Lab 07: Object Oriented Programming
CSE-021-01, Spr 2018 **Points: 20**

Overview

This week we are going to be introduced to the Object Oriented Programming (OOP) paradigm. We are going to tackle the problem of creating a Cheese Shop in a new way using OOP. Luckily, we have provided substantial amount of the code for gentle introduction into this new territory. This lab serves as a study guide for OOP and recursion. Recursion occurs when a method calls itself to perform a smaller version of the same task [cf. Sections 12.1 and 12.2]. We use recursion to allow the customer to redo their whole order any number of times. We also implement additional logic that provides the customer a 1% chance of getting their order for free when checking out.

Before you get started, read chapters 7.1, 7.2, 7.4 and 7.7 if you haven't already done so. Answer activities 7.2.3, 7.4.1, 7.7.2 and 7.7.4 in a Word or text file. Also, answer the Assessment questions as you encounter them in the next section. The prompts for answering assessment questions are placed immediately following the material to which the listed questions relate.

Getting Started

You should have a Java project in Eclipse titled Lab 21_7. This PDF document is included in the project in the **doc** directory. The Java files you will use in this lab are in the **src** directory.

We start from the simpler version of the shop that only sells three fixed types of cheeses. When we're solving problems using OOP, we need to undertake two necessary tasks. The first task is to **define** the class along with its expected behaviors. Second task is to **instantiate** an object of that class type. Usually we can only manipulate an instantiated object of a class.

To create a Cheese Shop, the first thing (object) we need is a shop. So, we define a `Shop` class which behaves as follows:

A. List all the cheese types available and the prices
B. Asks the user how many pounds of each type of cheese to purchase
C. Calculate Sub Total (price*amount of each cheese added together)
D. Discount of Sub Total -
   - A $10 discount if their purchase is $50 or over
   - An additional $15 discount ($25 total) if $100 or over
E. Ask the user if they would like to see a list of what they purchased
   - If yes, a list comes up showing how much of each type of cheese they bought and the cost of each cheese
   - Display only the cheese they actually bought
   - If no, then no itemized information is displayed
F. Display Sub Total, Discount and Total Price

Sample Output:

```
We sell 3 kinds of Cheese
Dalaran Sharp: $1.25 per pound
Stormwind Brie: $10.00 per pound
Alterac Swiss: $40.00 per pound
Enter the amount of Sharp: 1
Enter the amount of Brie: 1
Enter the amount of Swiss: 1
Display the itemized list? (1 for yes) 1
1 lbs of Sharp @ $1.25 = $1.25
```

```
1 lbs of Brie @ $10.00 = $10.0
1 lbs of Swiss @ $40.00 = $40.0
Sub Total: $51.25
-Discount: $10.0
Total:     $41.25

We sell 3 kinds of Cheese
Dalaran Sharp: $1.25 per pound
Stormwind Brie: $10.00 per pound
Alterac Swiss: $40.00 per pound
Enter the amount of Sharp: 1
Enter the amount of Brie: 2
Enter the amount of Swiss: 3
Display the itemized list? (1 for yes) 1
1 lbs of Sharp @ $1.25 = $1.25
2 lbs of Brie @ $10.00 = $20.0
3 lbs of Swiss @ $40.00 = $120.0
Sub Total: $141.25
-Discount: $25.0
Total:     $116.25

We sell 3 kinds of Cheese
Dalaran Sharp: $1.25 per pound
Stormwind Brie: $10.00 per pound
Alterac Swiss: $40.00 per pound
Enter the amount of Sharp: 1
Enter the amount of Brie: 0
Enter the amount of Swiss: 0
Display the itemized list? (1 for yes) 1
1 lbs of Sharp @ $1.25 = $1.25
Sub Total: $1.25
-Discount: $0.0
Total:     $1.25

We sell 3 kinds of Cheese
Dalaran Sharp: $1.25 per pound
Stormwind Brie: $10.00 per pound
Alterac Swiss: $40.00 per pound
Enter the amount of Sharp: 0
Enter the amount of Brie: 1
Enter the amount of Swiss: 0
Display the itemized list? (1 for yes) 1
1 lbs of Brie @ $10.00 = $10.0
Sub Total: $10.0
-Discount: $0.0
Total:     $10.0

We sell 3 kinds of Cheese
Dalaran Sharp: $1.25 per pound
Stormwind Brie: $10.00 per pound
Alterac Swiss: $40.00 per pound
Enter the amount of Sharp: 1
Enter the amount of Brie: 1
Enter the amount of Swiss: 1
Display the itemized list? (1 for yes) 0
Sub Total: $51.25
-Discount: $10.0
Total:     $41.25

We sell 3 kinds of Cheese
Dalaran Sharp: $1.25 per pound
```

```
Stormwind Brie: $10.00 per pound
Alterac Swiss: $40.00 per pound
Enter the amount of Sharp: 0
Enter the amount of Brie: 0
Enter the amount of Swiss: 0
Display the itemized list? (1 for yes) 1
Sub Total: $0.0
-Discount: $0.0
Total:     $0.0
```

`Shop` class will need to have different cheese types and that's where the `Cheese` class comes in. `Shop` will create (**instantiate**) objects of type `Cheese` to sell to the customers.

We need to **define** the `Cheese` class whose objects will be used and manipulated by the `Shop` class. Each cheese has the same basic behavior and state, so we only need to define it once. A cheese has a name, a price and an amount the customer wants to buy. These can be naturally modeled by *instance (or member) variables (or fields)* declared inside the `Cheese` class. Then we need to provide member methods in the `Cheese` class as the primary way to manipulate these variables by other objects, namely of type `Shop` in our case.

Finally, we need a class with a `main` method to execute the whole program, and that's where the `RunShop` class comes in. This `main` method will create a `shop` object of type `Shop` and call a method called `shop.run()` to start our Cheese Shop.

Constructors:

Each class needs to contain at least one type of constructor (method with the same name as the class) that is called when an object or an instance of the class is created. Everything we have done before is static and you may notice a lack of any constructors. `RunShop`, for example, does not have a constructor as Java provides one as default. However, when we need a more sophisticated behavior, we need to have specialized constructors.

We have talked briefly about **overloading** but this will be the first time we will utilize it in our labs. The `Cheese` class will have three different ways of creating an instance using three types of constructors. The difference lies in the parameters each constructor expects that are unique among the three -- this way, Java can figure out which one needs to be called depending on the arguments passed during object instantiation. The first `Cheese` constructor takes no argument, and simply initializes all variables. The second constructor takes the name of a cheese as input, and uses it to initialize the `name` variable. The third constructor is the most complete, taking the price of a cheese along with its name as inputs.

```
public Cheese()
public Cheese(String name)
public Cheese(String name, double price)
```

[Answer assessment questions 1 and 2]

Accessors:

Each cheese type requires three instance variables: `name`, `price` and `amount`. If the `shop` or any other object wants to know the value of these variables, they should use accessor methods. As this will be the only way to get access, three accessors must be implemented by `Cheese`. These accessors each contain one `return` statement, returning the value of one of the three aforementioned variables to the caller.

```
public String getName()
public double getPrice()
public int getAmount()
```

[Answer assessment question 3]

Mutators:

Each of the variables in the `Cheese` class will also need to be set or modified, and that's where mutators come in. `Cheese` is a simple class, so there is one mutator method corresponding to each accessor method as seen below:

```
public void setName(String newName)
public void setPrice(double newPrice)
public void setAmount(int newAmount)
```

[Answer assessment question 4]

Cheese Class Details:

Let us look at a simplified version of the `Cheese` class implementation:

```
public class Cheese {
   private String name;
   private double price;
   private int amount;

   public static int numCheese = 0;

   public Cheese() { // Constructor with no parameters
      name = "";
      price = 0;
      amount = 0;
      numCheese++;
   }

   public Cheese(String name) { // Constructor with name as parameter
      this.name = name;
      price = 0;
      amount = 0;
      numCheese++;
   }

   public String getName() { // Accessor
      return name;
   }

   public void setName(String newName) { // Mutator
      name = newName;
   }
}
```

First, we have declared three variables `name`, `price` and `amount` with their appropriate types as instance variables. Instance variables are unique to each instance (object) of the class type. In addition, they are all declared as `private` so only objects of type `Cheese` can access their values; otherwise objects of other class types will be able to access the variables directly without going through the accessor methods. We have a `public` *class variable* called `numCheese` that is used to count how many cheese instances have been created. Class variables are shared by all objects of that class type.

[Answer assessment question 5]

Initialization:

It is very important to have a base constructor which initializes all variables to a default value. In our case, we set `name` to an empty string, `price` to `0` and `amount` to `0`. We increment the current count of cheese in each constructor so we can keep track of the total number of cheese objects created.

Variable Resolution:

The second constructor takes a cheese name as input, and initializes the `name` variable to the value passed as the argument. However, you may notice that there are two variables called `name` that this method knows about. First is the instance variable declared in the class, and second is the input parameter to the method. This is a similar situation to overloading where we need to provide a way for Java to differentiate between the two. So, we introduce a way of accessing the implicitly-passed object reference using `this` [cf. Section 7.8]. Remember everything is an object of some class type now, so we can access the class member (or an object's instance variable) using `this`. Thus, we have the following line to appropriately resolve both names and do the correct thing:

```
This.name = name;
```

Notice we did not need to use `this` in the first constructor for any of the variables. We also did not add it to price and amount in the second constructor. It is equally valid to write:

```
this.name = name;
this.price = 0;
this.amount = 0;
```

You may write code as shown above to avoid any confusion about the actual variable being referred to by the code. We provided both ways so you know about each method of accessing variables, and the situations in which they should be used. You will write the third constructor code based on the two examples given above.

[Answer assessment question 6]

Get and Set Methods:

`getName` basically returns `name` and `setName` assigns the value of `newName` to `name`. These two basic methods need to exist and perform the same function for every member variable of the class. You will need to implement this method pairing for the other two variables.

Shop class details

We will step through the code for the `Shop` class to explain things that are of interest. Two methods, namely `discount` and `printTotals`, are unchanged from Lab 04. A method called `printFree` has been newly created to emulate behavior when a shop gives everything away for free. Let us begin with the constructor code for `Shop` which takes no parameters.

```
Cheese Sharp, Brie, Swiss;

public Shop() {
   Sharp = new Cheese();
   Sharp.setName("Sharp");
   Sharp.setPrice(1.25);

   Brie = new Cheese("Brie");
   Brie.setPrice(10.00);
```

```
        Swiss = new Cheese("Swiss", 40.00);
    }
```

You can see that this constructor basically creates 3 instances of cheese using three different constructors. The class contains three instance variables of type `Cheese` so the member methods can access them freely without having to pass them as parameters. Creating the `Sharp` object calls the empty parameter constructor of `Cheese`, and the subsequent calls to the two accessors correctly set the instance variables. Creating the `Brie` object calls the constructor with the cheese name as input parameter, and only needs one subsequent call to an accessor to set the price. Creating the `Swiss` object does not need any subsequent calls to accessors since both values are passed in as parameters to the constructor. So, after this we have successfully created all three cheeses that shop is going to sell.

[Answer assessment question 7]

Next, `intro` needs to ask the amounts for all three cheeses. You will have to add code for `Brie` and `Swiss`. The partial code for this member method is as follows:

```
    private void intro(Scanner input) {
        System.out.println("We sell 3 types of Cheese");
        System.out.println(Sharp.getName() + ": $" + Sharp.getPrice() + " per pound");
        System.out.println(Brie.getName() + ": $" + Brie.getPrice() + " per pound");
        System.out.println(Swiss.getName() + ": $" + Swiss.getPrice() + " per pound");

        System.out.print("Enter amount of " + Sharp.getName() + " : ");
        Sharp.setAmount(input.nextInt());
    }
```

[Answer assessment question 8]

The `intro` method now takes a Scanner type as a parameter. This avoids the need to create a new object of type Scanner every time `intro` is called. Since we know that `input.nextInt()` will return an `int`, we pass the returned value directly as an argument to `setAmount` without having to use a temporary variable to store it first.

The next method is `calcSubTotal` that does not need any parameters, since it now knows about the object instance variables (or class members) of type `Cheese`.

```
    private double calcSubTotal() {
        double subTotal = 0;
        subTotal += Sharp.getAmount() * Sharp.getPrice();

        return subTotal;
    }
```

[Answer assessment question 9]

`calcSubTotal` can call each cheese's amount and price accessors and multiply the returned values to add to the `subTotal`. The fact that each member of type `Cheese` simply knows all the information about itself, simplifies the code by using accessor methods. You must add in the rest of the logic so that `calcSubTotal` returns the correct value.

The code for displaying the list of purchased items is as follows:

```
    private void itemizedList(){
        int amt;
        System.out.println();
        if ((amt = Sharp.getAmount()) > 0)
            System.out.println(amt + " lbs of Dalaran Sharp @$1.25 = " + (amt *
                                                                    Sharp.getPrice()));
    }
```

We use a temporary variable called `amt` to store the result of `getAmount` so we can avoid calling the method multiple times. The expression `amt = Sharp.getAmount()` assigns the amount of `Sharp` cheese to `amt`. Then that value is compared against `0` to see if it's greater. If it is, then we print out the amount of the purchase and cost. This is a small programming trick that can speed up execution because method calls are potentially slow. You can complete the rest of the code to print out the other cheese purchased in similar fashion.

Lastly, we have `run()` which contains the guts of the program:

```
    public void run() {

        Scanner input = new Scanner(System.in);
        intro(input);
        double sub = calcSubTotal();
        double disAmt = discount(sub);

        System.out.println();
        System.out.print("Do you want to see the itemized list? (1 for yes): ");
        int list = input.nextInt();
        if (list == 1)
            itemizedList();

        int free = (new Random()).nextInt(100);
        //System.out.println("Random num is " + free);
        if (free != 0)
            printTotal(sub, disAmt);
        else {
            printFree();
            return;
        }

        System.out.println();
        System.out.print("Do you wish to redo your whole order? (1 for yes): ");
        int redo = input.nextInt();

        System.out.println();

        if (redo == 1)
            run();
        else
            System.out.println("Thanks for coming!");
    }
```

[Answer assessment question 10]

First thing we do is create a Scanner object called `input` and use it as an argument in the call to `intro`. After we are done with gathering the amounts of cheeses that a customer would like to purchase, we perform two method calls. First we call `calcSubTotal()` and assign the returned value to a variable `sub`. Then we pass `sub` as an argument to `discount`, and assign the returned value to `disAmt`. Next, we ask the customer if they would like to see an itemized list, and if the input is 1, we call `itemizedList`, otherwise we proceed to the next step. Now we see the code that provides the customer a 1% chance of getting their order for free. To do so, we generate a random number between 0 and 99, and if it is 0, the customer gets their order for free. Here a trick is used as shown in the following line:

```
int free = (new Random()).nextInt(100);
```

The Random class is used to get an integer between 0 – 99. However, since we only use the random number generator object once, we need not store the object in a variable (as we did in Lab 03), as it is not used later. Instead, by using extra set of parentheses, we can tell Java to directly use the object returned and access its method. `(new Random())` represents the new object, and `nextInt(100)` is the method call with input 100.

If the user is not lucky enough to get a free purchase then we print out the normal amount using `printTotal`. Otherwise we call the special `printFree` method and effectively end the current method's execution with `return;`. Even though the return type is void, we can end any method with empty return statement as shown below.

```
if (free != 0)
   printTotal(sub, disAmt);
else {
   printFree();
   return;
}
```

Finally we ask the user if they want to redo the whole order and if the answer is yes (`1`), we can just recursively call `run()` in the if statement shown below.

```
if (redo == 1)
   run();
```

This way the ordering process keeps going until the user is done or the purchase is free.

Part 1: Fill-in `Shop.java` and `Cheese.java`

Everywhere you see the comment to "Fill in Code" is where you need to add code to make this program behave correctly. If the comment is to "Fix Code", you need to change existing code. In most places a sample is provided to help you get started. The program currently runs but the values are obviously incorrect.

`main` is inside `RunShop.java` and there is where you should run the whole program.

Part 2: (Assessment) Logic Check and Level of Understanding

1) How can you tell a method is a constructor?
2) Would `public void cheese()` be considered a constructor?
3) Does it make sense to have `private` or `void` accessor method?
4) Would `public void setName()` be a good mutator declaration?
5) How can you tell the difference between instance and class variables?
6) Can we write `name = name;` and what would it mean?
7) How can you tell which version of the constructor is being called?
8) What does the `.` operator do for objects?
9) Can you use a loop to implement `calcSubTotal`?
10) Can you tell when and where we do the recursion in `run()`?
11) What does `this` refer to?
12) What should be the value of `numCheese` when `RunShop` terminates (i.e., the output of the `println` statement)?

13) Give the code to implement `public void setName(String name) { … }`.

What to hand in

When you are done with this lab assignment, submit all your work through CatCourses.

Before you submit, make sure you have done the following:

- Verified your solution with your TA or instructor
- Included answers to activities 7.2.3, 7.4.1, 7.7.2 and 7.7.4, and Assessment questions (1 – 13) in a Word document or text file named `Part2`
- Attached the `Shop.java`, `Cheese.java` and `Part2` files
- Filled in your collaborator's name in the "Comments…" text-box (if any) at the submission page.