CSCI 3901 Assignment 1

Due date: 11:59pm Thursday, September 19, 2024 in Brightspace

Problem 1

Goal

Get practice in decomposing a problem, creating a design for a program, and implementing and testing a program. Practice basic Java programming.

Background

The cost of living over the last few years has increased significantly. A rise in cost, as inflation, is an obvious sign of that increased cost of living. However, manufacturers may have also decreased the size of their products as less obvious cost increase; this latter cost is sometimes called “shrinkflation”.

You will be helping a consumer product agency report on the increases in the cost of living.

Specifically, you will be given a set of products whose quantity and cost are noted on given dates. From that data, we will need to answer questions like:

- Given a shopping cart of groceries, what was the cost of that shopping cart over particular periods of time?
- Given a shopping cart of groceries, what would the cart cost today using product package sizes available today?
- Given starting and ending dates, which products have exhibited shrinkflation?
- Which products are showing anomalous pricing, where larger packages are more expensive per measurement unit?

We assume that all product sizings use metric units, notably grams and kilograms for weight and millilitres and litres for liquid amounts. A good implementation should be ready to handle some other levels of granularity in units.

Problem

Write a Java class, called “CostOfLiving” that holds the configuration of product prices and provides methods to answer questions about the products or shopping carts of products.

You will also submit a small “main” method in a class called “A1” that will demonstrate a minimal use of the Java class.

CostOfLiving

The CostOfLiving class must have the following methods:

Constructor CostOfLiving( )
Initializes an instance of the object

int loadProductHistory(BufferedReader productStream)
The producdtStream contains the description of product price information gathered over time. Each line of the file represents one size of a product. That line consists of four tab-separated values:

- The date when the product cost was checked, in YYYY/MM/DD format
- The name of the product
- The size of the product
- The cost of the product

The size of the product will be a number and a unit pair, with a space between the two, such as “454 g” or “1.5 l”. The units will be limited as given earlier and in lower case, so “g” for grams, “kg” for kilograms, “ml” for millilitres, and “l” for litres.

If the cost of a product is 0 then that product size has been discontinued.

For simplicity, you may assume that there will be no blank lines in the stream until the end of the stream. You may also assume that a date that follows the given format is an actual date of the year, so you need not detect someone reporting a price check on February 30, 2024.

Return the number of product price information entries that are available in the product history. Return a negative value in the case of an error.

int loadShoppingCart(BufferedReader cartStream)
The cartStream parameter contains a set of products and desired quantities. Each product is on its own line and contains a tab-separated product name and size. The product size is as described for products in the loadProductHistory method.

Have this cart information available for future reference. Return a positive integer that represents the cart and that, if given as a cart number in the future to another method, will recall this cart content.

Return a negative value in the case of an error.

float shoppingCartCost(int cartNumber, int year, int month)
The cartNumber parameter refers to a set of products and desired quantities already loaded as a cart.

Given a shopping cart of products, the method will return the cost of buying at least that quantity of product on the first day of the given month and year.

The cost of a product should be the most cost-effective instance with existing packaging to meet or exceed the required amount. If using more than one package to meet the shopping cart size of a product then all those packages should be of the same size.[1]

For example, suppose that the shopping cart asks for 1kg of spaghetti. If the product list contains a 1kg box of spaghetti at the lowest per-gram cost then return the cost of that 1kg box. If there is also a 500g box of spaghetti where buying two 500g boxes is less than the 1kg box then return the cost of buying two 500g boxes. On the other hand, if the product list only contains 900g and 450g boxes of spaghetti and we've asked for 1kg of spaghetti then you are either returning two 900g boxes or three 450g boxes in the cost. The restriction on the solution says that I would not consider an answer of having one 900g box plus one 450g box to meet the 1kg shopping cart target.

If there is a product in the shopping cart that cannot be purchased at the given quote time then return a negative value as the method return value.

Map<String, Float> inflation(int startYear, int startMonth, int endYear, int endMonth)

Report the set of products that are undergoing inflation, whether from cost inflation or from shrinkflation, from the first day of the startMonth in the startYear to the first day of the endMonth in the endYear. The method returns a Map where the Map key is the product name, a space, and the product size, and the Map value is the amount of inflation seen in the product. For the size designation, use the notation of the product's last price check in the given range; for example, if the product size at the start was "1.5 l" and the last price check in the range reported the size as "1500 ml" then report the size as "1500 ml".

If the product size remains constant in the given time period then the inflation is the difference in cost from the start to the end of the period divided by the starting cost.

If the product size has changed in the given period, identified by one size being discontinued and a smaller size introduced within the same month, then the inflation is the difference in the per-unit cost from the start and the per-unit cost from the smaller instance at the end all divided by the starting per-unit cost.

Return a null object in the case of any error condition.

List<String > priceInversion(int year, int month, int tolerance)

We normally expect that bigger sizes of a product cost less, per unit, than smaller ones. A price inversion happens when the smaller sizes are more cost-effective than a larger size.

---

[1] This restriction on meeting the shopping cart demand exists to simplify the algorithm you use. You should be able to identify how this restriction makes your coding easier.

The priceInversion method reports products in a price inversion at a given point in time.

At the first of the month of the supplied month and year, look for all product pairs X and Y where the per-unit cost of the larger product (X) is larger than the per-unit cost of the smaller product (Y). If the percentage in per-unit cost difference is more than the given tolerance then report the pair X and Y as a single string, formatted as

<product name> <tab character> <larger size X> <tab character> <smaller size Y>

The percentage difference is relative to the per-unit cost of the larger product, so per-unit difference divided by the per-unit cost of the larger product

The tolerance exists to avoid reporting minor discrepancies that could arise from just rounding errors. The tolerance parameter is given as an integer out of 100, so a 5% difference will be a tolerance value of 5.

Return null as the list value if an error occurs in the method.

A1

The A1 class has the minimal code to show that you can load a product list and get the current shopping cart value of a shopping cart. The program will ask the user for two file names from the keyboard. The first file will contain the product list. The second file contains the shopping cart information. Given these two inputs, your program will print the cost of the shopping cart to the screen.

Assumptions

You may assume that

- All dates correspond to actual dates. The years are in the range of 1900 to 2024.

Constraints

- Product names and size units should be handled in a case insensitive manner.
- If in doubt for testing, I will be running your program on timberlea.cs.dal.ca. Correct operation of your program shouldn't rely on any packages that aren't available on that system.

Notes

- Make a plan for your solution before starting to code. Write that plan and the parts as part of your external documentation.
- You can create more than one class for your solution. Group information as it makes the most sense.
- Pick a few aspects of the solution to implement at a time rather than try to write code that solves everything in one pass of coding. Use the marking scheme to guide what parts could look like and/or where to prioritize your efforts.

- This assignment does not use exceptions to simplify the coding for anyone who is just learning Java. You may _not_ allow your public methods of CostOfLiving throw exceptions.
- There are no marks for efficiency in this assignment. The focus is on getting something working, not on being the most clever person in the class. If you find this assignment easy then feel free to add the challenge of efficiency to your design and implementation.
- Do not handle exceptions by printing a stack trace or printing to the screen. Neither of these approaches are appropriate in large-scale software so get used to not relying on them now. Also, do not handle options by just throwing them to whoever called you; that's not in the API that you are provided for the assignment.
- The application program interface (API) for this assignment is a poor one. It is designed to provide a simple structure for someone who might be new to Java.

Marking scheme

- Documentation (internal and external) – 3 marks
- Program organization, clarity, modularity, style – 4 marks
- Load a product history and print a shopping cart– 5 marks
- shoppingCartCost – 8 marks
- inflation – 5 marks
- priceInversion – 3 marks
- main method harness – 2 marks

The majority of the functional testing will be done with an automated script or JUnit test cases

Test cases for CostOfLiving class

You will note repetition of some tests between the two load methods. Both are loading information that involves products, so you would expect that they have some shared functionality and so shared test cases.

loadProductHistory

Input validation

- productStream is null

Boundary cases

- productStream is empty
- productStream is 1 item
- productStream has no ending blank lines
- productStream has an ending blank line
- input line has one too few fields
- input line has one too many fields
- input line has a negative quantity
- input line has a zero quantity
- input line has an empty product name
- input line has all spaces for the product name
- input line as an empty size
- input line has a size that is all spaces
- input line has no space separating an input quantity and the units in the size
- input line has a negative cost
- input line has a zero cost
- input line has a cost of 1 cent

Control flow

- input line has the correct number of fields, formatted correctly
- product appears in the input more than once using the same capitalizations
- product appears in the input more than once using different capitalizations
- product is repeated with the same size (different cost and date)
- product is repeated with different sizes
- product is repeated with the same size but a different representation of the size, like “1 l” and “1000 ml”
- product quantity is an integer
- product quantity is a float
- size units are valid
- size units are not valid
- cost is an integer
- cost is a float

- product size has units that are ok but using capitalizations
- data is provided in chronological order
- data is provided grouped by product
- data is provided in a random order

Data flow

- call the method twice, with different data each time

loadShoppingCart

Input validation

- cartStream is null

Boundary cases

- cartStream is empty
- cartStream is 1 item
- cartStream has no ending blank lines
- cartStream has an ending blank line
- input line has one too few fields
- input line has one too many fields
- input line has a negative quantity
- input line has a zero quantity
- input line has an empty product name
- input line has all spaces for the product name
- input line as an empty size
- input line has a size that is all spaces
- input line has no space separating an input quantity and the units in the size

Control flow

- cart contains a unique set of items
- cart stream contains the same item twice
- cart stream contains the same product twice but different sizes
- product quantity is an integer
- product quantity is a float
- size units are valid
- size units are not valid
- product size has units that are ok but using capitalizations

Data flow

- call before loading product history
- call after loading product history
- load the same cart information stream twice
- load two different cart streams

shoppingCardCost

Input validation

- provide an invalid shopping cart identifier

Boundary cases

- month parameter is 0
- month parameter is 1
- month parameter is 12
- month parameter is 13
- negative year
- empty cart
- cart has 1 item in it

Control flow

- cart has more than 1 item in it
- cart item exists and has 1 price quote before the cart cost date
- cart item exists and has a price quote but only after the query date
- cart item most recent quote is for a discontinued product and there is no alternative
- cart item most recent quote discontinues the item but a past item size exists that is smaller
- cart product name (any size) is not in the product history
- cart product name exists, but only at a larger size than in the cart
- cart item doesn't have the size in the product list, and the most cost-effective buy uses the next smaller item
- cart item doesn't have the size in the product list, and the most cost-effective buy uses the smaller items that are not the next smallest
- cart item doesn't have the size in the product list, and the most cost-effective buy would use two different sizes of the product (assignment says that you shouldn't allow this solution to happen)

Data flow

- call before loading any product history
- call before loading any shopping carts
- call after both product history and shopping carts are loaded

inflation

Input validation

- none

Boundary cases

- no product history exists

- only one item in the product history
- start or end month is 0, 1, 12, or 13
- start year is after the end year
- start and end are the same years, but the start month is bigger than the end month
- the start and end dates are the same
- an item in the history has a first price check in or before the time window and a second after the end date
- an item in the history has two prices in the time window

Control flow

- more than one product in the product history
- an item in the history only has one price check at all
- an item in the history has more than two prices in the time window
- no item is inflated in the given time window
- 1 item is inflated by price in the time window
- 1 item is inflated by shrinkflation via a product discontinued and a new product size introduced
- 1 item is discontinued and not added soon enough to be shrinkflation in the time window
- 1 item in the time window has multiple prices using different equivalences for product sizes (same size, different units)
- An item gets cheaper within the time window
- An item is discontinued and a bigger item is added within the time window
- The name of an item appears in different cases in the history within the time window
- More than one item is detected with inflation in the time window
- More than one size of the same product undergoes inflation in the time window

Data flow

- Call inflation twice in a row
- Inflation called before any shopping cart is loaded

priceInversion

Input validation

- none

Boundary cases

- year is negative
- month is 0, 1, 12, or 13
- tolerance is negative
- tolerance is 0
- tolerance is 100
- tolerance is 101

- product history is empty
- just one product in the product history
- use a date before all history began
- use a date after the last date in the history

Control flow

- single size of a product in the history at the given time
- two sizes of the product at the given time
- more than two sizes of the product at the given time
- larger product is per-unit cheaper than the smaller product and the difference is more than the tolerance
- larger product is per-unit cheaper than the smaller product and the difference is equal to the tolerance
- larger product is per-unit cheaper than the smaller product and the difference is less than the tolerance
- larger and smaller products are the same per-unit cost
- larger product is per-unit more expensive than the smaller product and the difference is more than the tolerance
- larger product is per-unit more expensive than the smaller product and the difference is equal to the tolerance
- larger product is per-unit more expensive than the smaller product and the difference is less than the tolerance
- multiple products in the product history at the given time

Data flow

- call before loading a shopping cart
- call after loading a shopping cart
- call twice in a row