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 3rd March 2023
Value: 55%

Date due: Friday 12th May 2023 3pm
Returned by: Friday 9th June 2023
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 toll road 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

A toll road is a special type of road where users pay for passage, typically using a private road to reduce travel time by taking a shorter route to their destination or avoiding traffic on congested public routes. A private company, *Shady Roads Inc.*, has approached you to design and implement a system for managing a simple automated toll road that they would like to build on the outskirts of Norwich.

Figure 1: An example of a toll road.

The classes that you are required to create for this system are described in Description section. Your tasks are to:

1. Describe this abstract system using a UML Class Diagram.

2. Implement each of the classes in Java (using test harnesses to check each class as you make them).

3. Create a main method class to read data from file to populate your system, and then read in another file to simulate the use of your toll road.

(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)

Overview

To model this system, you will need to create a number of classes. A **toll road** class will store information for a single road and have a list of **customer accounts** that are registered with it. Each of these customers will in turn have a single **vehicle** associated with their account. The types of vehicle that are to be included in the system are: **cars**, **vans**, and **trucks**. Each type of vehicle has a subset of *base* information (e.g. they all have a vehicle registration plates and manufacturers), but each *subclass* of vehicle has specialised behaviour too. For example, the cost of using a car on the toll road will be cheaper than using a truck, but the number of seats in the car will affect how much it costs too. Each customer may have a special discount ("staff" and "friends or family" have different levels of discount), and you will also need to implement **two simple** `Exception` **classes** to help your system elegantly handle exceptional behaviour. Please note that to avoid floating point errors, we will model all currency in pence by using integers (e.g. £1 should be represented as an `int` of 100).

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 (`TollRoadMain`). 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).

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 Vehicle

A `Vehicle` class should be implemented to store two fields for each object of this type: a vehicle registration, which may contain a mixture of letters and numbers (e.g. "EK02DEU"), and the make/manufacturer of the vehicle. The class should have a single constructor that takes two arguments to set the fields and an abstract public method called `calculateBasicTripCost` that takes no arguments and returns an `int`. The class should also have accessor methods for both fields.

2.2 Car

This should be a subclass of `Vehicle` that stores an additional field called `numberOfSeats`, which is used to store the number of people that can be transported in this vehicle. You should override `calculateBasicTripCost` to return 500 (e.g. £5.00) if the car has fewer than 6 seats, or 600 (£6.00) if it has a greater capacity. You should have a single constructor that sets values for all attributes of `Car` and a further accessor for the new attribute of this class.

2.3 Van

This should be a subclass of `Vehicle` that stores an additional field called `payload` to store the amount of cargo (a whole number in kilograms) that this van can carry. A single constructor should be implemented with arguments to set all fields, and `calculateBasicTripCost` should be overridden to return:

- 500 if the payload is less than or equal to 600KG;
- 750 if the payload is less than or equal to 800KG, but greater than 600KG;
- 1000 if the payload is greater than 800KG.

2.4 Truck

This should be a subclass of `Vehicle` that stores an additional field called `numTrailers` to store the number of trailers that this truck is designed to tow. A single constructor should be implemented with arguments to set all fields, and `calculateBasicTripCost` should be overridden to return:

- 1250 if the truck can tow a single trailer;
- 1500 if the truck can tow two or more trailers.

2.5 CustomerAccount

The CustomerAccount class will be used to store information about a customer, including their first name, their last name, their account balance, the `Vehicle` that is associated with their account, and the level of discount that this customer has (if any). There should be a single constructor that takes arguments for four fields (first name, last name, starting account balance and a `Vehicle`), but no argument should be provided for this discount as it should be assumed the customer has no discount when created. Instead, an appropriate method can be called to set discounts where necessary (as described below).
The methods implemented in this class should include:

1. `activateStaffDiscount()` to set the discount type of this account to be staff;
2. `activateFriendsAndFamilyDiscount()` to set the discount type of this account to be friends and family;
3. `deactivateDiscount()` to remove any active discount on the account;
4. `addFunds(int amount)` to add more credit to the account balance;
5. `makeTrip()` to simulate the customer making a trip on the toll road. This method should calculate the trip cost by calling the correct `calculateBasicTripCost` method of the `Vehicle` associated to this account. Once this is retrieved, this cost should *then* be discounted by 50% for staff members, or by 10% for friends and family. If the calculated cost of the trip after any discount has been applied is not a whole number then it should be rounded down to the nearest pence (e.g. 50.9 would become 50). If the account has sufficient funds to make this trip, the balance should be reduced by cost of the trip and the method should then return the cost as an `int`. If the account does not have a sufficient balance, the method should throw an `InsufficientAccountBalanceException`.

A reminder that this class should have a single constructor that sets the first name, second name, vehicle, and balance of an account. An account should have a default value of no discount in this constructor, which can be modified by calling either of the relevant methods: `activateStaffDiscount` or `activeFriendsAndFamilyDiscount`.

`CustomerAccount` should implement the `Comparabale` interface. The `compareTo` method should compare `this` account to another `CustomerAccount`; it should return -1 if the registration number of the vehicle associated to `this` account comes before that of the vehicle in the other account (according to alphabetical order), 0 if they are the same, or 1 if it is greater. Finally, your class should also have accessor methods for all fields.

2.6 TollRoad

The `TollRoad` class should have two attributes: one to store a collection of `CustomerAccount` objects, and an `int moneyMade` to keep track of the total money that has been made by the toll road. In addition to a default constructor and accessor methods for both fields, `TollRoad` should have three methods:

1. `addCustomer(CustomerAccount acc)` should add a new account to the list of customers associated with `this` road.
2. `findCustomer(String regNum)` should search through the list of customers associated with `this` road and return the matching `CustomerAccount`. If no match is found, a `CustomerNotFoundException` should be thrown.

3. `chargeCustomer(String registrationNumber)` should search through all accounts associated with `this` road to find a match to the input registration. If no match exists, the method should throw a `CustomerNotFoundException`. If a matching account is found, the method `makeTrip()` should be called on the matching account. If that is successful, the cost of the trip should be added to the `moneyMade` attribute. If the account does not have sufficient funds then an `InsufficientAccountBalanceException` should be thrown.

It is up to you to decide how to store the `CustomerAccount` objects associated with a `TollRoad`. It is fine to use an ArrayList here, but a small number of marks will be awarded for using a more efficient choice of data structure for storing customers (hint: think about the most common functionality that will happen with the list of customers during the normal use of a TollRoad). If you use something other than an ArrayList, make sure to add a small comment in your class to also explain *why* your choice of data structure would be preferable (note: you do not need to implement your own data structure! Refer to the Week 6 lecture on data structures and feel free to do you own research into the built-in data structures within Java).

CustomerNotFoundException and InsufficientAccountBalanceException

You will need to implement these two classes as subclasses of `Exception`. They do not need any additional functionality above what is already provided by the `Exception` class; no overridden or additional methods are required and a default constructor in each is sufficient. The purpose of these classes is to allow you to throw and catch specific exceptions when different types of exceptional events occur in your application, rather than just using a generic `Exception`.

2.7 TollRoadMain

The final class that you need to implement is a main method class, `TollRoadMain`. This class should have three methods:

1. `initialiseTollRoadFromFile()`
2. `simulateFromFile(TollRoad road)`
3. `main`

2.7.1 initialiseTollRoadFromFile()

This method should create a new `TollRoad`. It should then read through `customerData.txt` to populate this road with new `CustomerAccount` objects. The file is in the format:

```
<vehicleType>,<regNum>,<firstName>,<lastName>,<vehicleInfo>,<startingBalance>,<discountType>#
```

where each customer is delimited by a hash (#), and each piece of information about a customer is delimited by a comma (,). Take specific note of `vehicleType` and `vehicleInformation`:

- `vehicleType` determines whether this customer has a Car, Van, or Truck;
- `vehicleInformation` is specific to that type of vehicle (e.g. number of seats for a car, payload for a van, or number of trailers for a truck).

For example:

```
Car,CB13IIZ,Cyndi,Banister,Kia,7,1500,STAFF#
Truck,FM66JPE,Blondell,Boaz,Ford,1,700,FRIENDS_AND_FAMILY#
Van,YJ55XPM,Jessie,Ruhland,Vauxhall,450,1200,NONE#
```

This method should finish by returning the `TollRoad` object.

2.7.2 simulateFromFile(TollRoad road)

This method should have a single argument to pass in a `TollRoad`. This method should read `transactions.txt` and carry out the actions listed in the file on the toll road. Data in this file can be in two forms:

- `addFunds,<registrationNumber>,<amount>$`
- `makeTrip,<registrationNumber>$`

addFunds

If the instruction is `addFunds`, you should call the appropriate methods to add `amount` to the balance of the account with the registration `registrationNumber`. For example:

```
addFunds,EK02DEU,200$
```

This should lookup the account with the vehicle registration number `EK02DEU` and add `200` to the balance. If this operation is successful, a line should be printed to the console in the form of:

```
EK02DEU: 200 added successfully
```

If the operation could not be completed (e.g. account does not exist), catch the Exception and print a line stating that this operation was unsuccessful:

```
EK02DEU: addFunds failed. CustomerAccount does not exist
```

Your method should then continue to process subsequent instructions until the end of the file is reached.

makeTrip

If the instruction is `makeTrip`, a trip should be attempted with the account that matches the specified registration number. If successful, a line should be printed to the console to indicated this:

```
EK02DEU: Trip completed successfully
```

If the trip is not successful (account does not exist, or it does exist and has insufficient funds), the appropriate Exception should be caught and a line should be printed to the console stating the appropriate message in the following form:

```
EK02DEU: makeTrip failed. CustomerAccount does not exist
```

or

```
EK02DEU: makeTrip failed. Insufficient funds
```

Your method should then continue to process subsequent instructions until the end of the file is reached.

2.7.3 main method

This class should have a `main` method that is executed as the project's `main` method. It should simply create a `TollRoad` using your `initialiseTollRoadFromFile` method, and then call your `simulateFromFile` method with the `TollRoad` as the arguement. Finally, it should print out a message stating how much money the toll road made during the simulation.

Your Tasks

1. **Create a UML Class Diagram** of the system that is described in this document. You should include all classes, attributes, operations, and relationships between classes.

2. **Implement** each of the object classes described in Section 1 (Vehicle, Car, Van, Truck, CustomerAccount, TollRoad, TollRoadMain).

3. **Implement** the `TollRoadMain` main method class as described above. Use the provided input files as input to your methods. Once your code is complete you must run it through pass.cmp.uea.ac.uk to output a formatted .pdf file that includes all of your code and the output that it generates (more info in the Deliverables section below). Once completed, you must submit this file on Blackboard as PASS is not a submission point.

There should be evidence of testing in *all of your object classes* (with the exception of any abstract classes). You should at least use a test harness in each class to construct objects and call methods on them to test that the functionality of your classes is correct. Also, it is good practice to override `toString` generally, and a requirement of this assignment is that all of your classes must override `toString` to return an appropriate `String` value (with the exception of the main method class).

Deliverables

Your solution **must** be submitted via blackboard. The submission **must** be a single `.pdf` file generated using PASS, using the PASS server at `http://pass.cmp.uea.ac.uk/` . The PASS program formats your source code, and compiles and runs your program, appending any compiler messages and the output of your program to the `.pdf` file. If there is a problem with the output of PASS, contact me (j.lines@uea.ac.uk). Do not leave it until the last moment to generate the `.pdf` file, there is a limit to the amount of help I am able to give if there is little time left before the submission deadline.

PASS is the target environment for the assignment. If your program doesn't operate correctly using PASS, it doesn't work, even if gives the correct answers on your own computer:

- The PASS program 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 any user input from the keyboard.

- Do not use absolute path names for files as the PASS server is unable to access files on your machine. If your program is required to load data from a file, or save data to a file, the file is expected to be in the current working directory. If the program is required to load data from a file called `rhubarb.txt` then the appropriate path name would be just `"rhubarb.txt"` rather than `"C:some\long\chain\of\directories\rhubarb.txt"`.

- If you develop your solution on a computer other than the laboratory machines, make sure that you leave adequate time to test it properly with PASS, in case of any unforeseen portability issues.

- If your program works correctly under InteliJ on the laboratory machines, but does not operate correctly using PASS, then it is likely that the data file you are using has become corrupted in someway. PASS downloads a fresh version of the file to test your submission, so this is the most likely explanation

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.
- **Live lecture code:** Remember that I have been writing code in the live lectures of the module each week and have been pushing the code to GitHub. I have intentionally covered many of the topics that will appear in the coursework, so this is a good place to revise some of the topics that you will need to tackle in this assignment. A reminder that you can find the codebase here: https://github.com/jasonlines/CMP-4008Y-2023

Tips

- Identify appropriate places where you could use enumerative types - these have not been specifically requested, but there are some obvious places where they would be more appropriate than using in-built Java types and it is expected that you can recognise this and use `enums` appropriately.
- File reading should **only** happen in the main method class (`TollRoadMain`). You should not have *any* file reading logic within the object classes themselves.
- It is fine to implement additional helper methods that are not specified in the coursework description if you choose, but this is not required and you will not gain extra marks. Do not add additional constructors or functionality that would change the intended implementation of the classes that have been described, however.
- Make sure to follow the specification and do not make unnecessary changes to it. If specific class names and method names are requested then you must follow these requirements.
- You do not need to load the data from files to test any of your object classes - you should start by using hard-coded values in your test harnesses to check that you implementations are correct before even considering the data within the text files.
- Generally, exceptions are thrown within object classes and caught in the main method of your project - if you catch them too early (e.g. as soon as they are thrown in an object class rather than in the main method that called it) then there is no chance to handle the exceptional behaviour appropriately. Similarly, aside from when developing/debugging

your code, printing to the console should normally only happen in test harnesses and the main method class (with the exception of any methods that specifically require this - i.e. if a requested method specifically states that a message should be printed). For example, if you print a message every time you construct an object, this may lead to annoying output if you were to reuse your class in another project (which is common in real-world development) because there would be no way to turn off the output.

Plagiarism, collusion, and contract cheating

The University takes academic integrity very seriously. You must not commit plagiarism, collusion, or contract cheating in your submitted work. Our Policy on Plagiarism, Collusion, and Contract Cheating explains:

- what is meant by the terms 'plagiarism', 'collusion', and 'contract cheating'
- how to avoid plagiarism, collusion, and contract cheating
- using a proof reader
- what will happen if we suspect that you have breached the policy.

It is essential that you read this policy and you undertake (or refresh your memory of) our school's training on this. You can find the policy and related guidance here:

https://my.uea.ac.uk/departments/learning-and-teaching/students/academic-cycle/regulations-and-discipline/plagiarism-awareness

The policy allows us to make some rules specific to this assessment. Note that in this assessment working with others is *not* permitted. All aspects of your submission, including but not limited to: research, design, development and writing, must be your own work according to your own understanding of topics. Please pay careful attention to the definitions of contract cheating, plagiarism and collusion in the policy and ask your module organiser if you are unsure about anything.

Marking Scheme

Marks will be awarded according to the proportion of specifications successfully implemented, programming style (indentation, good choice of identifiers, commenting, testing, etc.), and appropriate use of object oriented programming constructs. Note that it is **not sufficient** to ignore the specification to simply produce the "correct" output - marks are not given for the output specifically and having the correct output only implies that the specification *may* have been implemented correctly. 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. **Vehicle class (10 marks)**
3. **Car class (10 marks)**
4. **Van class (10 marks)**

5. **Truck class (10 marks)**
6. **CustomerAccount class and InsufficientAccountBalanceException class (15 marks)**
7. **TollRoad class and CustomerNotFoundException class (15 marks)**
8. **TollRoadMain class (15 marks)**

Please note that these marks will be influenced by your ability to follow Java conventions, correctly formatting your code, and providing evidence of testing within your classes.