Designing with Abstraction

Purpose:

The purpose of this lab is to practice using abstraction and polymorphism in class design.

Set up:

Create a directory named `Lab27/inventory` in your `Java` directory. Copy all the files from `~labCourse/Labs/Lab27/invnetory` to your `inventory` directory.

Base classes:

The root class of a class hierarchy is called a `base class`. In a well designed hierarchy, the root class and any intermediate classes are `abstract`. Concrete classes occur only as “leafs” and are not extended.

![Class Hierarchy Diagram]

Problem:

Assume that a retail shop wants to classify items into three categories: regular retail items, salvaged items, and hot items. Salvaged items are items that the shop wants to get rid of at a very cheap price. The items may be damaged or discontinued. The retail price of these items is 20% of the gross price. Hot items are items in high demand and hard to restock. The retail price for these items is 10% over the gross price. Any additional discount is applied to the modified retail price. Taxes are applied to the discounted price. Any other item is considered to be a regular item.

A first solution:

A first attempt to build a solution is to start with our old class `RetailItem`. We can use this class as the basis for a hierarchy of retail items, where `RetailItem` will represent regular items. You can read the specifications of that class here.

To deal with the other categories, we can define two subclasses of `RetailItem`: `SalvagedItem`, and `HotItem`.

- Do you need to specify new methods for `SalvagedItem` or `HotItem`? If so, what are they?
• Do you need to override methods? If so, which ones?
• Is the structure of this first solution good? Why or why not?

A second solution:

Since we can have three kinds of retail items, we can use RetailItem as originally specified (here) as basis for defining an abstract class. We now call this abstract class RetailItem, and use this class to derive three concrete classes: RegularItem, SalvagedItem, HotItem.

The abstract class RetailItem:

We provide a specification and implementation of the class RetailItem as we have used it in previous labs.

• Using the class RetailItem provided, modify it to be an abstract class intended to be used as the root of a hierarchy. This abstract class provides all the functionality as the original RetailItem. Default implementations of all the methods must be given in this case.
• This class contains a constructor. What should the access modifier for the constructor be?
• Save and compile the class.

The class RegularItem:

This class is specified as a subclass of RetailItem. It must provide its own constructor, which is implemented by calling super:

    public RegularItem (int code, double price, double disc,
                        double taxRate, int onHand) {
        super(code, price, disc, taxRate, onHand);
    }

• Why must this class define a constructor?

The class RetailItem provides a toString method. This method should be overridden to extend the information the root class provides. Here, we simply invoke the method as implemented in the super class

    super.toString()

and append two more pieces of information to the resulting String: the classification of the item (“Regular”) and the net price of the item.

• Do we need to override any other methods?
• Specify and implement the class RegularItem.
• Compile class to insure that your implementation is syntactically correct.

The class SalvagedItem:

This class is specified as a subclass of RetailItem.
• The method `netPrice` must be overridden. Why?

As with `RegularItem`, a constructor must be provided and the method `toString` must be overridden.

• Implement and compile this class.

**The class `HotItem`:**

This class is implemented in a manner similar to `SalvagedItem`.

• Implement and compile this class.

At this point you should have four classes in your package.

• Draw a static diagram showing the hierarchical structure of these classes.

• Is this hierarchy well designed?

• Is this solution better that the first solution suggested?

**Forecast analysis:**

We want to determine the income from selling all of the store’s inventory.

• Look at the definition of the class `Inventory`. Can the inventory contain an instance of `RetailItem`? Why or why not?

• Can the inventory contain an instance of `RegularItem`? Of `SalvagedItem`? Of `HotItem`? Why or why not?

For forecasting, we can use the `Inventory` query `expectedIncome`:

```java
public double expectedIncome ()
    The total net price (including taxes) of items in this `Inventory`.
```

• Read the implementation of this query carefully. Will it compute the correct value if the `Inventory` contains several `RegularItem`, `SalvagedItem`, and `HotItem` instances?

**Testing:**

We want to test an inventory containing `RegularItem`, `SalvagedItem`, and `HotItem` instances.

• What type should the list be so that it can contain `RegularItem`, `SalvagedItem`, and `HotItem` instances?

**Data file format:**

As in a previous lab, we will read data from a file to initialize the inventory. The file will contain one line per item, formatted as in the previous lab: item code, formatted as an integer literal; price, discount, and tax, formatted as double literals; quantity, formatted as an integer literal. We must also specify the classification of the retail item. To do this, each line will end with a character indicating the classification:

```java
    r    regular item;
```
For example:

100 100.0 0.0 0.0 100 r
200 200.0 0.0 0.0 10 s
300 300.0 0.0 0.0 100 h
110 100.0 0.0 0.0 100 s
120 100.0 0.0 0.0 100 h
130 100.0 0.0 0.0 100 r

- Complete the implementation of Lab27.inventory.RetailItemDB. You must modify two segments of code.
  
The method `readRetailItemList` must read the last character, and use this to determine what kind of retail item to create.

  The method `writeRetailItemList` must append an appropriate character to the end of the line, indicating the kind of retail item.

- Use Lab27.inventory.InventoryTest to test your implementation.

**Post-lab:**

Submit the following, as directed by your lab instructor:

- implementations of classes `RetailItem`, `RegularItem`, `SalvagedItem`, and `HotItem`;
- implementation of the class `RetailItemDB`;
- test plan for the modified `Inventory`: your plan should create several different files testing different cases;
- script listings showing the results of your tests.