DUE DATE: FEBRUARY 6, 2019 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.
- 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.

Assignment overview

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

- **`Customer`**: A customer who orders drinks
- **`Ingredient`**: A coffee ingredient with a name, amount (in ml) and cost ($ per Liter)
- **`Drink`**: A specification for a drink, with a name and set of ingredients
- **`DrinkOrder`**: A drink ordered by a customer
- **`CoffeeShop`**: A coffee shop that decides how much to charge for a drink and maintains a list of orders to complete, total profit at a particular location, etc.

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. 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).

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 **toString** methods. The only exceptions in this assignment are **`displayOrdersPending`** in **`CoffeeShop`** (Phase 3) and **`displayStoresVisted`** in **`Customer`** (Phase 4).

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.

Phase 1: Customers and Ingredients

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

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

- Two instance variables: the customer's first name (a **`String`**) and last name (a **`String`**)
- A constructor that has two parameters, in the above order, which are used to initialize the two instance variables.
- A standard **`toString`** method, which returns a **`String`** containing the customer's last and first names separated by a comma (as shown in the output below).
- An **equals** method that checks if two customers are equal. Two customers are equal if they have the same first and last name.

The **Ingredient** class should have:

- Three instance variables: the ingredient's name (a **String**), the ingredient's amount in ml (an **int**) and the ingredient's price (in $per L) (a **double**)
- A constructor that has three parameters, in the above order, which are used to initialize the three instance variables.
- A method **getCost** that calculates and returns the total cost of the ingredient (a double). Due to the way floating point arithmetic works, you might see some rounding errors as in the sample output below.
- A **toString** method which returns a **String** containing the ingredient name, amount, and price (per L) separated by commas (as shown below).

You can test your classes using the supplied test program **TestPhase1.java**. You should get the output shown below (plus or minus any different rounding errors with floating point arithmetic).

```
Customer1: Mars, Bruno
Customer2: Swift, Taylor
Customer3: Swift, Taylor
C1 equals C2? false
C2 equals C3? true

Ingredient1: Milk, 500 mls, $2.55/L
Ingredient1 cost: $1.275

Ingredient2: Espresso, 100 mls, $4.05/L
Ingredient2 cost: $0.405

Ingredient3: Hazelnut Syrup, 30 mls, $13.5/L
Ingredient3: $0.40499999999999997
```

Phase 2: Drinks and DrinkOrders

Next, implement the **Drink** class, which represents the specification for one type of drink.

The **Drink** class should have:

- Two instance variables: the name of the drink (a **String**), and an array of **Ingredients**. Use a simple partially-filled array for this list of ingredients. You may assume that the list will never become full – no error checking is needed.
- A constructor that accepts only one parameter, the drink's name (type **String**).
- A method **addIngredient**(**Ingredient**) that adds an ingredient to the drink's list of ingredients
- A method **calculateCost** that returns a **double** representing the total cost of the drink (based on the costs of each individual ingredient).
- A **toString** method that returns a **String** containing the drink's name and list of ingredients. Each ingredient should appear on a separate line, with a tab (\t) character at the beginning of the line.

Then, implement a **DrinkOrder** class, which represents a particular customer's drink order. This class should have:

- Three instance variables: the customer who ordered the drink (a **Customer**), the drink requested (a **Drink**), the amount charged (in $) (a **double**)
- A constructor that has three parameters, in the above order, which are used to initialize the three instance variables.
- A method **belongsTo**(**Customer**) that returns true if the drink order belongs to the customer passed as a parameter and false otherwise
- A **getProfit** method that returns a **double** representing the profit made for this drink (i.e., the shop's financial gain for the drink order in light of the drink's actual cost).
- A **toString** method that returns a **String** containing the customer, amount charged, and drink information (as shown below).

You can test your class with **`TestPhase2.java`**. You should get the output shown below (plus or minus any different rounding errors due to floating point arithmetic).

```
Customer1: Mars, Bruno
Customer2: Swift, Taylor
Customer3: Ada, Lovelace

Drink1: Latte, Ingredients:
     Milk, 500 mls, $2.55/L
     Espresso, 100 mls, $4.05/L

Drink1 cost $1.68

Drink2: Soy Latte, Ingredients:
     Soy Milk, 500 mls, $4.75/L
     Espresso, 100 mls, $4.05/L

Drink2 cost $2.7800000000000002

Drink3: Hazelnut Latte, Ingredients:
     Milk, 500 mls, $2.55/L
     Espresso, 100 mls, $4.05/L
     Hazelnut Syrup, 30 mls, $13.5/L

Drink3 $2.085

Mars, Bruno, $4.5: Latte, Ingredients:
     Milk, 500 mls, $2.55/L
     Espresso, 100 mls, $4.05/L

Profit: $2.8200000000000003
Belongs to Taylor? false

Mars, Bruno, $5.0: Hazelnut Latte, Ingredients:
     Milk, 500 mls, $2.55/L
     Espresso, 100 mls, $4.05/L
     Hazelnut Syrup, 30 mls, $13.5/L

Profit: $2.915
Belongs to Bruno? true

Swift, Taylor, $6.0: Soy Latte, Ingredients:
     Soy Milk, 500 mls, $4.75/L
     Espresso, 100 mls, $4.05/L

Profit: $3.2199999999999998
Belongs to Taylor? true

Ada, Lovelace, $4.75: Latte, Ingredients:
     Milk, 500 mls, $2.55/L
     Espresso, 100 mls, $4.05/L
```

```
Profit: $3.0700000000000003
Belongs to Ada? true
```

Phase 3: CoffeeShop

Next implement the **`CoffeeShop`** class, which contains the information and methods for a particular coffee shop.

The **`CoffeeShop`** class should have:

- A **`String`** instance variable to hold the name of the shop
- An instance variable that contains a list of drink orders to be filled. Use a simple partially-filled array to do this. You may assume that the list will never become full – no error checking is needed.
- A **`double`** instance variable that contains the shop's "mark-up" factor. The coffee shop will use this mark-up to calculate how much to charge for a drink given its actual cost (see **`newOrder`** below for details).
- An instance variable for the shop's total profit thus far (a **double**)
- A static class variable that stores the total number of drink orders filled across all CoffeeShops (an **`int`**)
- A constructor with two parameters: the shop's name (a **`String`**) and the shop's mark-up factor (a **double)**
- A simple accessor method **`getName`**() that returns the shop's name (a **`String`**).
- A method **`void`** **newOrder**(**Customer, Drink**) that creates a **`DrinkOrder`** for that customer and adds the order to the partially filled array. When creating the DrinkOrder, use the mark-up factor instance variable and the drink's actual cost to determine how much to charge the customer. We are using a simple multiplicative formula for this assignment. For example, a mark-up factor of 1.1 means that the shop charges 1.1 times the actual cost of any drink. We won't worry about rounding the calculation to two decimal places. The method should also update the shop's profit instance variable and the class variable representing the total number of orders across all coffee shops.
- A method **`void`** **orderFilled**(**Customer**) that removes the customer's drink order from the partially filled array. You can assume that each Customer has only one order waiting to be filled at the shop. To keep this method short, you can create a private "helper" method that returns the position of the drink order in the array.
- A method **getNumOrdersPending**() that returns the number of drink orders needing to be filled at the shop (an **int**).
- A method **`void`** **displayOrdersPending**() that displays the drink orders that still need to be completed at the shop. Note: this is one of only two instance methods in this assignment that outputs directly to the screen.
- A method **`getProfit`** that returns the shop's total profit (a **`double`**).
- A class method **`getTotalOrders`** that returns the number of drink orders across all coffee shops (an **`int`**).

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

```
Orders pending at Starbucks:
1:  Mars, Bruno, $5.88: Latte, Ingredients:
     Milk, 500 mls, $2.55/L
     Espresso, 100 mls, $4.05/L

Starbucks profit: 4.2

Orders pending at Starbucks:
1:  Mars, Bruno, $5.88: Latte, Ingredients:
     Milk, 500 mls, $2.55/L
     Espresso, 100 mls, $4.05/L

2:  Swift, Taylor, $9.73: Soy Latte, Ingredients:
     Soy Milk, 500 mls, $4.75/L
     Espresso, 100 mls, $4.05/L

Starbucks profit: 11.15
```

```
Filled order at Starbucks: Swift, Taylor
Orders pending at Starbucks:
1:  Mars, Bruno, $5.88: Latte, Ingredients:
     Milk, 500 mls, $2.55/L
     Espresso, 100 mls, $4.05/L

Orders pending at Second Cup:
1:  Ada, Lovelace, $3.528: Latte, Ingredients:
     Milk, 500 mls, $2.55/L
     Espresso, 100 mls, $4.05/L

2:  Swift, Taylor, $5.838000000000001: Soy Latte, Ingredients:
     Soy Milk, 500 mls, $4.75/L
     Espresso, 100 mls, $4.05/L

Second Cup profit: 4.906000000000001

Orders pending at Second Cup:
1:  Ada, Lovelace, $3.528: Latte, Ingredients:
     Milk, 500 mls, $2.55/L
     Espresso, 100 mls, $4.05/L

2:  Swift, Taylor, $5.838000000000001: Soy Latte, Ingredients:
     Soy Milk, 500 mls, $4.75/L
     Espresso, 100 mls, $4.05/L

3:  Mars, Bruno, $4.3785: Hazelnut Latte, Ingredients:
     Milk, 500 mls, $2.55/L
     Espresso, 100 mls, $4.05/L
     Hazelnut Syrup, 30 mls, $13.5/L

Second Cup profit: 7.1995000000000005

Trying to fill an order not at Starbucks: Ada, Lovelace
Orders pending at Starbucks:
1:  Mars, Bruno, $5.88: Latte, Ingredients:
     Milk, 500 mls, $2.55/L
     Espresso, 100 mls, $4.05/L

Filled order at Starbucks: Mars, Bruno
Orders pending at Starbucks:
None

Total coffee orders: 5
```

Phase 4: Interactions between Customers and CoffeeShops

In this phase, you will create some interactions between your **`Customer`** and **`CoffeeShop`** classes. This is advanced material, designed to stress-test your understanding of objects and their relationships. You might find this phase tough.

Specifically, make the following modifications to your existing classes, so that a **`Customer`** will contain information on the different stores they have visited.

- Add instance variables to the **`Customer`** class which can hold a list of references to **`CoffeeShop`** objects. Use a simple partially-filled array. You can assume this list will never become full.
- Modify your constructor as needed.
- Add a method **`void addStore(CoffeeShop)`** to the **`Customer`** class which will add a **`CoffeeShop`** to the list, but only if it does not already appear in the list (there should never be duplicates).
- Modify the **`newOrder`** method in the **`CoffeeShop`** class, so that when the shop receives an order from a customer, the coffee shop will be added to that Customer's list of shops visited.
- Add an instance method **`void displayStoresVisited`**() to the **`Customer`** class that displays the names of the stores that the customer has visited (this is the second instance method that prints right to the screen).
- Add an instance method **`getShortestWait`**() that returns the **`CoffeeShop`** from the customer's previously visited stores that has the smallest number of orders to fill (using the **`getNumOrdersPending`** instance method from the CoffeeShop class)

You can test your Phase 4 additions with TestPhase4.java. You should get the output shown below.

```
Orders pending at Starbucks:
1:  Mars, Bruno, $5.88: Latte, Ingredients:
      Milk, 500 mls, $2.55/L
      Espresso, 100 mls, $4.05/L

Starbucks profit: 4.2

Orders pending at Starbucks:
1:  Mars, Bruno, $5.88: Latte, Ingredients:
      Milk, 500 mls, $2.55/L
      Espresso, 100 mls, $4.05/L

2:  Swift, Taylor, $9.73: Soy Latte, Ingredients:
      Soy Milk, 500 mls, $4.75/L
      Espresso, 100 mls, $4.05/L

Starbucks profit: 11.15

Filled order at Starbucks: Swift, Taylor
Orders pending at Starbucks:
1:  Mars, Bruno, $5.88: Latte, Ingredients:
      Milk, 500 mls, $2.55/L
      Espresso, 100 mls, $4.05/L

Orders pending at Second Cup:
1:  Ada, Lovelace, $3.528: Latte, Ingredients:
      Milk, 500 mls, $2.55/L
      Espresso, 100 mls, $4.05/L

2:  Swift, Taylor, $5.838000000000001: Soy Latte, Ingredients:
      Soy Milk, 500 mls, $4.75/L
      Espresso, 100 mls, $4.05/L

Second Cup profit: 4.906000000000001

Orders pending at Second Cup:
1:  Ada, Lovelace, $3.528: Latte, Ingredients:
      Milk, 500 mls, $2.55/L
      Espresso, 100 mls, $4.05/L

2:  Swift, Taylor, $5.838000000000001: Soy Latte, Ingredients:
```

```
      Soy Milk, 500 mls, $4.75/L
      Espresso, 100 mls, $4.05/L

3:  Mars, Bruno, $4.3785: Hazelnut Latte, Ingredients:
      Milk, 500 mls, $2.55/L
      Espresso, 100 mls, $4.05/L
      Hazelnut Syrup, 30 mls, $13.5/L

Second Cup profit: 7.1995000000000005

Trying to fill an order not at Starbucks: Ada, Lovelace
Orders pending at Starbucks:
1:  Mars, Bruno, $5.88: Latte, Ingredients:
      Milk, 500 mls, $2.55/L
      Espresso, 100 mls, $4.05/L

Filled order at Starbucks: Mars, Bruno
Orders pending at Starbucks:
None
Orders pending at Starbucks:
1:  Swift, Taylor, $7.297499999999999: Hazelnut Latte, Ingredients:
      Milk, 500 mls, $2.55/L
      Espresso, 100 mls, $4.05/L
      Hazelnut Syrup, 30 mls, $13.5/L

Starbucks profit: 16.3625

Total coffee orders: 6

Swift, Taylor has visited:
Starbucks
Second Cup

Mars, Bruno has visited:
Starbucks
Second Cup

Ada, Lovelace has visited:
Second Cup

shortest wait for: Swift, Taylor is: Starbucks
shortest wait for: Ada, Lovelace is: Second Cup
```

Hand in

Submit your five Java files (**`Customer.java`**, **`Ingredient.java`**, **`Drink.java, DrinkOrder.java, CoffeeShop.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 four 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 also TestPhase2.java, and then run the main method in TestPhase2. If it 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!***