Vending Machine

Objectives:

1. Use the various programming tools:
    a. Java Collections ArrayLists and Queues
    b. File IO and Data Files
    c. Polymorphism
    d. ~~Overloading~~ (eh… not in Java, but….)
    e. Inheritance
    f. and eventually… JavaFX???
    g. use data structures within data structures
2. Have the student be active in creating test cases and scenarios
    a. create simple code for adding items to the vending machine
    b. create simple code to pulling items from the vending machine
    c. create simple code to test fringe cases that could check the validity of the data structure

Code given:

Driver.java **(UPDATED 11/28/22)**

This is literally the most basic/minimal driver file required for your project. Please notice there is one note at the bottom of that driver. Don't forget that I should also be able to add code to the end of the driver to manually remove or add items to the Vending machine. (Without using the data files). So manually.

Also, please use Java version 11 to create and complete this project.

Background:

The world is full of examples of logic and use of data structures. Everyone is familiar with vending machines. Loading a vending machine is straightforward with no tricks (at least in this project). What you might not know, behind the scenes, there is a very particular way of emptying vending machines.

Remembering that in retail, image is everything. Having an empty vending machine looks bad. To the customer, an empty vending machine may signal to them that:

a. there is not enough of the product, need to choose somewhere else
b. that machine is not well maintained
c. the machine could be broken
d. the contents inside might be old

Keeping the appearance that shelves of items are “full”, and only empty at the last resort is a consideration that the programmers of the vending machine are tasked with. For example:

Base Example
from: https://www.mentalfloss.com/article/57164/why-dont-drinks-outdoor-vending-machines-freeze

Notice from the base example that those drinks that are more popular appear in many "slots" within the vending machine. From the image above, Coca-Cola has three slots, Pepsi has two, and Red-Bull at least 4 (although from the picture it would appear to be five, but the last must be a different flavor). Also note, that in each slot, contains multiples of the same product. For example, starting with the uppermost left of Coca-Colas would appear to have 2 sodas, 4, then 1 respectfully moving right. For our sake, and explained more later, we will consider those same slots as [0], [1] and [2] in our program.

Here is where the logic steps in. Notice that there is one soda left for slot[2] or the third Coca-Cola. But there are other slots that contain more Coca-Colas and will not empty the slot if chosen. To avoid any empty slots, and only when about to empty a slot, even if the user selects slot[2], **the machine will pull from the largest similar product queue instead**.In this particular case, slot[1]. If there is a tie in length of similar product queues, our program logic will pull a product from the lowest similar item slot. If there were one Coca-Colas for every slot, it would pull from the lowest index, in this case slot[0]. Only if there is no other choice will it empty the slot selected.

While the logic is simple enough, it's getting it to work and setting up the vending machine is where most of your work will be.

Requirements:

For this project, there are some standards we will use for simplification.

File and Class(es) setup

a. there will be a Driver (Driver.java) that contains the main()
    i. this file will be replaced with our various versions to test your logic, code and setup when grading
    ii. the driver will call the DataFile class functions described below first, before anything dealing with the Vending Machine
b. there will be a DataFile (DataFile.java) class
    i. this class will help retrieve and organize our data and should be called early in the driver
    ii. contains a constructor that will accept a fileNames of the Directory and Input filenames to gather data from
    iii. contains a <ArrayList (String)> "loadDirectory" function

1. reads each line from “directory.txt” as a simple String, places that dat into a ArrayList of Strings to be parsed later in Vending Machine
2. should be completed first before setting up the Vending Machine class
3. will return an ArrayList of Strings from the directory.txt file to be used later

iv. contains a <ArrayList (Integer)> “loadSampleInput” function
1. reads from “input.txt” file that contains the selection of products from users
2. Choice selection will be an index number. Using the Base Example, if the user chooses 0, they will receive a Coca-Cola. If they select 4, they will receive a Pepsi.
3. will return an ArrayList of Integers from the input.txt file to be used later

v. example below

c. there will be an “Item” (Item.java) class
 i. it will contain name(String), calories (float) and **itemType** (String)
  1. **itemType** is ONLY to be for debugging, not the polymorphism
  2. you still need to do the polymorphism
 ii. this is the SUPER class to Snack and Drink below
 iii. the class will require a constructor, setter and getters for each data member. But will NOT have an toString(). Classes below will.

d. there will be a “Snack” (Snack.java) class
 i. this is the first time this class has been mentioned
 ii. this class will also inherit from Item
 iii. this will also enable our polymorphism since ***it is an*** Item
 iv. it will contain the data members weight(float) and containsNuts(boolean)
 v. the class will require a constructor, setter and getters for each data member and toString() to display items within that instance for debugging purposes

e. there will be a “Drink” (Drink.java) class
 i. this class will also inherit from Item
 ii. this will also enable our polymorphism since ***it is an*** Item

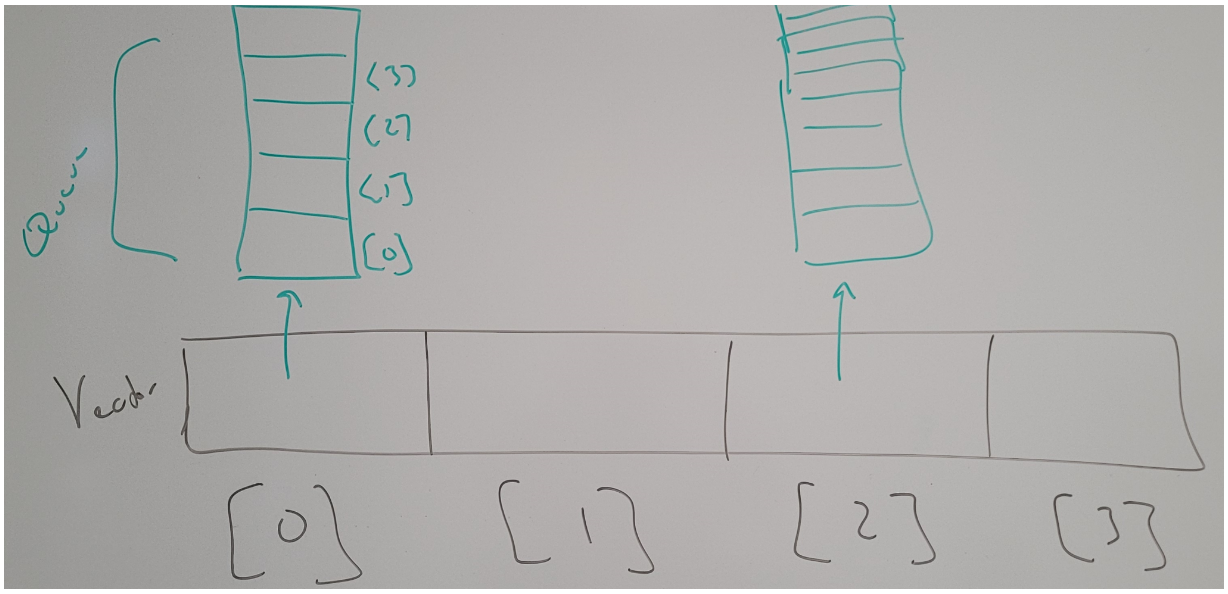
iii. it will contain the data members ounces(float) and type(String)
   1. type will be of these specifically: soda, water, other

iv. the class will require a constructor, setter and getters for each data member and toString() to display items within that instance for debugging purposes

f. there will be a “Vending” (Vending.java) class
   i. this is where everything comes together
   ii. it will contain two major data structures
      1. an Java Collections ArrayList called “directory” (Strings)
         a. This will be used to locate your product
         b. this will be a parallel ArrayList simply containing the string names of all the products.
         c. Again, using the base example, if would be:

Coca-Cola, Coca-Cola, Coca-Cola, Coca-Cola Zero, Pepsi, Pepsi, etc…

      2. an Java Collections ArrayList called “slots” that contain a QUEUE of your Items (Drinks or Snacks).
         a. the failure to use this data structure for for its intended purposes will

Visual Setup of Vending DataStructure


iii. Your visual (and debugging) display for each of the products will be the toString() for THIS class. **This is a part of the displayItems() function.** Your virtual vending machine will look as follows, using the base example and remembering we are only working with one row:

```
Coca-Cola: (Soda): 2
Coca-Cola: (Soda): 4
Coca-Cola: (Soda): 1
Chips: (Snack): 8
```

iv. will contains at LEAST these functions
   1. constructor
      a. will accept the <ArrayList (String)> created from the DataFile class that contains all the data the Vending Machine needs
      b. This same data will be passed to the “loadItem” function below
   2. void loadItem(<ArrayList> (String))

   a. used to populate the Vending Machine
   b. is will parse the data passed in by the <ArrayList (String)>
      i. parseData might be a suggested function for this
   c. use the data file "loadDirectory" (explained below) to physically set up the Vending Machine's directory and load Items into the respective slots.
   d. notice the return or parameters for this function are open for your interpretation

3. void unloadItem(int index)
   a. pull one product item from the slot
   b. this will need our logic check to avoid empty slots of a product
   c. notice the return or parameters for this function are **NOT open** for your interpretation
   d. this function will be called from the main in order to simulate user responses
   e. can/will be a debugging function as well
4. void unloadItem(ArrayList <Integer>)
   a. overload of above
   b. this will receive the data originally from input.txt, but via DataFile
   c. will then conduct unloading items possibly using the original unloadItem(int index) function
5. findProduct(String)
   a. function that returns ArrayList index number(s) that match the product
6. toString()
   a. would suggest using the ArrayList <Queue> toString() to have it display the result of the simulation

Directory File (directory.txt) Example:

```
Drink, Coca-Cola, 120, 16, soda, 2          ⇒ place in slot [0]
Drink, Coca-Cola, 120, 16, soda, 4          ⇒ place in slot [1]
Drink, Coca-Cola, 120, 16, soda, 1          ⇒ place in slot [2]
Snack, Chips, 220, 4, false, 8             ⇒ place in slot [3]
```

For Drink:

(<Item> itemType, <Item> name, <Item> calories, <Soda> ounces, <Soda> type, # of items in slot **VM** queue)

For Snack:

(<Item> itemType, <Item> name, <Item> calories, <Snack> weight, <Snack> containsNuts, # ***of items*** in slot **VM** queue)

Please note that each line in the data file would be ONE ArrayList<String>. So in the example above, you would have a vector/arraylist of 4 strings, each a line from the file. Parsing those strings for your VM data is next!!

Also note, there should be 2 Sodas in the slot vector/arraylist[0] queue after reading the directory file!!!!

LoadSampleInput File (input.txt) Example:

This file is simply the **selections** that users made. For example:

```
0
2
1
```

Would select products from slots [0], [1] and [2]. The data would reflect this and display:

```
Coca-Cola: (Soda): 1
Coca-Cola: (Soda): 2
Coca-Cola: (Soda): 1
Chips: (Snack): 8
```

Notice that using our non-emptying slot logic, a product was pulled from a different slot in order to maintain our objective.

Other Standards and clarifications:

1. Our vending machines will consist of only one row of items
2. Slots - means a queue where MULTIPLE items OF THE SAME PRODUCT can be placed in a queue
3. not every Item similar in product will be grouped in the data files. This means a Pepsi could be placed in slot[0] and then again in slot[6]
4. there could be some Items that do not have alternative slots
5. This project description contains mostly examples of Sodas as a Drink.
    a. You are welcome to create other drinks and snacks
    b. Getting this project to work solely with Drinks (and finishing other requirements) and no polymorphism will get you a passing grade. But adding and enabling the ability to mix Drinks and Snacks will get your project into a possible A grade category.
6. ~~Valgrind will be used to check for memory leaks~~

Creating Test Cases Extra Credit:

Having the students be actively involved in creating test cases, scenarios and solutions is important, but also rewarded. In general, creating a sample data set that contains:

1. In a sample Driver.java
    a. create simple code for adding items to the vending machine
    b. create simple code to pulling items from the vending machine
    c. create simple code to test fringe cases that could check the validity of the data structure
    d. call the sample data files
2. A set of complete data files

3. Expected output named “results.txt”

This effort will earn you a maximum of 3 points of credit on the project. (Can be extra credit, if you receive a 100 on the project).

<u>Be careful that your data and output are correct!!!</u>

Submission instructions:

You must submit the following files to the proj1 submit directory:

- Item.java
- Vending.java
- DataFile.java
- Drink.java
- **Driver.java (contains main())**
- Snack.java
- makefile
  - should contain ALL target command to run, clean and compile your project
- README.txt - Optional. Any notes you want to give the grader.

You will simply export your project into a ZIP called **P1-code.zip**, and submit through our Course Page.