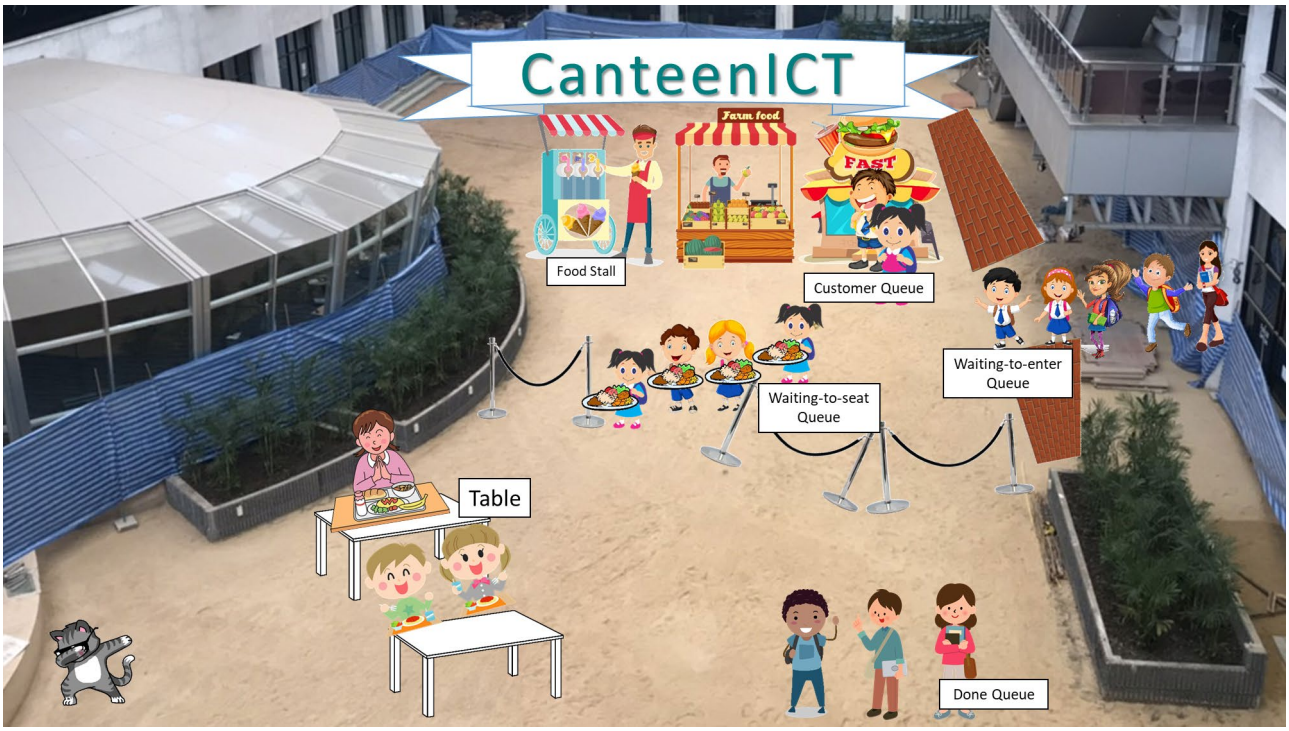
ITCS 209 – Object Oriented Programming

Project 01: CanteenICT Simulation



Update Logs:
2023-02-08 Initial Version 2.0

Due: Monday, March 13, 2023 11:59PM

Learning Objectives:

In the course of implementing this programming project, you will learn some of the basic concepts of object-oriented programming and how to apply them to practical, real-world programming applications. Specifically, upon accomplishing this project, we expect you to be able to:

1. Differentiate and explain the purposes of:
   - private, public, static, and final class variables.
   - static, void, and value-returning methods.
2. Implement classes.
3. Instantiate objects from a class and use constructors to initialize objects.
4. Model entities and situations using objects, each of which has certain properties and exhibits certain abilities, along with arrays, ArrayList, and loops to functionalize a workable program.
5. Strengthen your overall programming skills.
6. Get a glimpse of needs for abstraction, interfaces, inheritance, and polymorphism.
7. Enjoy coding with Java.

Introduction:

You have a brilliant idea that a canteen should be constructed to facilitate hungry ICT students and visitors in the innovative space area. You would like to present your idea to the ICT board with a feasibility analysis. For example, how many food stalls can be built with a limited budget? How many tables should be arranged? How long does it take to serve all the customers? Therefore, you have an idea to write a program that simulates a canteen scenario where customers queue up to enter the canteen, purchase food, be seated, eat, and finish their meals.

CanteenICT Components:

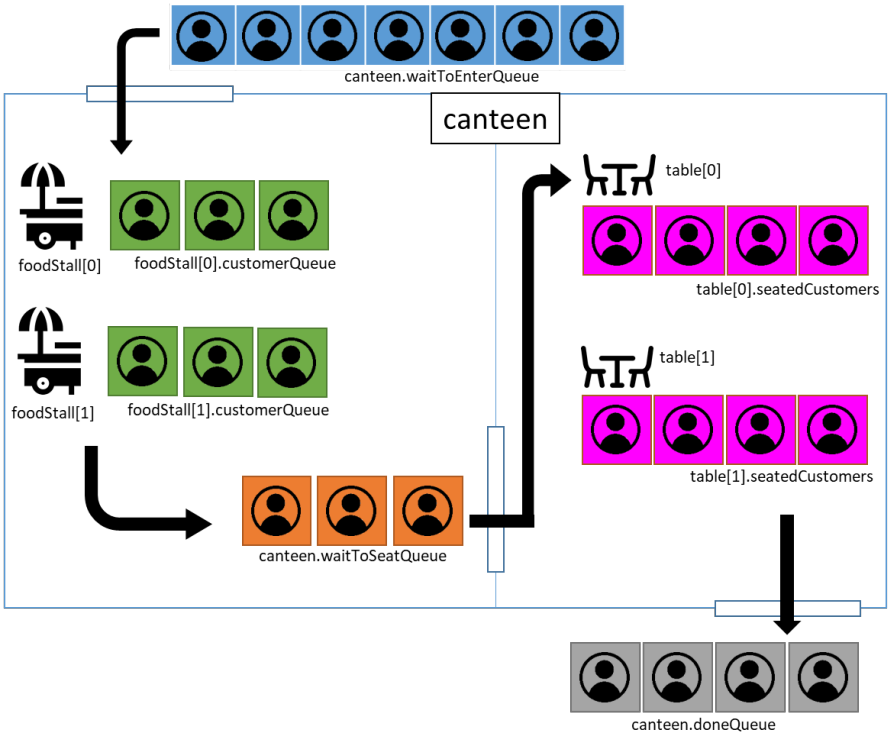


Figure 1: Canteen Components

CanteenICT discrete-simulates the flow of customers from waiting to enter the canteen until finishing their meals, as depicted in Figure 1. The simulation system involves the following components.

Food Stall

A food stall sells food to customers in its customer queue. Constant MAX_QUEUE defines the maximum number of customers waiting in the customer queue. Different types of food are defined in the enum `Menu`, currently including `NOODLES`, `DESSERT`, `MEAT`, `SALAD`, and `BEVERAGE`. Each type of food has a different installation cost, cooking time, and eating time. For example, if the faculty wants a food stall to sell noodles, it must pay 4,000 baht for the installation (to buy equipment, ingredients, etc.). Additionally, the food stall takes 2 time steps to cook a bowl of noodles. A customer takes 6 time steps to eat a bowl of noodles. The installation cost, cooking time, and eating time for each food type are defined in `INSTALLATION_COST`, `COOKING_TIME`, and `EAT_TIME`, respectively.

A food stall is initialized with a name, available menu, and an empty customer queue. The implementation of the FoodStall class is in FoodStall.java. You may find some of the existing implementations helpful. Feel free to modify this file by adding variables and methods to facilitate your algorithm. However, you must not modify code in the "`DO NOT MODIFY`" zones.

Customer

A customer can be one of the following types, `DEFAULT`, `STUDENT`, `PROFESSOR`, `ATHLETE`, or `ICTSTUDENT`, defined in the enum `CustomerType`. For the regular part of this project, a customer is assumed to be a `DEFAULT` customer. The other types of customers are for the bonus credits. A default customer requires the following dishes to satisfy his/her meal:

Menu	Quantity
NOODLES	1
DESSERT	1
MEAT	1
SALAD	1
BEVERAGE	1

Besides the customer's type, a customer can choose how he/she will pay for the meal. There are three payment methods, `DEFAULT`, `CASH`, and `MOBILE`, defined in the enum `Payment`. Each type of payment will take 3, 2, and 1 time period(s) to complete the payment transaction respectively.

Each customer also has a `state` attribute to identify his/her activities and where he/she is in the canteen at each time period.

State	Activity	Location at the end of each time period
0	The customer enters the wait-to-enter queue and waits to enter a food stall queue.	Wait-to-enter queue
1	When the food stall queue is available, the customer queues up at the food stall, and waits to order.	Food stall queue
2	The customer orders food and waits for food to be ready in the food stall queue.	Food stall queue
3	When the food is ready, the customer makes a payment and waits for the payment transaction to be completed in the food stall queue.	Food stall queue
4	When the payment is done, the customer retrieves food, enters the wait-to-seat queue, and waits to sit at the table.	Wait-to-seat queue
5	When the table is available, the customer goes to sit at the table.	Seat at the table
6	The customer starts eating at the table until finished.	Seat at the table
7	When the customer finished eating, the customer goes to the done queue.	Done queue

The `Customer` class is partially implemented in Customer.java. Besides understanding the provided skeleton of the code, you are required to implement the method `takeAction()`, which the canteen will routinely call at every time step. Feel free to modify this file by adding variables and methods to facilitate your algorithm. However, you must not modify code in the "`DO NOT MODIFY`" zones.

Table

A table is where customers sit while eating their food. Each table is assigned a unique ID. The list of seated customers is maintained in `seatedCustomers`. The constant `MAX_SEATS` defines the maximum number of customers that a table can seat. The implementation of the class `Table` is in Table.java. You may find some of the existing functionalities useful. Feel free to modify this file by adding variables and methods to facilitate your algorithm. However, you must not modify code in the "`DO NOT MODIFY`" zones.

CanteenICT

A CanteenICT (in CanteenICT.java) houses all the components, checks for installation validity, and is the main simulation engine. Specifically, a CanteenICT instance maintains the list of all customers (`allCustomers`), wait-to-enter queue (`waitToEnterQueue`), wait-to-seat queue (`waitToSeatQueue`), list of finished customers (`doneQueue`), list of food stalls (`foodStalls`), and the list of tables (`tables`). You are required to understand the existing skeleton of the code and implement the missing code in certain methods such as `getInstallCost()` and `validateCanteen()`. You may find some of the existing functionalities useful. Feel free to modify this file by adding variables and methods to facilitate your algorithm. However, you must not modify code in the "`DO NOT MODIFY`" zones.

The Simulation Algorithm:

CanteenICT.simulate()

After initializing the canteen with customers, food stalls, and tables, the canteen must first be validated to check for certain conditions that prevent the simulation from finishing, such as:

- The cost of setting up food stalls exceeds the maximum budget defined by `MAX_BUDGET`.
- A customer's required food types must be satisfied by at least one food stall. This is because a customer is only allowed to purchase all of his dishes from at most one food stall. I.e., A customer **cannot** buy one dish from one food stall and then another dish from another food stall.
- There is at least one table.

Once the canteen passes the validation, the method `simulate()` is invoked to start the simulation. Inside the method `simulate()`, the `timer` variable is incremented by 1. If needed, the method preprocess() is called to do any preprocessing. The program then loops through each of the customers in `allCustomers` and invokes the method `takeAction()`. Each customer's `takeAction()` is invoked only once during an iteration. After that, the method `postprocess()` is invoked to perform any post-processing actions, if any. The simulation stops when all the customers are in `doneQueue`.

Customer.takeAction()

Each customer's `takeAction()` is called once during an iteration. A customer's action depends on where the customer is in the canteen and can perform at most one of the following actions during an iteration. This method also returns `true`, if the customer moves from one queue to another queue. Otherwise, returns `false`.

1. At the end of the previous period, if the customer is the first in the wait-to-enter queue, the customer finds the food stall that serves all the required dishes and has the shortest line of customers, then enqueue the food stall's customer line and return `true`. Otherwise, if all the eligible food stalls cannot accept more customers, simply remain in the wait-to-enter queue and return `false`. In the event that two eligible food stalls have equal customers in their queues, then choose either one.
2. At the end of the previous period, if the customer is the first in the customer queue of a food stall and the food stall is available to take an order (i.e. `isWaitingForOrder() == true`), then the customer orders the dishes by calling the food stall's `takeOrder(this.requiredDishes)` method. This `takeAction()` method must return `false` because the customer still remains in the customer queue.

3. At the end of the previous period, if the customer is the first in the customer queue of a food stall and the food stall is cooking his dishes, then do nothing. The time it takes to cook all the customer's required dishes is the sum of the time to cook each dish. For example, cooking a `DEFAULT` customer's required dishes would take 2+1+3+2+1 = 9 time steps. If the food stall has finished cooking (i.e., `isReadyToServe() == true`), then the customer makes the payment by calling the food stall's `takePayment(this.payment)` method. This `takeAction()` method must return `false` because the customer still remains in the customer queue.

4. At the end of the previous period, if the customer is the first in the customer queue of a food stall and the food stall is still processing the payment, then do nothing and return `false`. The time it takes to process the payment is depended on the customer's payment method. If the food stall has finished processing payment (i.e., `isPaid() == true`), then the customer takes the food by calling the food stall's `serve()` method, leaves the food stall's customer queue, and goes to the end of the wait-to-seat queue. At this point, this `takeAction()` method must return `true` because the customer moves to a new queue.

5. At the end of the previous period, if the customer is the first in the wait-to-seat queue, then find an available table (i.e., `table.isFull() == false`), then put himself in the table's seated customers list (i.e., `seatedCustomers`). Then, this method must return `true` because the customer moves to a new queue. Otherwise, if all the tables are currently occupied, the customer remains in the wait-to-seat queue and this method returns `false`.

6. At the end of the previous period, if the customer is sitting at a table and has not started eating yet, then start eating. The time it takes to eat all the dishes is simply the sum of the time to eat each dish. For example, for a `DEFAULT` customer, it takes 6+5+10+5+2 = 28 time steps to finish eating. If the customer is still eating, then continue eating and return `false`. If the customer finishes eating, then move himself to the done queue and return `true`.

Logging:

At the end of each iteration, if both `CanteenICT.VERBOSE` and `CanteenICT.WRITELOG` are set `true`, then for each simulation, two log files will be generated: `xxx_state.log` and `xxx_summary.log` where `xxx` is the canteen name. state.log displays the snapshot printed at the end of each iteration, primarily for your debugging purposes.

1) xxx_state.log:

Below is an example content in a **oneway_state.log** file. *Log Explanation:* In the first line, `@D2-4` means the default customer whose ID is 2, and the current state is 4 (waiting to sit) at the time step 31 (`t=31`). You can see that `D2` is currently in the waiting-to-seat queue. Then, in the next time step (`t=32`), `D2` sits at the table 1, so his/her current state is changed to 5 (sitting). Next, the `D2` customer starts eating at time step 33 (`t=33`) with the state value 6 (eating) and stays at the table queue for 28 periods until done eating. Later at time step 61 `(t=61)`, the customer is done eating and goes to the done queue.

```
@D2-4 retrieves food from Bamboo Shop, and goes to Waiting-to-Seat Queue.
====================== T:31======================
[Waiting-to-Enter Queue]: D7-D8-D9-D10
[Food Stall: Bamboo Shop]: D3-D4-D5-D6
[Waiting-to-Seat Queue]: D2
[Table 1]: D1
[Done Queue]:
```

```
@D2-5 sits at Table 1.
@D3-2 orders from Bamboo Shop, and will need to wait for 9 periods to cook.
@D7-1 queues up at Bamboo Shop, and waiting to order.
====================== T:32======================
[Waiting-to-Enter Queue]: D8-D9-D10
[Food Stall: Bamboo Shop]: D3-D4-D5-D6-D7
[Waiting-to-Seat Queue]:
[Table 1]: D1-D2
[Done Queue]:

@D2-6 eats at the table, and will need 28 periods to eat his/her meal.
====================== T:33======================
[Waiting-to-Enter Queue]: D8-D9-D10
[Food Stall: Bamboo Shop]: D3-D4-D5-D6-D7
[Waiting-to-Seat Queue]:
[Table 1]: D1-D2
[Done Queue]:

. . .

@D2-7 is done eating.
@D4-4 retrieves food from Bamboo Shop, and goes to Waiting-to-Seat Queue.
====================== T:61======================
[Waiting-to-Enter Queue]: D9-D10
[Food Stall: Bamboo Shop]: D5-D6-D7-D8
[Waiting-to-Seat Queue]: D4
[Table 1]: D3
[Done Queue]: D1-D2
```

2) xxx_summary.log:

summary.log contains the summary of the numbers of customers at each state at the end of each iteration. Below is the format:

```
<T=t> <# customers waiting to enter> <# customers waiting in food stalls' queues>
<# customers waiting to be seated> <# of customers sitting at tables>
<# done customers>
```

summary.log is primarily used to automatically compare the simulation results and will be used to grade your simulation implementation's correctness. Below is an example content in a summary.log file:

```
T=31  4     4     1     1     0
T=32  3     5     0     2     0
T=33  3     5     0     2     0

. . .

T=61  2     4     1     1     2
```

Your Tasks:

Task 1: Initialization and Validation

In this task, essentially, you need to implement the following methods. Refer to the provided skeleton code for further instruction.

- Customer → `Customer(CanteenICT _canteen)`: Customer's constructor
- CanteenICT → `public int getInstallCost()`: Compute the total installation cost from building all the food stalls. A food stall's installation cost is the sum of the installation cost of each type of food that it sells.

- CanteenICT → `public boolean validateCanteen()`: Check if the total installation cost exceeds the max budget and for all the conditions that prevent the simulation from finishing.

Executing `StudentTester.testValidation()` should give the following output.

`Canteen test1 does not pass the validation. Installation cost: 80500 baht, while maximum budget is 80000 :(`
 	
`Canteen test2 does not pass the validation. There must be at least one food stall that sell all the dishes requried by each customer.`
 	
`Canteen test3 does not pass the validation. Have you added some tables to your canteen where customers can sit and eat their food?`
 	
`Canteen test4 passes the validation. Allright! Good to go.`

Task 2: Simulation

For this task, you will be implementing the mechanism that simulates each customer's action given the current time step (T=t) and the snapshot at the end of the previous iteration. Essentially, you need to implement the following methods.

- Customer → `public void takeAction()`: The simulator routinely calls this method during every iteration (See `CanteenICT.simulate()`). Your task is to implement appropriate actions to place the customer in the right queue at the end of the iteration.
- CanteenICT → `private void preprocess()`: [OPTIONAL] If you need to perform any preprocessing before each iteration begins (See `CanteenICT.simulate()`), you can implement it here.
- CanteenICT → `private void postprocess()`: [OPTIONAL] If you need to perform any postprocessing after each iteration ends (See `CanteenICT.simulate()`), you can implement it here.

Flexibility Policies:

We understand that there are many OOP and Java features that you would like to explore and use to present your creativity. However, some guidelines must be set to allow fair evaluation while leaving room for flexibility to explore many wonderful Java-based OOP features.

- **Additional variables/methods**: You are free to implement your own additional class variables and methods that facilitate the implementation of your algorithms. However, you must not modify the code inside the "`DO NOT MODIFY`" zones. Furthermore, you cannot modify StudentTester.java since the autograder will also use the same interfaces to interact with your code.
- **Additional classes**: You are also free to add additional classes to facilitate your implementation. Also include additional class implementation files (i.e., YourOwnClass.java) with your submission.
- **No external libraries**: You are not allowed to use third-party Java libraries. Your code should be able to compile and run without having to install external jar files
- **No packages**: The use of packages are not allowed in this project, as it is not compatible with the autograder. Put everything in the default package (i.e., there should not be a package declaration on top of each java file).

- **Result differentiation**: We understand that small details can slightly differ when it comes to actual implementation. Therefore, when we grade your simulation's correctness, a small deviation of ± 10% is OK.

Suggestions:

1. [**Discrete-event simulation (DES)**] The action of a customer during the time T=t is based on the state of the customer *at the end of the previous iteration* (i.e., T=t-1). Therefore, special consideration is needed when checking if a customer is in the front of the queue during an iteration. For example, at the end of T=t-1, the waiting-to-enter queue has [D4][D5][D6]. Then, during T = t, you process D4 by removing it, right after which the queue would become [D5][D6]. Then you process D5. Even if D5 may be the first during T=t, *D5 was not the first in the queue at the end of T=t-1*. In this case, D5 would not be removed from the wait-to-enter queue. There are many ways to handle this. One way would be to create another "shadow" set of queues of what they would look like at the end of the iteration, while processing the current set of queues. Another would be to introduce certain variables that keep track of whether each customer is the front-most in each queue at the end of the previous step.

2. [**Understand the existing code first**] It helps to understand the provided implementation before designing your algorithm. This way, you know what is available for you to use and what needs to be implemented.

3. [**Incremental tests**] Though CanteenICT is a fairly simple simulation software compared to other commercial ones in the market, the small details can overwhelm you during the course of implementation. Our suggestion is to spend some time understanding the requirements from the specs. Then, incrementally implement and test one method/part at a time to see if it behaves as expected. It is discouraged to implement everything up and test it all at once. For example, you can implement and test the code that takes a customer from the wait-to-enter queue and places him at the right food stall first, before moving on to the other simulation parts. Try to break down the simulation into smaller modules, each of which can be easily tested.

4. [**Start early**] Don't wait until the last weekend to start working on this project. Start early, so you have enough time to cope with unforeseen situations. Plus, it can take some time to completely understand the project's specs.

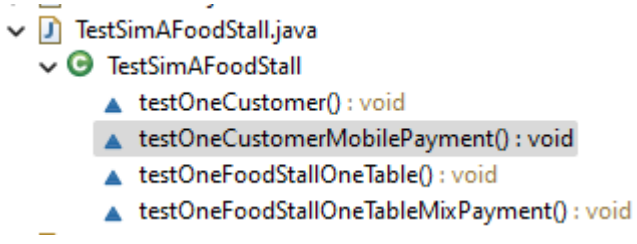
Testcases:

There are two ways to test your program

1) **Tester class**: Testcases are provided in **StudentTester.java**. The xxx_state.log files are provided as part of the package to help you to debug your code. The xxx_summary.log files are provided to help you to validate your output with the expected output. You must run the method in the StudentTester class one-by-one to generate those two files for each case including *oneway*, *debug*, *simple*, *congestion*, *large,* and *bonus* (for bonus - optional).

2) **JUnit5 test:** A JUnit test is a method which is only designed for testing purposes. Assertion functions are used to validate the actual with expected output. There are many test files whose names start with **`TestXxxx.java`** provided in the test folder. You **MUST** copy and paste them into the `src` folder before running. DO NOT run them inside the test folder. You may also have to import JUnit lib into your project. My suggestion is to run the files in the following order to test from a small set of code and basic cases first.

a. TestCanteen, // test customer, food stall, and basic canteen setup and validation
b. TestSimAFoodStall, // test simulation for one food stall case
c. TestSimBasic, // test simulation for a basic case, including congestion case
d. TestSimLarge, // test simulation for a large number of customers (several sec.)
e. TestBonus, and TestBonusLarge // for bonus (optional)



Note: Besides, running all methods in the test file as once, you can choose to run on a specific test method. First, go to the test file in the *Project Explorer* panel, drill down on the file name until you see the method name, right-click on the method name and select *Run As -> JUnit Test*

Submission Instruction:

It is important that you follow the submission instructions. Failing to do so may result in a deduction and/or delay of your scores.

1. Run `testCongestion()` and `testLarge()`, and collect `large_state.log`, `large_summary.log`, `congestion_state.log`, and `congestion_summary.log`.
2. Prepare the following files to submit (* note that the filenames must be exactly the same as shown here. Otherwise, the autograder will not recognize the files.):

   a. `CanteenICT.java`
   b. `Customer.java`
   c. `FoodStall.java`
   d. `Table.java`
   e. `large_state.log`
   f. `large_summary.log`
   g. `congestion_state.log`
   h. `congestion_summary.log`

3. Make sure to put your name, ID, and section in a comment on top of each of the .java files.
4. Submit the above files MyCourses before the deadline.
5. Redownload the submitted files and make sure they are what you want to submit.

Note: Late submission will suffer a penalty of 20% deduction of the actual scores for each late day. You can keep resubmitting your solutions, but only the latest version will be graded.

Coding Style Guideline:

It is important that you strictly follow this coding style guideline to help us with the grading and for your own benefit when working in groups in future courses. Failing to follow these guidelines may result in a deduction of your project scores.

1. Write your name, student ID, and section (in commented lines) on top of every ***Java*** code file that you submit. Do not write your name on the log files.
2. Comment your code, especially where you believe other people will have trouble understanding, such as each variable's purposes, loop, and a chunk of code. If you implement a new method, make sure to put an overview comment at the beginning of each method that explains 1) Objective, 2) Inputs (if any), and 3) Output (if any).

[Optional] Special Challenge (Bonus 10 Points): But wait a minute….not all people eat the same things?

This bonus will count towards this project's score. For example, if you receive 90% for the main part of this project, the 10 points bonus will be applied to close the gap. Note that the total score of this project does not exceed 100%. For example, if your total score (regular + bonus) comes out 105%, you will earn 100%.

Challenge Task1:

Use the inheritance mechanism that you just learned in class to implement the following classes by **extending** from `Customer` with the following required dishes.

Class	NOODLES	DESSERT	MEAT	SALAD	BEVERAGE	Note
Student		5				Students eat a lot of sweet things to energize their brains.
Professor	1				1	Professors are poor, so they can only afford to eat instant noodles. They also need water because they use a lot of voice.
Athlete			3	1	1	Athletes need protein. A lot of protein from meat and protein shake. And some veggies.

Challenge Task2:

Implement class `ICTStudent` by **extending** from `Student`. An ICT student eats the same things as other students. The only difference is that ICT students are very busy. They can start eating right after they leave the food stall to get to classes in time. That is, an ICT student can eat while waiting in the wait-to-seat queue.

You may also need to modify `CanteenICT.setCustomers()` to also support these newly created classes. To submit the bonus challenge, run `testBonus()` and submit additional Java files along with `bonus_state.log` and `bonus_summary.log` with your submission.

Bug Report:

Though not likely, it is possible that our solutions may contain bugs. In this context, a bug is not an insect but an error in the solution code that we implemented to generate the test cases. Hence, if you believe that your implementation is correct, yet yields different results, please contact us immediately so proper actions can be taken.

Need help with the project?

If you have questions about the project, please first ask them on the class' general channel on Teams, so that other students with similar questions can benefit from the discussions. The TAs can also answer some of the questions and help to debug trivial errors. If you still have questions or concerns, please make an appointment with one of the instructors. ***We do not debug your code via email or messages***. If you need help with debugging your code, please come see us.

Academic Integrity (Very Important)

Do not get bored with these warnings yet. But please, please ***do your own work***. Your survival in the subsequent courses and the ability to get desirable jobs (once you graduate) depend heavily on the skills you harvest in this course. Though students are allowed and encouraged to discuss ideas with others, the actual solutions must be originated and written by themselves. Collaboration in writing solutions is not allowed, as it would be unfair to other students. Students who know how to obtain the solutions are encouraged to help others by guiding them and teaching them the core material needed to complete the project, rather than giving away the solutions. ***You can't keep helping your friends forever, so you would do them a favor by allowing them to be better problem solvers and life-long learners.*** Your code will be compared with other students (both current and previous course takers) and online sources using state-of-the-art source-code similarity detection algorithms, which have been proven to be quite accurate. If you are caught cheating, serious actions will be taken, and heavy penalties will be bestowed on all involved parties, including inappropriate help givers, receivers, and middlemen.