Lab #6 – Object Oriented Programming

Overview

This week we are going to be introduced to the Object Oriented Programming paradigm. We are going to use a new problem of creating a Cheese shop. Luckily, we have provided substantial amount of the code for gentle introduction into this new territory. This lab can serve as a study guide for OOP and recursion. Recursion is where we call the method in itself since it's solving the same yet smaller program. We solve the cheese problem using recursion in the main code. We also implement additional logic that gives has 1% chance of giving free cheese when checking out.

Reading : Chapter 7.1, 7.2, 7.4, & 7.7

- Answer Participation Activity 7.2.3
- Answer Participation Activity 7.4.1
- Answer Participation Activity 7.7.2 and 7.7.4

Getting started

You should have a Java project in Eclipse title Lab 21_6. This pdf is included in the project in the *doc* directory. The Java files you will use in this lab are in the *src* directory, as usual.

We start from the simpler version of the shop that only sells three fixed type of cheese. When we're doing OOP, there are two different steps we need to for each object or Class. First is to **define** the Class along with the behavior expected from it. Second is to **instantiate** an object of that type. Usually we can only manipulate an instantiated object of a class.

So the first thing (Object) that we need is a shop. So we will **define** a Shop class, which will behave as the following:

- List all the cheese types available and the prices
- Asks the user how many pounds of each type of cheese to purchase
- Calculate Sub Total (price*amount of each cheese added together)

- Discount of Sub Total
  - for a $10 discount if their purchase is $50 or over
  - an additional $15 discount ($25 total) if $100 or over
- 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
- 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 cheese to sell to the user.

We need to **define** the Cheese class for the above Shop class to use and manipulate. 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 user wants to buy. These are naturally modeled by variables declared inside the Cheese class. Then we provide methods as the primary way to manipulate these variables by other objects namely Shop in this case.

Finally we need something with *main* to executable the whole program and that's where RunShop Class comes in. So it creates a shop object inside and call 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) so an object or an instance of the class can be created. Everything we have done before is *static* and you may notice a lack of any constructors. RunShop for example also does not have one as Java provides one as default. However when we need a more sophisticated behavior then we start having the 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. Cheese class will have three different ways of creating an instance. The difference lies in the parameters each constructor expects and each is unique among the three. So therefore java can figure out which one needs to be called. First Cheese constructor takes no argument so it just initializes variables. Second constructor has a name of the cheese as input as it should use that instead of initializing to empty string. Third constructor is the most complete with the price of the cheese along with the name given as inputs.

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

Q1. How can you tell a method is a constructor?

Q2. Would `public void cheese()` be considered a constructor?

Accessors:

Each cheese type requires three variables: name, price and amount. If the shop or any other object wants to know the value of these variables then they have to use accessor methods. As this is the only way to get access, three accessors must be implemented by Cheese. Mostly they contain one *return* statement to give the value of the corresponding variable to the caller.

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

Q3. Would it make sense to have `private` OR `void` accessor method?

Mutators:

Each of the variables will also need to be set or modified and that is where mutators come in. Cheese is a simple class so there is a corresponding mutator method to the accessor method as you can see below:

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

Q4. Would `public void setName()` be a good mutator declaration?

Cheese Class:

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

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

	public static int numCheese = 0;

	public Cheese() { // initialization
		name = "";
		price = 0;
		amount = 0;
		numCheese++;
	}

	public Cheese(String name) { // Constructor with name input
		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 of the class object. In addition they are all declared as *private* so only Cheese can access their values; otherwise other objects can access the variables directly without going through the accessor methods. We have a public variable *numCheese* as a *class variable* to count how many cheese instances have been created. Class variables are shared by all the objects of that class definition.

Q5. How can you tell the difference between instance and class variables?

Initialization
It is very important to have a base constructor which just makes all our variables to a default value. In this 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
Second constructor has *name* as input so it does not need to be initialized to empty string but instead to the value passed in. However, you may notice that we have two variables called *name* that this method knows about. First is the one declared in the class and second is the input parameter to the method. This is similar situation to **overloading** where we need to provide a way for Java to differentiate between the two. So we introduce a way of addressing oneself with this. Remember everything is an object now so we can say use the object's variable and in this case the object is oneself. Thus we have the following line to appropriately resolve both names and do the correct thing:

```
this.name = name;
```

Noticed we did not need to use this in the first constructor for all three 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 do so if you wish in order never be confused about the actual variable being referred to by the code. I provided both ways of addressing so you know about each and the situations they should be used. You will write the third constructor code based on the two examples given.

Q6. Can we write `name = name;` and what would it mean?

Get and Set
getName basically return *name* and setName makes *name* be the value of *newName*. These two basic methods need to exist and perform the same function for every class variable. You will need to implement this method pairing for the other two variables.

Shop Class:

We will step through the code for Shop to explain things that are of interest. We provided code for you that is unchanged from Lab 4, namely discount and printTotal methods. We created a new method called printFree that is similar to when the shop gives everything 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 it basically creates 3 instances of cheese using three different constructors. The class contains three Cheese instance variables so the member methods can access them freely without having to pass it as parameters. Sharp is calling the empty parameter constructor of Cheese and call 2 accessors to make the values correct. Brie will call constructor with name input parameter and only needs to set price. Swiss does not need to set everything 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.

Q7. How can you tell which version of the constructor is being called?

```
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());
}
```

Q8. What does the . operator do for objects?

Intro needs to ask for all three cheese amounts so you will have to add code for Brie and Swiss. Intro now takes in a Scanner type just so you can see these things can be passed in as parameter. This avoids the need to create new object every time intro is called. You know that input.nextInt() will return an int so we pass that returned value directly as parameter to setAmount without having to use a temporary variable to store it first.

```
private double calcSubTotal() {
	double amount = 0;
```

```
        amount += Sharp.getAmount() * Sharp.getPrice();

        return amount;
    }
```

Q9. Can you use a loop to implement calcSubTotal?

SubTotal does not need any parameters since it knows about Cheese variables. It can just call each cheese's amount and the price to multiply them in order to add to the sub total. It makes our lives simple that each Cheese simply knows all the information about itself so we can just use accessor methods . You can just add in the rest of the logic so the subTotal is returning the correct value.

```
    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 want to use temporary variable *amt* to store the result of getAmount so we can avoid calling the method multiple times. By using the assignment amt = Sharp.getAmount()we can first put the value returned into *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 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 = subTotal();
        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!");
      }
}
```

Q10. Can you tell when and where we do the recursion in run()?

First thing we do is create a Scanner object *input* and promptly used it in the call to intro(input). After we are done with gathering the amount of cheese the user like to purchase then we do two method calls. First we call subTotal() and put the result in variable *sub*. Then we use *sub* in the call to getDiscount and put the return value into *disAmt*. We need to ask the user if they would like to see an itemized list and if the input is 1 then call printList() otherwise skip to the next step. One more trick used is the following line:

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

We want to use a Random class to get an integer 0-99. However it is only going to be used once so instead of storing to a variable first we are just going to use it. By using extra set of parentheses we can tell java to use the object returned and access a method. (**new** Random()) represents the new object and nextInt(100) is the method call with input 100.

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

If the user is not lucky enough to get 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.

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

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

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

(Exercise) Fill-in – Shop.java & Cheese.java

Everywhere you see comment to "Fill in Code" is where you need to add code to make this program behave correctly. If it says "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.

(Assessment) Level of Understanding

1. What does **this** refers to?
2. What should be the value of numCheese when the RunShop terminates (ie the println output)?
3. Give code to implement **public void** setName(String name) { … }

What to hand in

When you are done with this lab assignment, you are ready to submit your work. Make sure you have done the following ***before*** you press Submit:

- Include Answers to Question Set 7.2.3, 7.4.1, 7.7.2, & 7.7.4
- Include answers to questions (Q1-Q10)
- Include answers to Level of Understandings (1-3)
- Attach filled in Shop.java and Cheese.java
- List of Collaborators
