaves.
  - if there is another customer waiting, the server starts serving the waiting customer immediately.
  - if there is no waiting customer, then server becomes idle again.

We are interested in the following. Given a sequence of customer arrivals (time of each arrival is given):

- What is the average waiting time for customers that has been served?
- How many customers are served?
- How many customers left without being served?

In your Lab 2, you are given a simple discrete event simulator to answer the questions above. There are two classes: `Simulator` and `Event`.

Class `Event`

The event class is written in procedural style, not unlike a `struct` in C. All members are public, and there is no method. Each `Event` keeps track of two information: the `time` the event occurs, and `eventType`, which signifies what type of events is this. Instead of using time like 9:45pm, we simply and represent time as a double value.

```
1   static class Event {
2     public double time; // The time this event will occur
3     public int eventType; // The type of event, indicates what sh
4   }
```

We handle two types of events for this particular scenario: an event of type `CUSTOMER_ARRIVE` means that a customer arrives during this event; while an event of type `CUSTOMER_DONE` means that the customer is done being served. `CUSTOMER_ARRIVE` events are created based on the given input. `CUSTOMER_DONE` events are created and scheduled to occur sometime into the future when a customer is being served.

```
1   public static final int CUSTOMER_ARRIVE = 1;
2   public static final int CUSTOMER_DONE = 2;
```

Class `Simulator`

The simulator class is again written in procedural style. All members are public, and there is no method.

The `Simulator` class contains two configuration parameters, `MAX_NUMBER_OF_EVENTS` indicates the maximum number of events that the simulator can store at one time; `SERVICE_TIME` indicates the time the server takes to serve a customer.

```
1     public int MAX_NUMBER_OF_EVENTS = 100; // Maximum number of e
2     public double SERVICE_TIME = 1.0; // Time spent serving a cus
```

The `events` is an array of `Event` that store all events scheduled for the future in the simulator.

```
1     public Event[] events; // Array of events, order of events no
2     public int numOfEvents; // The number of events in the event
```

The simulator needs to keep track of three states:

- is a customer being served?

- is a customer waiting?
- if a customer is waiting, when did he start waiting?

These states are represented as:

```
1      public boolean customerBeingServed; // is a customer currentl
2      public boolean customerWaiting; // is a customer currently wa
3      public double timeStartedWaiting; // the time the current wai
```

Remember we are interested in the following statistics:

- What is the average waiting time for customers that has been served?
- How many customers are served?
- How many customers left without being served?

which can be computing from the following members:

```
1      public double totalWaitingTime; // total time everyone spent
2      public int totalNumOfServedCustomer; // how many customer has
3      public int totalNumOfLostCustomer; // how many customer has b
```

Finally, for debugging purposes, the simulator assigns unique IDs 1, 2, 3, ... to the customers, in the order of their arrivals. It then keeps track of the ID of the customer being served (if any) and the customer waiting to be served.

```
1      public int lastCustomerId; // starts from 0 and increases as
2      public int servedCustomerId; // id of the customer being serv
3      public int waitingCustomerId; // id of the customer currently
```

Interaction between `Simulator` and `Event`

We create a `Simulator` by calling the method:

```
1    static Simulator createSimulator() {..}
```

and an `Event` by calling the method, specifying `when` the event will occur, and the `type` of the event.

```
1   static Event createEvent(double when, int type)  {..}
```

We can schedule the event `e` to be executed by simulator `sim` by calling:

```
1   static boolean scheduleEventInSimulator(Event e, Simulator sim)
```

This method will return `true` if the event is scheduled succesfully, `false` if the simulator run out of space to store the event (i.e., `MAX_NUMBER_OF_EVENTS` is reached).

We always execute the events in increasing sequence of their time. Once the simulator starts running, it repeatedly find the next event with the smallest timestamp (i.e., earliest event), remove it from the list of events, and execute the event. The simulator stops when there is no more event to run.

```
1   static void runSimulator(Simulator sim) {
2     while (sim.numOfEvents > 0) {
3       Event e = getNextEarliestEvent(sim);
4       simulateEvent(sim, e);
5     }
6   }
```

Here, `e = getNextEarliestEvent(sim)` removes and returns the earliest event in the simulator, and `simulateEvent(sim, e)` update the states of the simulator according to the type of the event `e`.

Simulated System

The logic of the system being simulator (i.e., behavior of customers and server) is implemented in `simulateEvent`. There are four methods being called from here:

- `serveCustomer(sim, time, id)`: called to start serving a customer with ID `id`
- `makeCustomerWait(sim, time, id)`: called to make the customer with ID `id` wait
- `customerLeaves(sim, time, id)`: called when the customer with ID `id` who just arrived leaves immediately (as someone else is waiting)

- `servedWaitingCustomer(sim, time)` : called to start serving the customer that is currently waiting.

You should read through `LabTwo.java` and clarify if you are not sure about any part of the given code.

Input: Arrival Time

The input consists of a sequence of double values, each is the arrival time of a customer (in any order). We can read from standard input (if no command line argument is given)

```
java LabTwo
java LabTwo  < TESTDATA1.txt
```

or read from a given filename

```
java LabTwo TESTDATA1
```

Given an input, the output might not be deterministic, since if two events occur at exactly the same time, we break the ties arbitarily. For this reason, we will only test your code with input where no two events occur at exactly the same time.

Your Task

The given `LabTwo.java` is written in C style, no minimal encapsulation. As you read through the code, you should appreciate how messy and difficult to understand the code is.

Your mission, in Lab 2, is to rewrite the code using encapsulation, applying OO paradigm, properly maintain the abstraction barrier when the objects interact. Here are some rules:

- You can add as many classes as you like. Each class must be in its own `.java` file

- The `main` method should remain in a class named `LabTwo`. We must be able to run your code with:

```
javac *.java
java LabTwo < TESTDATA1.txt
```

- You must not change the formatting of the *last line* of output (`System.out.printf("%.3f %d %d", ..")`). We rely on it to check for correctness of your logic.

Submission

When you are ready to submit your lab, on `cs2030-i`, run the script

```
~cs2030/submit02
```

which will copy all files matching `*.java` (and nothing else) from your `~/lab02` directory on `cs2030-i` to an internal grading directory. We will test compile and test run with a tiny sample input to make sure that your submission is OK.

You can submit multiple times, but only the most recent submission will be graded.

Extra Java Stuff

You are exposed to three new Java syntax/class in this Lab:

- Nested classes: In the code given to you, we define `Simulator` and `Event` within the class `LabTwo`. This is called *nested class* in Java. Usually, this is useful if we need to create a class that is only useful to another class. We can group logically relevant classes together.
- `assert` keyword: `assert` works like in C and Python. You use `assert` to check for conditions that has to be true in your code. If an assertion fails, the program will bail, informing you what went wrong. This is useful to catch bugs quickly. Use this by passing a `-ea` (enable assertions) flag when running a Java program e.g. `java -ea LabTwo TESTDATA1.txt`

- `FileReader` [http://docs.oracle.com/javase/8/docs/api/?java/io/FileReader.html]: a useful class for reading a stream of characters from a file.

---

1. This is so that, if you just take the code we give you and submit as is, you cannot claim that it works so we must have you 1 mark.