CSCI165 Computer Science II
Lab Assignment: Food Truck

Objectives

- Define simple class specifications and UML association diagrams
- Implement encapsulation and privacy protection
- Define conventional toString and equals methods
- Implement simple domain validation
- Define classes to pass an existing unit test framework
- Define unit test frameworks to test existing classes
- Build a small application to demonstrate object interaction/association/aggregation in a meaningful way
- Read instructions and associated documents carefully before beginning the coding process.

Scenario: You have been asked to create some basic objects to manage remote orders for a Food Truck. Using an app, customers will place an order prior to arriving at the food truck. When their order is ready, the app will send a text message to the customer's phone. Weekly emails will be sent to repeat customers alerting them of where the truck will be located. This phase of the project is just to create the objects to store the different attributes and to perform some simple **proof of concept** testing.

After consulting with your team you have decided upon two basic classes.

- **MenuItem**: This class will encapsulate the properties for items that are being sold
- **Customer:** This class will encapsulate the properties for the food truck's customers. Essentially, they just want to track names, email addresses and cell phone numbers so they can send out notifications on where the truck will be located during the week and alert customers when their orders are ready.
- **Date:** An object to store the date of an order

Your team leader wants to ensure that your basic class specifications meet the client's needs, so the first step in the process will be to create UML diagrams. You will then use these diagrams as a communication medium between your team and the client; for verification purposes.

Since the Date class was created for an earlier project, you will just be reusing it here. It has been provided for you. Just drop into your new lab folder

I recommend **https://app.diagrams.net/** for your diagramming needs

Testing: To ensure that your class designs meet the criteria, your team has decided to implement some unit testing. You all agree to experiment with the Java unit testing framework ***JUnit*** and VS Code's unit testing extension ***JavaTestRunner***.

- Your project lead (a big proponent of **Test Driven Development)** has given you a unit test file for the MenuItem class: ***MenuItemTests.java***
- Your implementation of the MenuItem class must pass all of those tests. The tests have been ***designed first*** and cannot be modified unless you have verifiable evidence that a test is incorrect (It has been known to happen, as it did just last week :))***. Any existing test modification must be cleared with the project lead (your instructor) before***

moving forward.
https://en.wikipedia.org/wiki/Test-driven_development

- The project lead realized that they did not include a test for the compareTo method for the MenuItem class so you have been assigned to define one of these. Details of this method follow.
- You are responsible for developing a series of unit tests for the provided Customer class. All set methods ***containing validation*** should be ***thoroughly*** tested. The equals and compareTo methods must also be tested. If you are unsure what constitutes a thorough test then let's talk.

TASK ONE: Create the MenuItem class

This class will define the basic attributes and behaviors for an item on the food truck's menu. It must pass all provided unit tests.

Include the following attributes (instance variables):

- Name: String
- Price: double
- Calories: int

Include the following behaviors (instance methods):

- Define an appropriate collection of getters and setters for the attributes
- Define a toString and equals method
- Define a compareTo method that orders (descending) based on the calories of an item. So if itemA has fewer calories then itemB then itemA would be considered as "coming before"

Domain Validation:

- When defining the set method for the price attribute include code to ensure that the price is greater than zero. Default value should be $1.00
- When defining the set method for the calories attribute include code to ensure that the value is greater than or equal to zero. Default value should be 0

Define constructors

- No argument constructor that simply uses default values.
- Overloaded constructor to accept Name, Price and Calories. Be sure to pass the constructor input through the domain validation in the set methods. Don't repeat yourself!

While you are building this class, ensure that your methods pass the provided unit tests.

TASK TWO: Test the Customer Class

The Food Truck app wishes to track some basic info on the customers. This class will define the basic attributes and behaviors of the Food Truck's customers. After close consultation with the client we have decided on a simple set of attributes: name, phone and email. The phone will be used to text the customer when their order is ready and the email will be used for the receipt.

Perhaps more will be added later.

NOTE: One of your team mates built this class but did not write any tests or create a UML class diagram. This task has been assigned to you. Design a UML class diagram and add it to the association/interaction diagram that was provided in the repo.

You have been assigned to write thorough tests for each of the "set" methods that include any validation. You **may** find during the testing that there are problems with the logic in the provided class. Be sure to **fully understand** what the logic is attempting to determine before writing tests. Come up with a few test cases on paper first. This methodology is key to writing unit tests. You problem solve this outside of the code by specifying sample inputs and determining what the outputs **should be.** Only then can you enshrine this in code.

While testing, if you find any semantic errors in the code, they should be documented and your solution explained. These descriptions should be included as additional line comments right where the problem exists in the source code file.

What follows here are meeting minutes from conversations with the client on the Customer data.

Include the following attributes

- name: String
- email: String
- phone: String

Include the following behaviors:

- Define an appropriate collection of getters and setters for the attributes
- Define a toString and equals method

Domain Validation:

- When defining the set method for the email attribute include code to ensure that the email is in a valid format based on the following needs. (NOTE: These are not 100% thorough. Don't worry about it)

    - **Email Validation**
        - **The @ symbol:** The symbol separating the email prefix from the domain
            - @ cannot be the first or last character
            - @ cannot be present in the top level domain (to the right of the last **.**)
            - String methods will help you with this.

- **Prefix:** The characters appearing before the @
  - The maximum upper limit of characters for a prefix is up to 64 in length. It can be a combination of any alphabetic characters, numeric values or any of the special character mentioned below
    - # ! % $ ‘ & + * – / = ? ^ _`. { | } ~
  - The period character (“.”) is valid for the prefix part with the following restrictions:
    - It is not the first or last character
    - There cannot be two or more consecutive periods anywhere in the email

- **Phone Validation**
  - The phone number must be 10 numeric digits or it should be listed as “unknown”

Define constructors

- No argument constructor
- Overloaded constructor to accept Name, email and cell phone number. Be sure to pass the constructor input through the domain validation in the set methods. Don’t repeat yourself!

Task Three: Define a compareTo method that orders (ascending) based on the Customer’s last name. So if customerA’s last name is alphabetically before customerB’s last name, then customerA is considered to come before.

You’ll need to consider how to deal with the name being in a single variable. Use the String class’ compareTo method as part of this definition. This method is part of the ***Comparable*** design pattern if anyone is interested in reading about it. Much more on this later.

Application

Once your testing has been finished and you are certain that your classes are well designed and secure, you can build a simple proof of concept application.

Define a **Driver** class. This class will serve as the **application context**. As part of your application development you’ve decided on testing the basic flow with a bit of random data. You have been provided with the file: **products.txt.**

This file contains a collection of data that maps to your **MenuItem** object: {name, price and calories}

```
Fresh Tomatoe,4.35,206
Lettuce - Arugula,6.25,979
Muffin Mix - Corn Harvest,6.2,428
Apples - Sliced / Wedge,6.09,419
Freedom Fries,3.57,448
Chocolate Chip Cookie,6.45,1000
Beer,9.0,635
Beef Patty,3.79,943
Apple - Royal Gala,8.75,248
Pork Sandwich,6.34,207
```

APPLICATION TASK ONE: Your first task for the driver is to load the data from the file into an appropriate data structure. You should be verifying your code after each step.

1. Create the following method in your driver file

   public static ArrayList<MenuItem> loadMenuItems(String fileName)

   Something to think about: Why are Driver methods defined as static when instance methods are not?

2. Create the following method in your driver file

   public static void printMenu(ArrayList<MenuItem>)

   This method will accept the list of menu items and print them to the screen in a nicely formatted way. ***Make the presentation look neat.*** Include a descriptive header above the items like **Food Truck Menu**

3. The Driver logic should follow these sequential operations

   1. Call **loadMenuItems**
   2. Call **printMenu**
   3. Create an instance of the provided Date class to store the date of the order. Populate the Date instance with the current system date. You ***must*** use the provided Date class for this. Consider it practice using objects that aren't part of the Java API.
   4. Prompt the user for a customer's information: name, email and cell number. Create a Customer instance from this data. Don't worry about dealing with invalid customer data. Just let the Customer class handle it and it is perfectly ok if some of the values are "unknown". I realize this isn't a great "real world" practice, but that isn't the point here. We will be covering better ways of dealing with these types of situations.

      If you have the time and desire and want to make this more robust, please do. For those of you that wish to use custom exception handling to trigger messages, go for it.

5. Allow the customer to choose a couple of items from the menu. **You design this sub-system.** Maybe use an additional list of menu items as a shopping cart. This is a good candidate for an additional method.
6. Decompose and define as many helper methods as you feel you need.
7. Print a neat receipt to the console and to a file. The receipt should contain summary information and 6% tax. Also show the total calories for the order. Design your own solution for naming the file something descriptive and unique. Here is an example of console and file output. Can you generalize this to use in both situations without modification?

```
Customer: Jim Lahey

Invoice Number: JILA3620231110
Date: 3/6/2023
Time:11:10:28

Item                          Quant   Price   Total
================================================
Fermented Ostrich Lips        3       $3.99   $11.97
Freedom Fries                 3       $2.99   $8.97
Jim Jones Juice               3       $2.50   $7.50

Calories: 2726
================================================
Subtotal              $28.44
6.25% sales tax       $1.77
Order Total           $30.21
```

Submission: Submit all relevant files. Review the instructions to ensure you have met all requirements.

Rubric

Requirement	Points
Correctness: Classes meet all requirements and functions as described in the instructions.	10
Application: Driver logic functions and is intuitive. All requirements are met	5
Javadoc: methods are documented using appropriate Javadoc style comments	3
UML: Diagram is complete and correct	2
Unit Tests: Customer tests are complete and robust. Logic errors are found and fixed and all tests pass. MenuItem class passes all provided tests	5