Homework 08 – Holiday Shopping

Problem Description

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

Holiday season is beginning to kick off, and you're in the mood for even more shopping (at home)! Halloween and Thanksgiving are currently in your sights, and you have two separate shopping lists prepared. You are (virtually and safely) headed off to a Halloween-Thanksgiving specialty store and will be purchasing the items. However, you find that your shopping list has some errors! Oh no! In this assignment, you are provided three comma-separated values (csv) files: halloween.csv, thanksgiving.csv, and store_inventory.csv. You will create **Store.java** to represent the store and create three exceptions: **OutOfStockException.java, InvalidDataException.java, and InvalidFormatException.java**. Let's get shopping!

To complete this assignment, you will use your knowledge of exceptions and file I/O.

Remember to test for Checkstyle Errors and include Javadoc Comments!

Solution Description

Provided Files

`halloween.csv` – A comma separated values (csv) file in the shopping list format representing what we want to buy for Halloween with no incorrect data entries

`thanksgiving.csv` – A comma separated values (csv) file in the shopping list format representing what we want to buy for Thanksgiving with some incorrect data entries

`store_inventory.csv` – A comma separated values (csv) file in the store inventory format representing an example store

Note: The commas separate value (CSV) files you will be interacting with will follow three main formats:

- Shopping lists should have the csv line format of **ItemName,MaxPrice**
    - To signify that a shopping list was intentionally created it will have a magic number "$hopping" as the file header on its own line. Check the provided halloween.csv and thanksgiving.csv files for examples.
- Store Inventory should have the csv line format of **ItemName,Stock,Price** (you can assume that store inventory files will always be correctly formatted)
- Receipts should have the csv line format of **ItemName,priceBought**
    - At very end, there should be one csv line with format **TOTAL,totalPrice** where TOTAL is the String literal TOTAL
    - You will be the one generating Receipt files

Store.java

This class represents a store and contains methods to open and parse csv files. As items are purchased, they will be added to a receipt and saved to a csv. We highly recommend that you **test methods independently and incrementally**. (I.e. see if your constructor is properly populating inventory and that your buyItem and checkLine

throw the right Exceptions given invalid input) Also, remember to **use the other methods** that you have created in the class!

Variables

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 Store class must have these variables.

- `String[][] inventory` – a 2d array that stores the name, stock, and price of each item
    - Note: each row will represent and item: [[Turkey, 20, 10], [Costume, 4, 15]]
    - Note: inventory will contain unique items

Constructors

- A constructor that accepts a File object of the store inventory file as a parameter
    - File that contains store inventory in the format "[itemName],[stock],[price]"
        - Each item has its own line
        - See example file "store_inventory.csv" for the format
        - You can assume that the inventory will be in the correct format every time and that there will only be unique items (no two "phone"s)
    - Parses and assigns the data to `inventory`, a 2D String array
    - Hint: Go through the entire File before initializing the 2D String array to find the length
    - Hint: String.split() will be very useful when dealing with comma separated values
    - Hint: The Scanner constructor that accepts a File object potentially throws a FileNotFoundException which is a checked exception
- A constructor that accepts a String of the file name (including the file extension) of the store inventory as a parameter
    - Chains to the other constructor
    - Hint: This should be done in 1-2 lines

Methods

Create helper methods as necessary. All methods must have the proper visibility to be used where it is specified they are used.

- `public void goShopping(File shoppingList, String receiptDestination)`
    - In this universe, a "shopping list" is a predefined data format where each file starts with the magic number of "$hopping" on its own line
        - Throw an `InvalidFormatException` if the header is not "$hopping" on its own line
        - This should be thrown to whatever is calling `goShopping` to handle (since it's a checked Exception, they will always have to deal with it somehow)
        - Note: Because this method has the possibility of throwing a checked exception, you must alter the method declaration
    - Initialize a `PrintWriter` with `receiptDestination` as the file name to write to (Hint: only initialize one `PrintWriter`)
    - Call `buyItem` on each line in the `shoppingList` file, keeping track of the cumulative total
        - If a `FileNotFoundException` is thrown when initializing the `Scanner`, then catch it and print the exception (its toString, not the message) to System.out
        - If `buyItem` throws an `OutOfStockException`, catch it and print the exception (its toString, not the message) to System.out

    - After going through all lines in `shoppingList` file, write the cumulative total into the last row of the receipt csv in the format "TOTAL,[cumulative total]".
    - Close the `PrintWriter`
- `private int buyItem(String shoppingListItem, PrintWriter receiptWriter)`
    - Call `parseLine` to parse the `shoppingListItem` and check for data errors
        - If `parseLine` throws an `InvalidDataException`, then catch it and print the exception (its toString, not the message) to System.out and return -1
    - If the item is found but doesn't have stock, throw a new `OutOfStockException` with the message "[item name] is not in stock!"
    - Otherwise, If the item is not in the store array or if the store price is higher than the max price in the shopping list, the item is not bought and return –1
    - Otherwise, if the item is successfully purchased:
        - Reduce stock by one in the inventory array
        - Pass in valid item name and price to the into `PrintWriter` to write into the receipt csv
            - This must be in the format "[name of item],[bought price of item]\n"
        - Return the bought-for price (which is the store price)
- `private String[] parseLine(String line)`
    - The line should come in the format "[item name],[max price]"
    - Check the line for data errors (note: these will be the only types of data errors that you will be expected to handle and there will still be a comma in every line)
        - If it is an empty item name, throw a new `InvalidDataException` with the message "Item has no name"
        - Otherwise, if the max price cannot be parsed as an int, throw a new `InvalidDataException` with the message String inside as: max price + "cannot be parsed as an integer price"
        - Otherwise, if the max price is negative, throw a new `InvalidDataException` with the message "Max price cannot be negative"
    - Otherwise, create a new size 2 String[] with the format {itemName, maxPrice} and return it

`OutOfStockException.java`

This class will contain our Exception that we will throw when the stock of the item is 0. This Exception should be unchecked (hint: think about what it should extend!)

Constructors

- A constructor that takes in a String and call the superclass's constructor passing in the String
- A constructor that takes nothing and calls the super method "There is no stock remaining!" to the super constructor.
    - Hint: There is a specified keyword in L12 to access the superclass's constructor.

`InvalidDataException.java`

This class will contain our Exception that we will throw we have a negative max price or an empty field. This Exception should be unchecked (hint: think about what it should extend!)

Constructors

- A constructor that takes in a String and call the superclass's constructor passing in the String

- A constructor that takes nothing and just has the super call
    - Hint: There is a specified keyword in L12 to access the superclass's constructor.

`InvalidFormatException.java`

This class will contain our Exception that we will throw if the first line of a ShoppingList csv does properly have the magic number. This Exception should be checked since we want relevant programs to explicitly acknowledge (by either throwing in the header or catching it) a potential invalid format (hint: think about what it should extend!)

Constructors

- A constructor that takes in a String and call the superclass's constructor passing in the String
- A constructor that takes nothing and just has the super call
    - Hint: There is a specified keyword in L12 to access the superclass's constructor.

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

[70] `Store.java`

- [5] Correct instance variables
- [15] Correctly creates constructors
- [50] Correctly implements specified methods
    - [20] goShopping()
    - [15] buyItem()
    - [15] parseLine()

[10] `OutOfStockException.java`

- [5] Both constructors are correct
- [5] Exception is an unchecked exception

[10] `InvalidDataException.java`

- [5] Both constructors are correct
- [5] Exception is an unchecked exception

[10] `InvalidFormatException.java`

- [5] Both constructors are correct
- [5] Exception is a checked exception

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 the following classes. You are *not* allowed to import any other classes or packages.

- `java.util.Scanner`
- `java.io.File`
- `java.io.FileNotFoundException`
- `java.io.IOException`
- `java.io.PrintWriter`

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. Checkstyle can be downloaded here.

To run Checkstyle, 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 **30 points**. This means that up to 30 points can be lost from 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 Gradecope:

- Store.java
- OutOfStockException.java
- InvalidDataException.java
- InvalidFormatException.java

Make sure you see the message stating "HW08 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