Homework 06 – Grocery Store

Problem Description

Hello! Please make sure to read all parts of this document carefully.

You have been asked by a Supermarket to simulate how they store physical goods such as food and supplies. It wouldn't be a grocery store without order either, so you will oversee keeping the supplies sorted as you go adding supplies to the catalogue. You are asked to create a Describable.java, GroceryItem.java, Grocery.java, Food.java, and Supply.java as the composition of the simulation.

Remember to test for Checkstyle Errors and include Javadoc Comments!

Solution Description

You will create one interface, one abstract class, and two concrete subclasses. You will be creating a few fields and methods for each file. Based on the description given for each variable and method, you will have to decide whether the variables/method should be static, and whether it should be private or public. To make these decisions, you should carefully follow the guidelines on these keywords as taught in the modules.

Hint: A lot of the code you will write for this assignment can be reused. Try to think of what keywords you can use that will help you! You should be able to put @Override on the line before the method header for any methods you override.

For all classes, include getters and setters when applicable.

Describable.java

This file defines an **interface** for objects that can be described.

Methods

- describe()

o Abstract method that does not take in anything and returns a description of the object as a String

> o (Note: any concrete class that implements Describable must provide a method definition for this method)

GroceryItem.java

This class represents a grocery item. You must not be able to create an instance of this class (**Hint:** there is a keyword that prevents us from creating instances of a class). **GroceryItem should implement the `Comparable` interface with the proper type parameter. All GroceryItems should also be `Describable`, but the implementation of the abstract method will be done only in the child classes.**

Variables

- `String name` – The name of the item
- `int itemCode` – This represents a code given to the item
- `double price` – This represents a price of the item and will be used to compare and sort items

Constructors

- A constructor that takes in the name, itemCode, and price in that order and sets the instance variables accordingly

Methods

Do not create any other methods than those specified. Any extra methods will result in point deductions. All methods must have the proper visibility to be used where it is specified they are used.

- compareTo(GroceryItem other)
  - Override the compareTo method (You should be able to put @Override on the line before the method header)
  - Takes in a GroceryItem object and returns an int, adhering to the API contract (Comparable Interface) (This will only work if you have the proper generic type in the class header)
  - A GroceryItem is greater than another GroceryItem if its price is less than the other ones. For example, if GroceryItem itemOne has price = 8 and GroceryItem itemTwo has price = 5, itemOne is less than itemTwo. (Logically, this will allow us to sort from highest to lowest price).
- toString()
  - Returns a String describing the GroceryItem as follows:
    - [itemCode]: [description]
      - Note, this is a concatenation of the itemCode and output from describe()
      - e.g.
      - 80: This is a description.
        - Descriptions are handled in child classes
  - Must override toString() method defined in Object class

Grocery.java

This file defines a class called Grocery.

Variables

- GroceryItem[] shelf - An array that simulates a shelf where we will hold objects of type GroceryItem. This array should not contain any null values for input. You don't have to check since we'll only input non-null values and full arrays but keep that in mind for your own testing. This array should always be sorted from highest price to lowest price.

Constructors

- A zero-argument constructor that initializes an empty shelf (size zero).
- A one argument constructor that takes in an array of GroceryItems that will be the items on the shelf. Remember that we want the shelf to be always sorted, and that you should not assume that the passed in array is sorted (there's an easy method for this operation)

Methods

- browseItems()
  - This method should print out the information of all GroceryItems in the shelf for the Grocery.
  - Format for the print statements: "[itemCode]: [description]" on a new line
  - Remember to disregard the brackets
  - The method should return nothing.
  - **Hint:** Think of what method could be useful to get the printed values
- addGroceryItem(GroceryItem newItem)
  - This method should create a new array of length + 1 that copies over all the previous elements as well as adds the new item. (Note: For Arrays.sort() to work, the array must not have any null values, which is why we keep it full)
  - It should then sort the array to maintain the shelf's order.
  - The method should return nothing.
- getGroceryItem(int code)
  - Returns the grocery item with given code. If the code doesn't exist, return null.

- o   If there are `GroceryItems` with the same code, return the first one found with the code.
- o   The method should return a `GroceryItem`
- getNumberOfItems()
  - o   Returns the number of items on the shelf
  - o   The method should return an int

Food.java

This file defines Food. **Have Food extend the GroceryItem class.**

Variables

All variables must be not allowed to be directly modified outside the class in which they are declared, unless otherwise stated in the description of the variable. **Hint**: there is a specific visibility modifier that can do this!

The Food class must have these variables. Do **NOT** re-declare any of the instance variables declared in GroceryItem class:

- `age` - an int that represents how old the Food is in months
- `expiration` - an int that represents how many months it takes for this Food to expire

Constructors

- A constructor that takes in the name, itemCode, price, age, and expiration and sets all fields accordingly. It must accept the variables in the specified order.

Methods

Do not create any other methods than those specified. Any extra methods will result in point deductions. All methods must have the proper visibility to be used where it is specified they are used.

- `calculateMonthsRemaining()`
  - o   Return the difference between expiration and age
    - ▪   Note, this is positive if the age is less than expiration, negative if it's greater
- `describe()`
  - o   Return "The food is [age] months old and has [remaining months value] months remaining before it expires."
    - ▪   Note, the brackets should not be included in the returned String

Supply.java

This file defines a supply item. A supply item is defined to be a non-food item that is sold at grocery stores, such as cleaning supplies, cards, etc. **Have Supply extend the GroceryItem class.**

Variables

All variables must not be allowed to be directly modified outside the class in which they are declared, unless otherwise stated in the description of the variable. **Hint**: there is a specific visibility modifier that can do this!

- `blurb` - A string that represents a short description of the supply item.

Constructors

- A constructor that takes in the name, itemCode, price, description and sets all fields accordingly. It must accept the variables in the specified order.

Methods

Do not create any other methods than those specified. Any extra methods will result in point deductions. All methods must have the proper visibility to be used where it is specified they are used.

- `describe()`
  - Return "The item looks like [short description]." without the brackets
    - Note, the brackets should not be included in the returned String

Reuse your code when possible. Certain methods can be reused using certain keywords. If you use the `@Override` tag for a method, you will not have to write the Javadocs for that method. These tests and the ones on Gradescope are by no means comprehensive so be sure to create your own!

Rubric

[4] Describable.java

- o [1] Correct interface header

- o [3] Correctly implements specified methods

- o [3] Correct describe() method

[13] GroceryItem.java

- o [2] Correct class header

- o [2] Correct instance variables

- o [3] Correctly implements constructors

- o [6] Correctly implements specified methods

- o [6] Correct compareTo(GroceryItem other)

[50] Grocery.java

- o [2] Correct instance variables

- o [5] Correctly implements constructors

- o [43] Correctly implements specified methods

- o [13] browseGroceryItems()

- o [12] addGroceryItem(GroceryItem newItem)

- o [10] getGroceryItem(int code)

- o [8] getNumberOfItems()

[20] Food.java

- o [1] Correct class header

- o [2] Correct instance variable(s)

- o [5] Correctly creates constructors

- o [12] Correctly implements specified methods

- o [6] Correctly implements and overrides describe()
- o [6] Correct calculateMonthsRemaining()

[13] Supply.java

- o [1] Correct class header
- o [1] Correct instance variables
- o [5] Correctly creates constructors
- o [6] Correctly implements specified methods
- o [6] Correctly implements and overrides describe()

We reserve the right to adjust the rubric, but this is typically only done for correcting mistakes.

Allowed Imports

To prevent trivialization of the assignment, you are only allowed to import `java.util.Arrays`. You are *not* allowed to import any other classes or packages.

Feature Restriction

There are a few features and methods in Java that overly simplify the concepts we are trying to teach or break our auto grader. For that reason, do not use any of the following in your final submission:

- `var` (the reserved keyword)
- `System.exit`

Javadoc

For this assignment, you will be commenting your code with Javadoc. Javadoc comments are a clean and useful way to document your code's functionality. For more information on what Javadoc comments are and why they are awesome, the online overview for them is extremely detailed and helpful.

You can generate the Javadoc overview for your code using the command below, which will put all the files into a folder named "javadoc". Note you should execute this after adding Javadoc comments to your code:

```
$ javadoc *.java -d javadoc
```

The relevant tags that you need to include are @author, @version, @param, and @return. Here is an example of a properly Javadoc'd class:

```
import java.util.Scanner;


/**
 *This class represents a Dog object.
 *@author George P.Burdell
 *@version 1.0
 */
```

```
public class Dog {

    /**
     *Creates an awesome dog(NOT a dawg!)
     */
    public Dog() {
    ...
    }


    /**
     *This method takes in two ints and returns their sum
     *@param a first number
     *@param b second number
     *@return sum of a and b
     */
    public int add(int a,int b) {
    ...
    }
}
```

A more thorough tutorial for Javadoc Comments can be found [here](#).

Take note of a few things:

1. Javadoc comments begin with /** and end with */.

2. Every class you write must be Javadoc'd and the @author and @verion tag included. The comments for a class should start with a brief description of the role of the class in your program.

3. Every non-private method you write must be Javadoc'd and the @param tag included for every method parameter. The format for an @param tag is @param <name of parameter as written in method header> <description of parameter>. If the method has a non-void return type, include the @return tag which should have a simple description of what the method returns, semantically.

Checkstyle can check for Javadoc comments using the -a flag, as described in the next section.

Checkstyle

For this assignment, we will be enforcing style guidelines with Checkstyle, a program we use to check Java style.

To run Checkstyle (both Checkstyle and Javadocs), put the jar file in the same folder as your homework files and run

```
java -jar checkstyle-6.2.2.jar -a *.java
```

The Checkstyle cap for this assignment is **20 points**. This means that up to 20 points can be lost from Checkstyle errors. Javadoc errors will be counted as Checkstyle errors.

Collaboration

Collaboration Statement

To ensure that you acknowledge a collaboration and give credit where credit is due, **we require that you place a collaboration statement as a comment at the top of at least one .java file that you submit**. That collaboration statement should say either:

I worked on the homework assignment alone, using only course materials.

or

In order to help learn course concepts, I worked on the homework with [give the names of the people you worked with], discussed homework topics and issues with [provide names of people], and/or consulted related material that can be found at [cite any other materials not provided as course materials for CS 1331 that assisted your learning].

Allowed Collaboration

When completing homeworks for CS1331 you may talk with other students about:

- What general strategies or algorithms you used to solve problems in the homeworks
- Parts of the homework you are unsure of and need more explanation
- Online resources that helped you find a solution
- Key course concepts and Java language features used in your solution

You may **not** discuss, show, or share by other means the specifics of your code, including screenshots, file sharing, or showing someone else the code on your computer, or use code shared by others.

Examples of approved/disapproved collaboration:

- **approved**: "Hey, I'm really confused on how we are supposed to implement this part of the homework. What strategies/resources did you use to solve it?"
- **disapproved**: "Hey, it's 10:40 on Thursday... Can I see your code? I won't copy it directly I promise"

In addition to the above rules, note that it is not allowed to upload your code to any sort of public repository. This could be considered an Honor Code violation, even if it is after the homework is due.

Turn-In Procedure

Submission

To submit, upload the files listed below to the corresponding assignment on Gradescope:

- Describable.java
- Grocery.java
- GroceryItem.java
- Food.java
- Supply.java

Make sure you see the message stating "HW06 submitted successfully". From this point, Gradescope will run a basic autograder on your submission as discussed in the next section.

You can submit as many times as you want before the deadline, so feel free to resubmit as you make substantial progress on the homework. We will only grade your last submission: be sure to **submit every file each time you resubmit**.

Gradescope Autograder

For each submission, you will be able to see the results of a few basic test cases on your code. Each test typically corresponds to a rubric item, and the score returned represents the performance of your code on those rubric items only. If you fail a test, you can look at the output to determine what went wrong and resubmit once you have fixed the issue.

The Gradescope tests serve two main purposes:

- Prevent upload mistakes (e.g. non-compiling code)
- Provide basic formatting and usage validation

In other words, the test cases on Gradescope are by no means comprehensive. Be sure to thoroughly test your code by considering edge cases and writing your own test files. You also should avoid using Gradescope to compile, run, or checkstyle your code; you can do that locally on your machine.

Other portions of your assignment can also be graded by a TA once the submission deadline has passed, so the output on Gradescope may not necessarily reflect your grade for the assignment.

Important Notes (Don't Skip)

- Non-compiling files will receive a 0 for all associated rubric items
- Do not submit `.class` files
- Test your code in addition to the basic checks on Gradescope
- Submit every file each time you resubmit
- Read the "Allowed Imports" and "Restricted Features" to avoid losing points
- Check on Piazza for a note containing all official clarifications