School of Computing Sciences

Module: CMP-4008Y Programming 1
Assignment: Coursework Assignment 2

Set by: Jason Lines (j.lines@uea.ac.uk)
Date set: Friday 5th March 2021
Value: 55%

Date due: Friday 14th May 2021 3pm **(Week 11)**
Returned by: Monday 14th June 2021
Submission: Blackboard

Learning outcomes

The aim of this assignment is to facilitate the development of **Java** and **object orientated programming** skills by designing and implementing a program to simulate the operation of a fictional catering company. In addition to using the fundamental concepts that were developed throughout the first assignment (such as **classes**, **objects**, **instance variables** and **instance methods**), this assignment will also develop skills in **UML class diagrams**, **file I/O**, **inheritance**, **exceptions** and basic **enumerative types** in Java. Further general learning outcomes include: describing abstract systems using technical diagrams; following specifications when developing software; documenting code to enhance readability and reuse; increased experience of programming in Java; increased awareness of the importance of algorithm complexity; and using inheritance to model relationships between classes.

Specification

KitchenCorp, a new catering company and sub-division of *GreedyJay Inc.*, have approached you to create a prototype system for simulating the use of their newly proposed snack shops. Due to necessary changes in the catering industry caused by COVID-19, *KitchenCorp* would like to open a new site on the UEA campus that will offer customers the opportunity to purchase snacks and drinks without needing to exchange physical currency.

Figure 1: An artist's impression behind the scenes at *KitchenCorp*'s proposed *Lakeside Cafe* location (probably).

Your task is to model *KitchenCorp*'s new system using object orientated programming in the Java programming language. You will need to model key objects in the system by creating classes to represent objects such as customers, products and shops, in addition to a number

of other classes to support the functionality of the system. A full description of the required implementation is given in the section **Description**, but in summary, you will:

- create a UML class diagram to give an overview of the classes in the system, and how they interact;
- implement the required classes in Java as detailed in the following specification, **including sufficient documentation and testing**;
- simulate the use of the system by loading and processing the provided text files.

Overview

To create a system that models customer interactions and transactions for a snack shop you will firstly need to be able to store information about **products**. This will be an abstract base class that stores basic information about generic products, such as the product ID, name of the product, and the base price of the product (in pence). Subsequently, you will need to *specialise* this class to represent **food** and **drinks** separately. In the proposed shops, *KitchenCorp* will provide a range of cold and hot food items. In the case of hot items, on-site microwaves will be provided for customers to use (and a heating surcharge will be added to the base price of each hot snack purchased, currently set to 10%, whether or not a customer chooses to actually use the equipment). Further, drinks that are sold by the snack shops need to record how much sugar they contain. In the UK, sales of high-sugar drinks are subject to the UK Government's Soft Drinks Industry Levy (aka the *sugar tax*). If the level of sugar is *high*, a 24p tax is added; if the level of sugar is *low*, then 18p will be added as tax to the base price; and if the product is sugar-free (a.k.a no sugar), no surcharge will need to be added.[1]

All **customers** in the system will have a unique customer ID, a name, and a balance. Functionality will also be required to both top-up and charge accounts when appropriate transactions are carried out. Anyone from the public can sign up for a standard customer account, but since *KitchenCorp* will be operating on UEA campus, they have agreed to special treatment of **students** and members of **staff**. Students, regardless of school of study, will get a flat discount of 5% off all items (after taxes and surcharges have been applied) and will also be able to have a negative balance of up to £5, effectively giving all students £5 credit to start with. Staff will not be able to have a negative balance, but staff from certain schools will still be able to benefit from discounts. CMP staff will receive an increased discount rate of 10% (as a thank you for outsourcing the development of the system to CMP students) while staff in BIO and MTH will receive a 2.5% discount on snacks and drinks (because Jason plays football with them). Staff from other schools will get no discount (tough luck, humanities!). Staff discounts will also be applied after any specific surcharges or taxes that apply to products.

Once products and customers can be modelled, each **snack shop** should maintain its own collection of products and its own collection of customers. It will need functionality to process transactions when passed a customer ID and a product ID, charging a customer the correct amount for a product if the customer has a sufficient balance (or rejecting the sale if they do not). A shop will also need to keep track of how much the total value of all sales is (this is the most important thing to *GreedyJay Inc.*, of course) and it will also require some specialised methods that are of interest to corporate, such as the most expensive item in a shop's catalogue and the average and median balance of customer accounts.

[1] Please note that this is not *exactly* how the tax is applied in real life, but this is how it should be applied in this prototype system/assignment.

Finally, this will all be tied together by populating a snack shop with customers and products by reading in a number of text files that have been provided. You will also be given a list of transactions in another file to **simulate** the use of a snack shop

Please note that you are expected to provide `toString` methods for all object classes, *appropriate* comments to aid someone reading your code, and evidence of testing in all classes (i.e. simple test harnesses in all of your non-abstract object classes).

(hint: please read the full assignment rather than diving in straight away - it will make your life a lot easier if you implement it in a logical order and plan ahead).

Description

1. UML Class Diagram (15%)

Your first task is to fully read this assignment specification and then create a UML class diagram for the proposed system. You should include all classes, and relationships between them, but you are not required to include accessor and mutator methods in this diagram, and you also should not include your main method class (`Simulation`). Marks will be awarded for the accuracy and correctness of your diagram, and presentation will also be taken into account (i.e. make sure that it is clear and easily readable, and make sure it follows conventions taught on this module for UML class diagrams and not conventions taken from anywhere else).

I recommend that you use diagrams.net as shown in the live lecture that accompanied UML class diagrams, but you are free to use any other simple tools (such as MS PowerPoint or Word) to *draw* your diagram if you wish.

To avoid issues when including your diagram in PASS, please make sure to save your UML class diagram as a .pdf and do not include spaces in your file name (you can use *export→pdf* in diagrams.net/PowerPoint/Word to do this, or ask for help from the lab assistants if you are struggling to format your work correctly - they cannot answer the coursework for you but they are free to help you with technical issues).

(hint: before starting, read the full coursework specification first and then come back to create your class diagram before writing code. It will help you understand all of the functionality and relationships between the classes, and give you something to refer to while working on the code - that is the whole point of class diagrams, after all!).

2. Object Classes

This section describes the classes that you must implement. Please note that *all* non-abstract object classes should have a main method to demonstrate simple usage/testing and an *appropriate* implementation of `toString`. You do not need to include a full javadoc, but you should include a short comment at the start of each class to explain its purpose and use *appropriate* comments throughout to explain any complex operations or calculations. If it is not immediately obvious what a piece of code is doing then this is a good candidate for a comment that would aid a reader.

2.1 `Product` (10%)

Products are split into two different categories: food and drinks. You will model this using an abstract base class for products and subclasses for food and drink.

All products have a unique product ID, a name, and a base price (stored as whole numbers in pence - e.g. £2.00 should be stored as 200). A valid product ID consists of a single letter followed by a single dash ('-') and then 7 numeric characters (for example D-1234567)

You are required to write an abstract class called `Product`. This class should include fields for the product ID, name and base price of a product. Your class should have a single constructor that takes values for each of those three fields and it should throw an `InvalidProductException` if the provided ID is not valid or the base price is negative (you will need to implement a class for this Exception). Your `Product` class should have accessor methods for all fields and an abstract method called `calculatePrice` that returns an int and takes no arguments as inputs.

2.2 `Food` and `Drink` (15%)

You are required to extend `Product` into two sub-classes:

- `Food` should have an extra field to determine whether it is a hot food item (true if so, false otherwise). `Food` should also have a public final class variable to store the current surcharge percentage that is to be applied to sales of hot food items (currently set at 10%). `Food` should have a single constructor that takes in values for all fields. Additionally, the first character of the product ID must be 'F' for food items and the constructor should throw an `InvalidProductException` if this is not the case.
- `Drink` should include an additional field to determine whether the item in question has sugar content that is considered high, low, or none. The class should have two constructors: one that takes in arguments for all fields, and one that takes in arguments for all fields except the information about sugar content (this constructor should set the sugar content to none as a default value). Similarly to food, the first character of a drink's product ID must be 'D' and the constructor should throw an `InvalidProductException` if this is not satisfied.

You classes should have accessor methods for the new fields and and each should have an appropriate implementation of `calculatePrice`. This method should consider the base price of a product and then apply any additional charge or tax that is applicable (remember, in the Outline section, it stated that hot foods should have a surcharge added to their base price and sugary drinks should have a tax of either 24p or 18p added to their base price if they have high or low sugar content respectively). If the calculated price is not a whole number then round *up* to the nearest penny.

2.3. `Customer` (10%)

The `Customer` class should store information about a customer's account. This includes basic information such as their account ID (an 6 digit alphanumeric identifier), their name and their account balance (as an integer in pence, e.g. 1000 for £10). You should have two constructors, one for ID and name that sets a default balance of 0, and another constructor that takes arguments for all fields. The account balance is the amount of money that is in a customers account, which should be a positive number that is reduced by the appropriate amount when a product is purchased. Your constructors should throw an InvalidCustomerException if the account ID is

incorrect or a negative balance is provided (you will need to provide this Exception class).

Your class should include accessors for all fields, and two specific methods for manipulating the balance of an account. First, there should be an `addFunds(int amount)` method to allow a customer to top-up their account balance (only add a positive amount to a balance and do not alter the balance if a negative amount is provided). Second, there should be an instance method for `chargeAccount(int productPrice)`. `chargeAccount` will simulate the process of a customer's account balance being reduced when they buy a product. If the customer's balance is sufficient, it should simply be reduced by the amount equal to `productPrice` and the price that was deducted should be returned (note that product price should normally be the price of a product including relevant taxes and surcharges) . However, if the customer does not have enough money in their account, an `InsufficientBalanceException` exception should be thrown (you will need to create this class as a sub-class of `Exception`) and no update to the account balance should be made.

2.5. `StudentCustomer` (7.5%)

The `StudentCustomer` class should extend the `Customer` class. It should have two constructors (as `Customer` did). And remember, all students are given a 5% discount and are allowed to have a negative balance of up to £5 (i.e. -500) (see **Overview** for more details) so `chargeAccount` should be overridden appropriately (and make sure that it returns the price after the student discount is applied so you can record how much was actually charged after the discount). As before, if rounding is necessary for a calculated price then round *up* to the nearest penny. The toString for this class should clearly indicate that this is a student customer.

2.6. `StaffCustomer` (7.5%)

The `StaffCustomer` class should also extend the `Customer` class. It should include an additional field to store the school that the staff member works in (which could either be CMP, BIO, MTH, or other). There should be two constructors: one that takes values for all fields, and one that requires all fields except balance (and assumes a default value of 0). This class will also have a suitable implementation of `chargeAccount` to take any discounts into consideration (recall from **Overview** that this should be 10% for CMP, 2.5% for MTH and BIO, and no discount for other schools). Once again, make sure that this returns the price that was actually charged to the customer (after discount) and round up to the nearest penny if needed. The toString for this class should clearly indicate that this is a staff customer.

2.7. `SnackShop` (20%)

You will also need to create a class to model a snack shop. This class should have fields for the shop's name, a field for the turnover of the shop, a collection of the products that this shop sells, and and a collection of the customers that have accounts at this shop. It should have a **single constructor that takes a single argument for the shop's name** and methods to add individual customers and products. Further, it should have accessor methods for the shop's name and the turnover of the shop.

You should also provide methods for:

- `getCustomer(String customerID) throws InvalidCustomerException`
- `getProduct(String productID) throws InvalidProductException`

Finally, you should also have `processPurchase(String customerID, String productID)` method which will be used to process a transaction when given a customer ID and a product ID. This method should tie together what you have already implemented - it should retrieve the correct product, the correct customer, and try to reduce that customer's balance by the appropriate amount. If successful, this amount should be added to the shop's `turnover` amount and you should return true to indicate that the transaction was a success. Otherwise, the method should throw an appropriate exception if the transaction could not be processed.

Additionally, you should have the following methods:

- `findLargestBasePrice()` which should return the largest base price of any product sold at a shop;
- `countNegativeAccounts()` which should return the number of customer accounts associated with this shop that have a balance of less than 0; and
- `calculateMedianCustomerBalance()` which should return the median balance across all customers with accounts at this store.

It is up to you to decide how you wish to store collections of products and customers. The simplest solution is to use arrays/`ArrayList`, but you can use any data structure that extends the Java `Collection` class (see the Java API [2] for more information on options). A small number of additional marks will be awarded for using a more appropriate data structures than array-based collections, but only if you also justify your choice in a small comment when you declare the fields in this class (i.e. I want to state why you think your solution is better than an array/ArrayList).

3. Main method class: `Simulation` (15%)

Your main method class will simulate the creation and use of a snack shop. You are provided with three files:

- `customers.txt`. This file contains information about customer accounts that should be created for your simulation. Each line includes information for an individual customer.
- `products.txt`. This file contains information about the products that should be created for your simulation. Again, each line includes information for an individual product.
- `transactions.txt`. This file includes a chronological list of transaction that you should simulate using the appropriate methods that you have implemented.

Simulation will be the main method class of your project and, in addition to a main method (more on that later), you should implement two static methods:

- `initaliseShop(String shopName, File productFile, File customerFile)` This method should return a SnackShop with its name set to `shopName`. Two files should also be passed into the method: one with information about products, and one with information about customers. This method should parse both of these files and add each customer and product from the respective files to the SnackShop (using the methods you implemented in the previous question) before it is returned.

[2] `https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html`

- `simulateShopping(SnackShop shop, File transactionFile)`
  This method should parse and proceed through a file that contains a list of transactions, simulating the running of a shop (such as customers buying products, adding funds, or signing up for a new account). Each line should be processed individually and the action should be carried out on the SnackShop that is also passed into the method (and catch any exceptions that occur). As you process each line of the transaction file **you should print out an informative summary line for each successful action**, and **also a summary line for each action that could not be performed** (such as a customer having an insufficient balance, or rejecting a purchase when an unrecognised product ID or customer ID is found, etc.). The actions that you can expect to see are adding new customers to the shop (`NEW_CUSTOMER`), customers attempting to purchase an item (`PURCHASE`), and customers adding funds to their account (`ADD_FUNDS`). Once all transactions have been processed, you should call your methods from `SnackShop` to find the largest base price, the number of negative balances, and the median customer balance, and you should print out a line to inform the user of the results of each of these. Finally, you should then print out a message to state the shop's turnover by calling your accessor.

Your main method should call initialiseShop to create a SnackShop, and it should then pass that SnackShop into your simulateShopping method. It is up to you what you want to call the shop, but make sure that you use the three files that have been provided (make sure that you place those files in the root directory of your project). Note: it is important that, when run, your code does not need any further information from the user (i.e. your main method should simply call `initialiseShop` with the provided files, and then pass the returned `SnackShop` into the `simulateShopping` method without requiring any keyboard input to run when PASS calls your main method).

Relationship to formative work

Please see the following lectures and labs for background on each of the specified tasks:

- **Gavin's content from Semester 1**. In particular, the fundamentals from the first assignment are relevant here. Lecture 5 introduces how to use objects, lecture 11 introduces writing classes, lecture 14 includes using collections to store objects, and lecture 16 introduces inheritance.
- **Classes and objects**: the use case example that I went through in week 2 of semester 2 demonstrates the structure of a typical class. The lab reinforced this.
- **UML class diagrams**: these were introduced in semester 2 week 2, and further explained in week 3 with relevance and examples including inheritance.
- **File I/O**: Gavin introduced this in semester 1, and I have shown an alternative method of doing this in the live lecture in week 2 of semester 2
- **Inheritance**: this was covered in more detail in week 3 of semester 2.
- **Exceptions**: this was covered in week 4 of semester 2 and examples were given in the recorded and live lectures, as well as the lab exercises.

Deliverables

Your solution must be **formatted using PASS** to produce a .pdf containing your UML class diagram (included as a .pdf itself), the source code of your project, the compiler messages (if any) and the output of your program. Once formatted, you **must** submit your .pdf on Blackboard.

IMPORTANT: PASS is only used to format your submission and you must submit the output of running your work through PASS to Blackboard yourself. **If you do not submit anything on Blackboard then you have not submitted anything**.

PASS is available on all lab machines and remotely via the server-based version of PASS that was provided for the first coursework assignment. Instructions are given on Blackboard for using the server-based version of PASS, and limited help will be available on the day of submission to help with any issues relating to your use of PASS, but please make sure you take the time to understand how to use PASS in advance of the deadline and practice formatting a solution, if necessary (you should already know how to use PASS if you completed the first coursework assignment).

And remember, PASS is not able to provide input to your program via the keyboard, so programs with a menu system, or which expect user input of some kind, are not compatible with PASS. Design your program to operate correctly without user input from the keyboard as specified in the `Simulation` class.

Resources

- **Previous exercises**: If you get stuck when completing the coursework please revisit the lab exercises that are listed in the *Relationship to formative work* section during your allocated weekly lab sessions. The teaching assistants in the labs will not be able to help you with your coursework directly, but they will be more than happy to help you understand how to answer the (very) related exercises in the lab sheets. You will then be able to apply this knowledge and understanding to the new problems in this coursework assignment.
- **Discussion board**: if you have clarification questions about what is required then you **must** ask these questions on the Blackboard discussion board to make sure that other students have the same information for fairness (there will be a specific discussion board topic with anonymous questions to enable this). Also, please check that your question has not been asked previously before starting a new thread.
- **Course text:** *Java Software Solutions* by *Lewis and Loftus*. Any version of this textbook is helpful for Programming 1 and will have specific chapters on topics such as inheritance and Exceptions. You can buy your own copy, but I'd suggest doing a simple online search as many editions of the text are available online for free. The library also has a few copies of the latest version of this textbook too.

Marking Scheme

Marks will be awarded according to the proportion of specifications successfully implemented, programming style (indentation, good choice of identifiers, commenting etc.), and appropriate use of programming constructs. It is **not sufficient** that the program just generates the correct output. Professional programmers are required to produce maintainable code that is easy to understand, easy to debug when bug reports are received, and easy to extend. Itemised marks

are provided throughout the assignment description, but to summarise the marks available for each part:

1. UML Class Diagram (15 marks)
2. Object Classes
    2.1. Product (10 marks)
    2.2. Product sub-classes (15 marks)
    2.3. Customer (10 marks)
    2.4. Customer sub-classes (15 marks)
    2.5. SnackShop (20 marks)
3. Simulation (15 marks)

Total: 100 Marks