Lab 2: Comments

Correctness (1 mark)

You get the 1 mark for correctness if your program prints these same outputs as the provided skeleton code:

```
Test Case #1: 0.000 1 0
Test Case #2: 0.000 4 0
Test Case #3: 0.450 2 8
Test Case #4: 0.614 7 0
Test Case #5: 0.405 76 24
```

Design (3 marks)

- 0 marks: No effort made at encapsulation. You also do not get the 1 mark for correctness.
- 1 mark: Code is shifted wholesale into different classes and files. Classes are used to house static methods instead of being treated as objects. `Simulator` is still being passed around as an argument in all the methods.
- 2 marks: All fields (except constants) are not exposed to external modification (via setting to private) and overall no major mistakes made. This is achievable with just the `LabTwo`, `Simulator`, and `Event` classes.
- 3 marks: Customer-handling logic is abstracted away from `Simulator`. To achieve this, naturally more classes are needed so that there is a better separation of concerns, and every class only has one single responsibility. Adding a `Customer` and a `Server` class (and implementing them well of course) will score 3 marks very stably.

These are the rubrics for the different scores for design. We were generally lenient, so as long as your submission is roughly at a certain level, we will give you the marks even if you missed out a few things. Take note that many of the submissions that scored 3 marks still have lots of room for improvement - and we **don't** mean by doing more advanced things like inheritance or interfaces.

General Comments on Design

One of the learning objectives of this lab is to identify **data (i.e. state / fields)** and **procedures (i.e. methods)** that should be kept within an abstraction barrier, so that classes do not directly manipulate the state of other classes.

Proper encapsulation will allow the specific implementation details of each class to be abstracted away from external classes, by only exposing a carefully selected set of public methods. This will make the relationships between all your classes much simpler, because they only interact with each other via those public methods.

Let's say we have two classes: `A` and `B`. `A` shouldn't know too much about `B`, otherwise it becomes highly-dependant on the internals of `B`. It's even worse if `B` is also highly-dependant on `A`. In software engineering terminology, we say that classes `A` and `B` are tighly-coupled [https://en.wikipedia.org/wiki/Coupling_(computer_programming)] (not tested, FYI only). Some of you will realize that your submissions are very "messy" because of this.

For Lab 2, we do not penalize having getters and setters. In the real world, they may be absolutely necessary in some situations. But getters and setters are a blunt tool, and having too many of them (we've seen submissions with >10) is probably a symptom that your design can be improved. You are also most likely violating the "Tell, Don't Ask" principle.

Tell, Don't Ask

a.k.a. Writing code declaratively instead of procedurally

Here are two articles that talk about this principle:

https://pragprog.com/articles/tell-dont-ask [https://pragprog.com/articles/tell-dont-ask]

> Procedural code gets information then makes decisions. Object-oriented code tells objects to do things.

https://robots.thoughtbot.com/tell-dont-ask [https://robots.thoughtbot.com/tell-dont-ask]

> Good OOP is about telling objects what you want done, not querying an object and acting on its behalf. Data and operations that depend on that data belong in the same object.

Many students used getter methods to "ask" an object for information, handled some logic, and then used setter methods to update the fields of that object. While this is better than making those fields `public` altogether, this often leads to a poorly-defined abstraction barrier. It is almost all the time better to "tell" your object to do something instead.

The Ruby programming language
The code in the second link is written in Ruby, another object-oriented language like Java, except that it is dynamically-typed. Even though it's a different language, it should be easy enough to understand for you to appreciate the examples in the article.

"I still don't understand!"

Here's a code snippet from the `Simulator` class of a 3 mark submission. Compare this with the original method we provided. This is a very good example of what a well-abstracted program looks like, and hopefully it helps to illustrate the concepts mentioned above.

```
1  private void simulateEvent(Event e) {
2    switch (e.getType()) {
3      case Event.CUSTOMER_ARRIVE:
4        Customer c = new Customer(e.getTime());
5        this.server.handleNewCustomer(c);
6        break;
7
8      case Event.CUSTOMER_DONE:
```

```
        this.server.finishServingCurrentCustomer(e.getTime());
        break;

      default:
        // ...
    }
  }
```

This is a reasonable way to use getters like `getTime()` and `getType()`. But you are highly encouraged to think about how you can avoid them for the subsequent labs.

Common Issues

- **Use constructors to directly instantiate Event and Simulator objects.** You need to tell the class what to do when someone does `new ClassName()`.
- **The expected initial state of an object should always be defined in the constructor. Don't pass in those initial values as arguments.** Otherwise, someone can possibly instantiate an object that has a wrong starting state. Some students had a `Simulator` constructor that took in 7 arguments. This violates encapsulation because `LabTwo` now needs to know that `Simulator` has fields such as `customerWaiting` and `customerBeingServed`, and worse, set them to `false` on `Simulator`'s behalf. You should try to guarantee that every object contains valid values right from the moment it gets created.
- **Also don't provide a "default constructor" that sets dummy values.** For example, some students had an extra `Event` constructor that took in no arguments and set `time` and `eventType` to 0. Can you see how this might go wrong? Don't give yourself (and other people) an option to do things that will lead to unexpected mistakes down the road.
- **Some students implemented a `public Event getEventAt(int index)` method in the `Simulator`.** This clearly breaks the abstraction barrier, because external classes should never have direct access to the ordering of `Event`s and how they are stored inside `Simulator`.
- **A few students misunderstood the meaning of encapsulation.** If a method deals with `Event` objects or has the word "event" in its name, that doesn't mean it should belong inside the `Event` class.

- **Another way to improve your design is to shift `printStats()` into the `run()` method of your `Simulator`.** `LabTwo` doesn't need to know that `Simulator` prints stats when it finishes running. Instead, `Simulator` maintains ownership and control over that behavior. In a 3 mark submission with well-separated concerns, the `Simulator` would fetch stats such as `totalNumOfCustomersServed` from the `Server` class.
- **Constants should be declared *and assigned* at one go, with the proper modifiers.** Here's how you should do it: `private static final MY_CONSTANT = 123`. Changing `private` to `public` is perfectly acceptable if many classes need access to that constant (e.g. `Event.CUSTOMER_ARRIVE`). If a "constant" might have a different value depending on arguments that you pass in, then it's not a constant anymore, is it?
- **Don't define constants in more than one place.** You should only have one source of truth. To achieve this, some students created a `Constants` class and put all their constants in there, but the correct way would be to put them in their appropriate classes. Do also note that not all constants need to be `public` (e.g. `MAX_NUMBER_OF_EVENTS` should be encapsulated in `Simulator`).
- **Avoid nesting classes.** There are very few situations where nested classes are acceptable.
- **Methods for internal use only should all be `private`.** For that matter, all fields and methods should either be `public` or `private`. Don't be lazy and skimp on the access modifier - so many students left `createScanner()` exposed to classes outside of `LabTwo`. The only time you might need to think about other levels of access control is when you're dealing with subclasses.
- It's not strictly required but it's a good practice to **use `this` to refer to fields** so that you distinguish them from other variables or arguments. Methods in the same class, however, should just be called directly without `this`. Same for static fields in the same class.
- When you're accessing a static method or field from outside the class, **always reference the class rather than the instance**. For example, do `Event.CUSTOMER_ARRIVE` and not `e.CUSTOMER_ARRIVE`.

- **Simplicity is underrated.** Always prefer to structure your code in the simplest way possible. Writing simple code is actually very difficult and takes a lot of conscious effort.
- **There is an exception to everything.** Well, almost everything. But you better make sure you know the rules well before you try and break them.