DUE DATE: OCTOBER 7, 2020 AT 11:59PM

Instructions:

- You must complete the "**Blanket Honesty Declaration**" checklist on the course website before you can submit any assignment.
- Only submit the **java files**. Do ***not*** submit any other files, unless otherwise instructed.
- To submit the assignment, upload the specified files to the **Assignment 1** folder on the course website.
- Assignments must follow the **programming standards** document published on UMLearn. You will lose marks for not following these standards.
- After the due date and time, assignments may be submitted but will be subject to a late penalty. Please see the ROASS document published on UMLearn for the course policy for late submissions.
- If you make multiple submissions, only the **most recent version** will be marked.
- These assignments are your chance to learn the material for the exams. Code your assignments independently. We use software to compare all submitted assignments to each other, and **pursue academic dishonesty vigorously**.
- Your Java programs must compile and run upon download, without requiring any modifications.
- Automated tests will be used to grade the output of your assignment. If you do not follow precisely the guidelines detailed below, these automated tests can fail and you will lose marks. In particular, make sure that **your code produces exactly the example outputs** (including correct spacing) shown in these guidelines.
- ArrayLists are not allowed in this assignment. You must use arrays and partially filled arrays, as described.

Assignment overview

In this assignment, you will implement a set of related classes of objects:

- **`Customer`**: A customer who orders food
- **`FoodItem`**: A specification for a food item that appears on a menu
- **`Restaurant`**: A restaurant that has a menu
- **`FoodApp`**: A food delivery app, used to order food items from restaurants
- **`Order`**: A class that contains all the information about a specific order

Keep all of your methods short and simple. In a properly-written object-oriented program, the work is distributed among many small methods, which call each other. Make sure to call methods already created when appropriate, instead of duplicating code for no reason. There are no methods in this entire assignment that should require more than 10 lines of code, not counting comments, blank lines, or {} lines (and many of them need no more than 3-4).

Unless specified otherwise, all instance variables must be **`private`**, and all methods should be **`public`**. Also, unless specified otherwise, there should be ***no*** **println** statements in your classes. Objects usually do not print anything, they only return **String** values from methods.

Testing Your Coding

We have provided sample test files for each phase to help you see if your code is working according to the assignment specifications. These files are *starting points* for testing your code. Part of developing your skills as a programmer is to think through additional important test cases and to write your own code to test these cases. *For marking, we will use longer and more comprehensive tests*.

Phase 1: Customer and FoodItem classes

First, implement two simple classes: a **`Customer`** class and a **`FoodItem`** class

The **`Customer`** class should have:

- Three instance variables: the customer's name (a **`String`**), email address (a **`String`**) and street address (a **`String`**)
- A constructor that has three parameters, in the above order, which are used to initialize the three instance variables.
- A standard **`toString`** method, which returns a **`String`** containing the customer name, email address and street address (as shown in the example output below).
- An **equals** method that checks if two customers are equal. Two customers are equal if they have the same email address, since an email address should belong to only 1 customer.

The **`FoodItem`** class should have:

- Four instance variables: a name (a **`String`**), a cost (a **`double`**), a selling price (a **`double`**), and the number of items available in stock for selling (an **`int`**)
- A constructor that takes four parameters, in the above order, which are used to initialize the four instance variables.
- A method **isAvailable** that checks if there are items available for selling (returns false is no items are available, true otherwise)
- A **`toString`** method which returns a **`String`** containing the name of the food item and the selling price. If the item is not available, add "(SOLD OUT)" next to the price, making use of the isAvailable method. Follow the format shown in the example output below.
- A method **setSellingPrice** that takes a new price as a parameter, and updates the selling price instance variable.
- A method **decrementStock** that decrements the number of items in stock by 1.
- A method **increaseStock** that takes an amount of additional stock as a parameter, and adds it to the existing stock available for selling.

You can test your classes using the supplied test program **`TestPhase1.java`**. You should get the output shown below.

```
-Customer Ellen Ripley
Email: ellrip@aol.com
Address: 245 Alien road

-Customer John McClane
Email: johncowboy@hotmail.com
Address: 560 Yippee-ki-yay boulevard

Both customers have different email addresses

- Spaget
$10.75
- Pristine deathclaw egg
$14.99 (SOLD OUT)

- Pristine deathclaw egg
$15.59
```

Phase 2: Restaurant and FoodApp classes

Next, implement the **`Restaurant`** class, and the **`FoodApp`** class.

The **`Restaurant`** class should have:

- Three instance variables: the name of the restaurant (a **`String`**), a menu (stored as an array of **`FoodItem`**), and a profit (a **`double`**).
- A constructor that accepts two parameters, the name and an array of FoodItems, and initializes the corresponding instances variables. The profit shall be initialized to 0.
- A **`toString`** method that returns a **`String`** containing the name of the restaurant, the profit and the menu on different lines (formatting it exactly as shown in the example below).

Then, implement a **`FoodApp`** class. This class should have:

- A class constant: the maximum number of customers a FoodApp can have (an **`int`**). Set it to 100.

- Four instance variables: the name of the FoodApp (a **`String`**), the fees percentage charged to restaurants (a **`double`**), an array of Customers, and the current number of customers in the array (an **`int`**)
- A constructor that takes two parameters, the name and the fees percentage, and initializes the corresponding instances variables. The array of customers will be a partially filled array. It should be initialized according to the maximum number of customers, and be empty at the beginning. The current number of customers in the array should be initialized accordingly.
- A method **`getCustomerIndex`**(**`Customer`**) that looks for and returns the index of the customer given as a parameter in the array of customers, using the equals method of the Customer class. The method must return -1 if the customer is not found in the array.
- An **`addCustomer(Customer)`** method that adds the Customer to the customer array, only if the customer is not already present, and returns true if the customer was added. You should make use of the getCustomerIndex method here. If there is already a customer with the same email address, the method should simply return false. You can assume that there will be enough space to add the customer otherwise (no checks required for space). Also, don't forget to update the counter of the current number of customers.
- A **`removeCustomer(Customer)`** method that removes the specified customer from the customer array, only if it is found (following again the equals rule of the Customer class). The method returns true if a customer is removed, and false otherwise. When removing a customer, you must fill in the gap by shifting up all the following customers. Also, don't forget to update the counter of the current number of customers.
- A **`toString`** method that returns a **`String`** containing the name of the FoodApp, the number of customers and the list of those customers (formatting it exactly as shown in the example below).

You can test your class with **`TestPhase2.java`**. You should get the output shown below.

```
-- Restaurant Burger Queen
Profit = 0.0
Menu:
- Queen wooper
$8.99
- Super extra steakhouse triple queen wooper with cheese
$11.99
- Poutine
$4.99
- Fries
$2.99

-- Restaurant City Sushi
Profit = 0.0
Menu:
- City maki
$4.99
- City roll
$6.99
- City sashimi
$11.99
- City nigiri
$3.99
- City teriyaki chicken
$9.99 (SOLD OUT)

-- FoodApp AvoidThePlates
2 customer(s) registered:
-Customer Ellen Ripley
Email: ellrip@aol.com
Address: 245 Alien road
-Customer John McClane
Email: johncowboy@hotmail.com
Address: 560 Yippee-ki-yay boulevard

This email address already exists!
```

```
Customer removed.
-- FoodApp AvoidThePlates
1 customer(s) registered:
-Customer Ellen Ripley
Email: ellrip@aol.com
Address: 245 Alien road

Customer added
-- FoodApp AvoidThePlates
2 customer(s) registered:
-Customer Ellen Ripley
Email: ellrip@aol.com
Address: 245 Alien road
-Customer John Copy
Email: johncowboy@hotmail.com
Address: 560 Yippee-ki-yay boulevard
```

Phase 3: Order class

Next, implement the **`Order`** class.

The **`Order`** class should have:

- Seven instance variables: the order number (an **`int`**), the Customer who made the order, the Restaurant that receives the order, the FoodApp through which the order was placed, a list of food items ordered (stored as an array of **`FoodItem`**), the total number of items ordered (an **`int`**), and the total price of the order (a **`double`**).
- A class constant to set the maximum number of items that can be ordered (an **`int`**). Set it to 10.
- A constructor that accepts three parameters, the Customer, the Restaurant and the FoodApp, and initializes the corresponding instances variables. The array of FoodItems will be a partially filled array. It should be initialized according to the maximum number of items that can be ordered, and be empty at the beginning. The total number of items ordered should be initialized accordingly. The total price shall be initialized to 0. The order number shall have 6 digits, and start with a 9. Every order should have a distinct order number, starting from 900001, then 900002, then 900003 and so on. You will need to add either an instance variable or a class variable to accomplish this (choose wisely).
- An **`addToOrder(FoodItem)`** method that adds the FoodItem received as a parameter to the FoodItem array, only if it is available (*i.e.* not sold out) and if there is space left in the array. If the FoodItem was added, the method returns true (it returns false otherwise). You can assume that the FoodItem belongs to the restaurant's menu (no need to check if it's on the menu). Don't forget to update here: the amount in stock for the FoodItem (decrement by 1), the total price of the order, and the total number of items ordered.
- A **`toString`** method that returns a **`String`** containing the FoodApp name, the order #, the customer name, the Restaurant name, the list of items ordered and the total price (formatting it exactly as shown in the example below). You will need to add some accessors in the previous classes (aka get methods) to get only the name of the Customer, FoodApp and Restaurant, instead of the full String representation of those.

You can test your class with TestPhase3.java. You should get the output shown below:

```
-- AvoidThePlates order #900001
For customer Ellen Ripley with restaurant Burger Queen
Total price: $0.0

-- AvoidThePlates order #900001
For customer Ellen Ripley with restaurant Burger Queen
- Super extra steakhouse triple queen wooper with cheese
$11.99
Total price: $11.99

-- AvoidThePlates order #900001
For customer Ellen Ripley with restaurant Burger Queen
- Super extra steakhouse triple queen wooper with cheese
$11.99
```

```
- Fries
$2.99
Total price: $14.98

-- AvoidThePlates order #900001
For customer Ellen Ripley with restaurant Burger Queen
- Super extra steakhouse triple queen wooper with cheese
$11.99 (SOLD OUT)
- Fries
$2.99
- Super extra steakhouse triple queen wooper with cheese
$11.99 (SOLD OUT)
Total price: $26.97

Could not add this item to the order: - Super extra steakhouse triple queen wooper with cheese
$11.99 (SOLD OUT)

-- AvoidThePlates order #900001
For customer Ellen Ripley with restaurant Burger Queen
- Super extra steakhouse triple queen wooper with cheese
$11.99 (SOLD OUT)
- Fries
$2.99
- Super extra steakhouse triple queen wooper with cheese
$11.99 (SOLD OUT)
Total price: $26.97

-- AvoidThePlates order #900001
For customer Ellen Ripley with restaurant Burger Queen
- Super extra steakhouse triple queen wooper with cheese
$11.99 (SOLD OUT)
- Fries
$2.99
- Super extra steakhouse triple queen wooper with cheese
$11.99 (SOLD OUT)
- Poutine
$4.99
Total price: $31.96

-- SuperConsume order #900002
For customer John McClane with restaurant City Sushi
- City roll
$6.99
- City roll
$6.99
- City roll
$6.99
- City roll
$6.99
- City roll
$6.99
- City roll
$6.99
- City roll
$6.99
- City roll
$6.99
- City roll
$6.99
- City roll
$6.99
Total price: $69.9
```

Phase 4: Adding some interactions between classes

In this phase, we will go back into previously defined classes and add a bit more functionality.

The **`FoodItem`** class should be updated with:

- A **`getMarkup`** method that returns the difference between the selling price and the cost of the FoodItem.

The **`FoodApp`** class should be updated with:

- An accessor (get method) for the fees percentage.

The **`Order`** class should be updated with:

- A **`submitOrder`** method that will simply call the fillOrder and the updateRecentOrders in the Restaurant and Customer classes respectively with the appropriate parameters (see details below).

The **`Restaurant`** class should be updated with:

- A **`fillOrder`** method that takes four parameters: the total price of the order, the array of FoodItems for the order, the number of FoodItems in the array, and the fees percentage taken by the FoodApp used for this order. This method updates the profit instance variable of the restaurant using the parameters representing the order. The profit on the order corresponds to the sum of markups on all the FoodItems minus the fees percentage paid to the FoodApp on the total price of the order. You can assume that the FoodItems array will not be empty.

The **`Customer`** class should be updated with:

- A new instance variable, which is a partially filled array of Orders, and stores only the 3 most recent orders. The size of the array must be 3, and you can define a constant for that similarly to what we have done earlier.
- Adjust the constructor to initialize the array of Orders. It should be empty at the beginning.
- A **`updateRecentOrders(Order)`** which takes a new Order as a parameter, and adds it to the array of most recent orders. The rank of the orders should always be preserved, in such a way that we can print them from most recent to oldest. When the array is already full, update the array so that the oldest order gets removed and the most recent one gets added in, conserving the full ordering. This method should be very short, if you do it in more than 4-5 lines of code, rethink your approach.
- A **`getStringRecentOrders()`** which returns a String representation of the most recent orders, preceded by "Recent orders of customer [Customer name]". Follow the formatting shown in the example below exactly. The orders must be ordered from most recent to oldest.

You can test your class with TestPhase4.java. Don't worry about the rounding errors that come with calculations with doubles for the profit (as long as you get the first two decimals correctly, it's ok). You should get the output shown below:

```
-- AvoidThePlates order #900001
For customer Ellen Ripley with restaurant Burger Queen
- Super extra steakhouse triple queen wooper with cheese
$11.99
- Fries
$2.99
- Queen wooper
$8.99
- Poutine
$4.99
Total price: $28.96

After submitting the order:
-- Restaurant Burger Queen
Profit = 6.870000000000001
Menu:
- Queen wooper
$8.99
- Super extra steakhouse triple queen wooper with cheese
```

```
$11.99
- Poutine
$4.99
- Fries
$2.99

########CHECKING RECENT ORDERS BELOW


Recent orders of customer Ellen Ripley
-- AvoidThePlates order #900001
For customer Ellen Ripley with restaurant Burger Queen
- Super extra steakhouse triple queen wooper with cheese
$11.99
- Fries
$2.99
- Queen wooper
$8.99
- Poutine
$4.99
Total price: $28.96


Recent orders of customer Ellen Ripley
-- AvoidThePlates order #900004
For customer Ellen Ripley with restaurant Burger Queen
- Poutine
$4.99
Total price: $4.99
-- SuperConsume order #900003
For customer Ellen Ripley with restaurant City Sushi
- City nigiri
$3.99
- City sashimi
$11.99
Total price: $15.98
-- SuperConsume order #900002
For customer Ellen Ripley with restaurant Burger Queen
- Poutine
$4.99
Total price: $4.99
```

Hand in

Submit your five Java files (**`Customer.java`**, **`FoodItem.java`**, **`Restaurant.java`**, **`FoodApp.java`**, **`Order.java`**). ***Do not submit .class or .java~ files!*** You do ***not*** need to submit the **`TestPhaseN.java`** files that were given to you. If you did not complete all phases of the assignment, **use the Comments field when you hand in the assignment to tell the marker which phases were completed**, so that only the appropriate tests can be run. For example, if you say that you completed Phases 1-2, then the marker will compile your files, and compile and run the tests for phases 1 and 2. If your submitted code fails to compile and run, you will lose ***all*** of the marks for the test runs. The marker will ***not*** try to run anything else, and will ***not*** edit your files in any way. ***Make sure none of your files specify a package at the top, otherwise your code will not compile on the marker's computer!***