CSCI165 Computer Science II
Lab Assignment
OOD: Composition (Food Truck Redux)

Objectives:

- Work with the **Composition/Aggregation/Association** design patterns
- Demonstrate how to work with encapsulated objects
- Make appropriate privacy decisions
- Protect privacy of internal objects when necessary
- Define and use copy constructors for proper cloning
- Define object ordering using compareTo

Scenario: You and your team continue the development of the FoodTruck application. Now that you have the basic classes for Customer and MenuItem, you need to develop two additional classes: ***The OrderItem class and the Order class***.

The purpose of the **OrderItem** class is to encapsulate a **Customer's** choice of **MenuItem** and the quantity of that MenuItem.

The purpose of the **Order** class is to encapsulate a single **Customer**'s order of multiple **OrderItems** and to create a record of the transaction.

You know that you have to utilize the ***composition design pattern*** (https://en.wikipedia.org/wiki/Object_composition) to make these classes work together and your project lead has made a big fuss about protecting privacy, so that is first and foremost in your mind. Also understand that there may be situations where you want things to be publicly available. This is all part of the design of an application.

Task One: Create the OrderItem class

This class associates a MenuItem with a quantity when an order is being taken.

- **Properties**
  - **MenuItem:** The item chosen as part of the ordering process
  - **Quantity:** The quantity of MenuItems on a particular order

- **Behaviors**
  - Getters for MenuItem and quantity. Protect privacy where appropriate. The team has decided to not provide setter methods for MenuItem and quantity. This functionality will be provided by the constructor. **Do not include a setter for these fields.**
  - **void updateQuantity(int):** Increases or decreases the quantity of a particular MenuItem by accepting a positive or negative value.
    - **Validation:** All positive numbers are accepted. Negative numbers are accepted as long the absolute value is not greater than the existing quantity. If this is the case, leave the quantity unchanged.

    - **String toString():** returns a nicely formatted String of an OrderItem’s state.
        - **Composition/Aggregation Task:** Use the toString method from the MenuItem class and concatenate the quantity.
    - **boolean equals(OrderItem):** performs a deep comparison on OrderItems.
        - **Composition/Aggregation Task:** Use the equals method of the MenuItem class as part of the logic.
    - Once a MenuItem has been created, only the quantity can be changed or the item can be removed completely from the order (but these are Order class behaviors).

        - You can set the quantity explicitly through the constructor
            - A customer says “Give me 3 fries”
            - **MenuItem item   = new MenuItem(“Freedom Fries”, 2.99, 448);**
            - **OrderItem oItem = new OrderItem(item, 3);**
            - add the item to the cart: **cart.add(oItem);**

        - You can increase the quantity of a previously ordered item
            - A customer says “Let me get 2 more fries”
            - Find the item in the cart and update the quantity
            - **oItem.updateQuantity(2);**

        - You can decrease the quantity of a previously order item
            - A customer says “Forget those last two fries”
            - Find the item in the cart and update the quantity
            - **oItem.updateQuantity(-2);**

- **Constructors**
    - Overloaded to accept the **MenuItem** and the quantity
    - Copy constructor for privacy protection. When defining copy constructors for classes that use composition/aggregation you also need to invoke the copy constructors of all mutable objects that are part of the class you are copying.
    - **Privacy Protection with Composition:** You will need to define a copy constructor for the MenuItem and Customer classes in order to protect privacy in OrderItem class. The provided unit tests expect this behavior.

- **Validation:** Add some common sense validation where appropriate
- **Javadoc:** Everything must be documented according to javadoc standards
- **JUnit:** You have been provided with a small collection of unit tests for this class in the file: **OrderItemTests.java**. Ensure that your class passes all of these tests. They will be used to assess your work. You are not permitted to modify the tests unless there is an approval. I will use the original unit test file to grade.

Task Two: Create the Order class

- **Properties**
  - **Date:** The date the order was placed. For composition practice, use the custom Date class we have been using and developing.
  - **InvoiceID:** A string of characters created by the application to identify an order. Ultimately you know want to guarantee 100% unique IDs but we will not worry about that. This can be reused from the previous Food Truck lab
  - **Customer:** The customer making the order. Reuse the Customer class from the previous Food Truck lab
  - **Cart:** An ArrayList of OrderItems. Each order will have a collection of OrderItems which will contain the MenuItem and the quantity. Only one OrderItem per MenuItem ordered should be added to the cart. Increase quantities instead. ***Big composition task here!***

- **Behaviors**
  - Get methods for Date, InvoiceID, Customer and Cart. **Protect Privacy!**
  - **void addItem(MenuItem**): Adds a quantity of one single item to the **cart**. If the item already exists in the "cart" simply increase the quantity.
  - **void addItem(MenuItem, int):** Overloaded method **a**dds a specified quantity of an item to the **cart**. If the item already exists in the "cart" simply increase the quantity.
    - These addItem methods will need to work with the methods of the OrderItem class when appropriate.
  - **OrderItem removeItem(MenuItem):** Removes and returns specified item from the order
  - **double calculateTotal( ):** Will sum the ***price*** * ***quantity*** of the items in "this" order by iterating through the **cart**
  - **double calculateTax(double total):** will calculate and return the tax amount of the order
  - **void writeToFile(void):** Will write "this" order to a file, creating a receipt. The file name should be the **invoiceID** plus **.txt**
  - Private helper method **setDate():** This method should get the current Date and add the month, day and year into your custom Date class. The Java API class **LocalDateTime** is perfect for this.

    https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/time/LocalDateTime.html

  - Private helper method **createInvoiceID():** The invoice number will be composed based on the following rules

    - Take the first two initials of the customer's first name, converted to uppercase
    - Take the first two initials of the customer's last name, converted to uppercase
    - Take the Unicode value of the first character of the first name
    - Take the Unicode value of the first character of the last name

        - Add them together and multiply by the length of the full name
        - Concatenate this value to the initials described above
        - Concatenate the day, month and year with no punctuation

            - **Example:** Jim Lahey 1/27/2020
            - **Initials:** JILA
            - (J = 74) + (L = 76) => 150
            - Length of "Jim Lahey" => 9
            - 150 * 9 => 1350
            - **Invoice Number:** JILA13501272020
            - It should go without saying that this process should work for any name, any date and any time. Don't worry about the chances of a duplicate ID, ultimately our database system would handle that.

    - **String toString( ):** returns a nicely formatted string showing the order details. Show tax and the final order total. This is your receipt formatting and should be used to write to a file and display in the terminal window.
    - **boolean equals(Order otherOrder):** Performs a deep comparison on two orders. Call all the equals methods of the internal objects. This is how composition deep equality checks work.
    - **int compareTo(Order otherOrder):** Specifies ordering of Order objects. The criteria should be based on the ascending Date of the order.
      Hint: Simply use the **compareTo** from the Date class. Nice composition task here too.

- **Constructors**
    - Overloaded to accept the Customer
    - Copy constructor. Be sure to utilize the copy constructors of your embedded classes when appropriate to prevent a shallow copy.
    - Add more if you feel it's appropriate
    - The constructors should call **setDate**

- **UML Diagram**
    - You have been provided with a partially complete UML diagram. Expand the diagram for the application by finishing the Order class. For this exercise let's say that the Order Class **owns** the OrderItem class. Show this composition using UML composition notation. You do not need to duplicate the labels that provide descriptions of aggregation, composition and association.

- **JUnit Tests**
    - Develop robust unit tests for the following methods in the Order class. These methods are the ones that could contain privacy leaks. Prove that each object returned from these methods is a valid clone.

NOTE: DO NOT USE THE CLONE METHOD. This method does not do a deep copy of class elements. We will cover this in detail in lessons on polymorphism. You should be defining explicit copy constructors.

A clone test consists of the following

- **Identity (reference) Check:** original != clone
- **State Check:** original.equals(clone) == true
- **Order class**
    - getCustomer, getDate, getCart
    - For practice and to develop a good understanding of cloning objects, write the **cart cloning** logic yourself. Iterate through the cart and clone each OrderItem.
    - **Research:** Does Java provide a privacy safe way of cloning ArrayList objects? Include a description of your research as code comments. Cite your sources.

- Your classes will be run through a thorough collection of unit tests to ensure that you have met 100% of the requirements for this lab. ***Review the instructions carefully.***

- **Application**
    - Either modify your existing Food Truck Driver or create a new Driver to prove that your objects work well together. Allow for the processing of **multiple** Orders. Write the receipts to a file.
    - The menu will be retrieved from a file as with the previous lab.
    - You do not need to prove any privacy protections in the Driver, those will all be proven via unit tests.

RUBRIC ON FOLLOWING PAGE

Rubric

Requirement	Points
Correctness: Classes meet all requirements and functions as described in the instructions. Classes pass all unit tests based on requirements.	10
Application: Driver logic functions and is intuitive. All requirements are met	10
Javadoc: methods are documented using appropriate Javadoc style comments	3
UML: Diagram is complete and correct	2
Unit Tests: Unit tests correctly prove privacy protection	5