# **CS 1331**: Introduction to Object Oriented Programming

## **Homework 7** **Due:** March 14, 2014 @ 8:00 PM

## Grocery Store!

In this assignment, you will be creating an application simulating a grocery store list. Users will be able to add custom food items or pre-made items from the Deli to their cart. They'll also have a constantly updated view of the list of items in the cart, subtotal (pre-tax), as well as the total cost (post-tax).

While a GUI would likely make sense for such a thing, we decided to do this HW all in the console, just to focus on some of the Object Oriented concepts it involves. Here is a sample of what a run of the final program would look like:

```
Welcome to the Grocery Store!

You have 0/10 items in your cart
Subtotal: $0.00 Total (with tax): $0.00

Press [1] for the Deli, or [2] for a custom item or 'q' to quit
2
Enter the name of your item: Potatoes
Enter the price (pre-tax) of your item: $4.00

You have 1/10 items in your cart
Potatoes cost $4.00
Subtotal: $4.00 Total (with tax): $4.40

Press [1] for the Deli, or [2] for a custom item or 'q' to quit
1
What item would you like to buy?
[1] Chicken
[2] Beef
[3] Cheese
3

You have 2/10 items in your cart
Potatoes cost $4.00
Cheese cost $2.00

Subtotal: $6.00 Total (with tax) $6.60

Press [1] for the Deli, or [2] for a custom item or 'q' to quit
q
Your cart has been cleared. Enjoy your groceries!
```

## `Purchasable` Interface

The first thing you should build is the `Purchasable` Interface. We'll be building classes that implement and use `Purchasable` in later sections, so it's important to get this done first!

**Constants**

Most things that get purchased come with some sales tax. So this interface should have a constant for the sales tax that can be used later on. This should be assigned to some reasonable amount (Georgia's sales tax is 4%, California's is 7.25%). The visibility of this should allow it to be accessed from other classes.

**Abstract Methods**

Generic things that are `Purchasable` should be able to perform 3 different tasks. Therefore, methods for these tasks should be declared in `Purchasable`.

- `public String getName()` to get the name of the item
- `public double getPrice()` to get the pretax price of the item
- `public double getAfterTaxPrice()` to get the after tax price of the item

## `GroceryItem` Class

This class is going to be used to create the custom grocery entries. It should implement `Purchasable` and define its methods.

**Constants**

We already have a sales tax constant in `Purchasable`, but in most states the food tax is different than the overall sales tax. So in `GroceryItem`, we need another constant for the food tax. Again, pick a reasonable value for this (most states' food tax is 0%, but Tennessee's is 5.5%), and make it have the proper visibility.

**Instance Variables**

Each `GroceryItem` only has a name and price as instance variables. These should be assigned in the constructor, which should take in parameters matching up to these variables.

**Instance Methods**

`GroceryItem` should implement all of the methods declared in `Purchasable`. You might notice that two of these methods will match up as getters for `GroceryItem's` instance variables, and should be implemented as such.

The other method, `getAfterTaxPrice()`, should return the price of the `GroceryItem` plus the tax on the item using the food tax constant.

The last instance method `GroceryItem` should have is a `toString()` method that returns a `String` with the name and price of the item. The price should be formatted using `NumberFormat` or `DecimalFormat`. Two possibilities of how this could look is `Hamburger costs $5.50` or `Hamburger: $5.50`.

## `DeliFood` Class

`DeliFood` is going to be quite similar to `GroceryItem` except in Deli foods are pre-prepared. It should implement `Purchasable` and also provide ways of making specific types of Deli foods.

### Instance Variables

`Delifood` will have a name and a price like `GroceryItem`

### Constructor

The constructor for `DeliFood` is fairly straightforward; it should take in two variables for the name and price, and assign those to its instance variables. However, we want our `DeliFood` objects to be prebuilt and not allow other classes to create new ones. So, we will provide static methods that allows other classes to pull out predetermined `DeliFood` objects. The constructor should have the proper visibility that allows for this behavior (Hint: What is something we can do to the constructor to make sure no one can make new objects from this class?)

### Instance Methods

Again, `DeliFood` should implement all of the methods from `Purchasable`. This time, `getAfterTaxPrice()`, should use the general sales tax from `Purchasable` to calculate the tax on the item. It should also have a `toString()` method that returns a String with the name and price similar to GroceryItem's `toString()` method, except it should have "Premade" prepended to the name. (e.g, "Potatoes" becomes "Premade Potatoes").

### Static Methods

`DeliFood` should have **at least three** class (static) methods that create and return different `DeliFood` objects. These will allow other classes to get their own (but predefined) `DeliFood` objects.

The naming convention for these methods should be of the form `create`*FoodName()* where FoodName is replaced by whatever the name of the `DeliFood` getting returned. (e.g, The method that returns a "Potato Salad" DeliFood would be `createPotatoSalad()`).

You are free to make these be any type of food and price. (In the examples, the 3 foods are PB+J, Potato Salad, and Beet Stew.)

## `ShoppingCart` Class

`ShoppingCart` objects are going to be able to hold and perform calculations on `Purchasable` items (like `GroceryItem` and `DeliFood`). By making it use `Purchasable` items, we can make even more classes that implement `Purchasable` in wildly different ways, and still add them to a `ShoppingCart` object.

### Instance Variables

This class should have only two instance variables: an **array** of `Purchasable` objects, and a variable that keeps track of the number of items in the cart.

### Constructor

The constructor for `ShoppingCart` should take in an int that describes the maximum capacity of the shopping cart. The array of `Purchasables` should be initialized to be this size. And the number of items in the cart should start off at 0. **Note: We will NOT go over the length of the shopping cart in this assignment. We'll save that challenge for another day :)**

### Instance Methods

`ShoppingCart` will have *many* instance methods that aid in adding `Purchasable` objects and getting information about the items that are in the cart.

- `addItem(Purchasable item)` takes in an item that we want to add to the shopping cart and returns whether or not we were successful at adding the item. Since arrays don't have a magical "add" method, this method will have to figure out where to put the item we want to add and whether or not there's a slot for it. We can check if the number of items in the cart is already the maximum capacity of the array; if it's full we cannot add the item, but if not we can add the item at the next available slot (we can use the number of items already in it to figure out where).
- `emptyCart()` should clear out the cart and set its state to how it was after its initial creation.
- `subtotal()` should return the total (before-tax) cost of all of the items in the cart. This should go through the array, adding each item's price up, and returns that overall cost.
- `total()` is very similar to `subtotal()`, but returns the total after-tax cost of all the items. Note that this method shouldn't be doing any tax calculations, but should use `getAfterTaxPrice()` from each item.
- `isFull()` should return whether or not the number of items in the cart fills the entire array or not. If it does, this method should return `true`, otherwise `false`.
- `isEmpty()` should return whether or not the cart is empty, i.e is the number of items equal to zero. If so return `true`, otherwise `false`.
- `getMaxItems()` should return the maximum number of items that this cart can hold. Note that we don't have an instance variable that simply tells us this, but we can use one of our instance variables to get this information. **Note: We don't *want* you to make an instance variable for this. Think about how to get this value with the information you already have**

- `toString()` should return a String with a list of all of the items in the cart, with each item separated by some character. (The examples use a new line character, but any distinct character will work.)
- This class should have a getter for the number of items in the cart. Remember to adhere to conventions when creating this.

## `GroceryStore` Class

This class will act as your driver for this program. It should contain the `main` method, and a sample output for your program is at the beginning of this document.

**NOTE: When the user quits the program, their cart should be emptied.**

### General Tips

- This is your biggest assignment to date. You should be writing out how all of your classes and interface interact with each other before you start coding.
- Furthermore you should outline what your classes and methods look like before coding them.
- When asking TAs for help on this assignment, you should reference your outlines and drawings of how you want to lay out your classes and how you expect them to interact with each other
- Remember when actually coding to use the debugging tools at your disposal. When asking a TA for help, walk them through the debugging process you've already gone through to give us a better idea of where you are stuck.

## Javadocs

We are going to have you do Javadocs again for this assignment (and for all assignments here on out). The samples from the last HW writeup are below for your reference.

```
import java.util.Scanner;

/**
* This class represents a Dog object.
* @author Ethan Shernan
* @version 1.0
*/

public class Dog {
....
}

/**
* This method takes in two ints and returns their sum
* @param a, b
* @return their sum
*/

public int add(int a, int b)){
...
}
```

## Checkstyle

Just in case you forget how to do Checkstyle, here are the instructions again!

You have been provided with a Checkstyle Jar file and the CS 1331 Checkstyle configuration file (`cs1331-checkstyle.xml`). We do this to make sure you are following the proper style guidelines as posted in the Style Guide under T-Square resources. Please refer to these guidelines for any help as you are writing your programs.

We will run checkstyle on all the Java source files you submit. You can run Checkstyle with the command below (in the directory containing all your Java source files):

```
$ java -jar checkstyle-5.6-all.jar -c cs1331-checkstyle.xml *.java
Starting audit...
Audit done.
```

The message above means there were no Checkstyle errors. You can easily count Checkstyle errors by using the command below and subtracting 2 from the number printed (which is how we will deduct points). For example:

```
$ java -jar checkstyle-5.6-all.jar -c cs1331-checkstyle.xml *.java | wc -l
2
```

Also means no errors.

If you have any questions about a Checkstyle error, please let a TA know! They should be pretty self-explanatory and you should be able to fix them easily. We will be taking points off for checkstyle on this assignment, so please make sure you run it!

# Turn-in Procedure

Submit all of the Java source files you created to T-Square. Do not submit any compiled bytecode (`.class` files). When you're ready, double-check that you have submitted and not just saved a draft.

**Please remember to run your code with Checkstyle!**

**Verify the Success of Your Submission to T-Square**
Practice safe submission! Verify that your HW files were truly submitted correctly, the upload was successful, and that the files compile and run. It is solely your responsibility to turn in your homework and practice this safe submission safeguard.

1. After uploading the files to T-Square you should receive an email from T-Square listing the names of the files that were uploaded and received. If you do not get the confirmation email almost immediately, something is wrong with your HW submission and/or your email. Even receiving the email does not guarantee that you turned in exactly what you intended.
2. After submitting the files to T-Square, return to the Assignment menu option and this homework. It should show the submitted files.
3. Download copies of your submitted files from the T-Square Assignment page placing them in a new folder.
4. Recompile and test those exact files.
5. This helps guard against a few things.
   (a) It helps insure that you turn in the correct files.
   (b) It helps you realize if you omit a file or files.[1] (If you do discover that you omitted a file, submit all of your files again, not just the missing one.)
   (c) Helps find last minute causes of files not compiling and/or running.

[1]Missing files will not be given any credit, and non-compiling homework solutions will receive few to zero points. Also recall that late homework will not be accepted regardless of excuse. Treat the due date with respect. The real due date is 8PM Friday. Do not wait until the last minute!