Advanced Programming in Java

Assessed Coursework

The deliverable for this coursework must be submitted via NESS on or before:

- **16:30 on Friday 26 October 2018**

The coursework, described below, involves development of the classes and tests to be used in development of a Car Rental application. The coursework contributes 33% of the overall module mark.

---

1. Aim

The aim of this coursework is to practice the design principles covered in lectures. You will develop interfaces and classes to demonstrate that you have learned and understood module material, including:

- appropriate overriding of `Object` class methods, including overriding `toString` and providing a static `valueOf` method when appropriate
- design of interface-based hierarchies, programming through interfaces and late binding
- the use of factories to control instantiation of objects, including guaranteeing the instantiation of unique instances
- defensive programming including the use of immutability
- the use of appropriate interfaces and classes from the Collections framework
- appropriate use of Javadocs to document your interfaces and classes
- the use of testing

The coursework is **not** algorithmically challenging. The focus is on good design and good practice.

The coursework is **not** about development of an end-user application. You are developing interfaces and classes that could be used for the development of an application. You should **not** develop a graphical user interface or a command line interface. They are not necessary and you will be given no credit for doing so. You can test all your interfaces and classes using unit tests from within the Eclipse environment.

Note: the car rental system specified below is a deliberate simplification. It is not an accurate model of real world car rental. Your solution should correspond to the simplicity of the specification. You risk losing marks if you attempt to provide a more realistic model of car rental or provide a solution that is more complicated than necessary.

---

2. System overview

A car rental company needs a set of interfaces and classes to manage car rentals.

The company rents cars from its fleet of 20 small cars and 10 large cars. For this coursework, the significant difference between small cars and large cars is that they consume fuel at different rates (see Section 3).

When all cars of a particular type have been rented out, no more cars of that type can be issued by the company until one of the rented cars is returned.

A car can only be rented out to one person at a time. So, a car that is out for rent cannot be rented out again until after it has been returned and the existing rental contract terminated. Once a car has been returned, it is available for rent again. Another way of expressing this guarantee is that a car that is not rented or has been returned cannot be driven (it is "deactivated"). That is, the status of any given car is either rented to one person (in which case it can be driven) or not rented (in which case it cannot).

A person can only rent out one car at a time.

The rental company needs to maintain a record of who has rented a given car (associating a person with the car they have rented). In addition, they need to be able to issue cars for rent and terminate rental contracts on return of cars. They also require information on cars currently out to rent. The following provides more detail on the required functionality:

availableCars(typeOfCar)
: This method returns the number of cars of the specified type that are available to rent.

getRentedCars()
: This method returns a collection of all the cars currently rented out (if any).

getCar(person)
: Given a person, this method returns the car they are currently renting (if any).

issueCar(person, drivingLicence, typeOfCar)
: Given a Person (the rentor), the person's DrivingLicence and a specification of the type of car required (small or large), this method determines whether the person is eligible to rent a car of the specified type and, if there is a car available, issues a car of the specified type. The car has a full tank of petrol at the start of the rental. The method associates the car with the person renting it (so that the company has a record of cars out for rent and the people renting them). If a car cannot be issued, the method returns an appropriate indication of the failure to issue a car. Note, this does not have to indicate why a car cannot be issued; it simply indicates that a car cannot be issued.

  The rules for determining whether a car can be issued or not are given below:

  - the person renting the car must have a full driving licence
  - they cannot rent more than one car at a time
  - to rent a small car, they must be at least 20 years old and must have held their licence for at least 1 year
  - to rent a large car, they must be at least 25 years old and must have held their licence for at least 5 years

terminateRental(person)

This method terminates the given person's rental contract. In effect, the person is returning the car. The car is then available for rent by someone else. The method removes the record of the rental from the company's records (disassociating the car from the person) and returns the amount of fuel in Litres required to fill the car's tank. The person returning the car must either have returned the car with a full tank or will be liable for the number of Litres required to fill the tank. This **terminateRental** method is not responsible for managing charges for the required fuel. It just reports the amount of fuel required to fill the tank. If a person attempts to terminate a non-existent contract, this method does nothing.

Note: you can assume that people renting cars have pre-paid vouchers for all rental costs except fuel. You are **not** concerned with the management of the vouchers.

---

3. Implementation

To complete the car rental system outlined in Section 2 you will need to provide interfaces and classes for the functionality described in this section. You must also implement test classes to unit test your solution.

Cars

All cars have the following functionality:

- a method to get the car's registration number
- a method to get the capacity in whole Litres of the car's fuel tank
- a method to get the amount of fuel in whole Litres currently in the fuel tank
- a method that indicates whether the car's fuel tank is full or not
- a method to add a given number of whole Litres to the fuel tank (up to the tank's capacity) and which, after execution, indicates how much fuel was added to the tank
- a method to "drive" the car for a given number of whole Kilometres that returns the number of whole Litres of fuel consumed during the journey

The following rules determine the meaning and functionality of the drive method (and of the method to get the capacity of a car's tank).

- A car cannot be driven if it is not currently rented.
- A car cannot be driven if it has 0 or less Litres of fuel in its tank.
- A small car consumes fuel at the rate of 20 Kilometres/Litre.
- A large car consumes fuel at the rate of 10 Kilometres/Litre for the first 50 Kilometres of a journey and then at the rate of 15 Kilometres/Litre for the remainder of the journey.
- The drive method calculates the amount of fuel consumed during a journey, deducts that amount from the fuel in the tank, and returns the amount of fuel consumed. If the car cannot be driven the method should provide an appropriate indication that no journey has been undertaken.

- The tank capacity of a small car is 49 Litres.
- The tank capacity of a large car is 60 Litres.

To reduce the complexity of calculations considerably, you can apply the following simplifications of the real world.

- All journeys are either for 0 Kilometres (the car cannot be driven) or for the full distance specified as a parameter to the drive method.
- A single journey is allowed to consume more fuel than is available in the tank. So, at the end of a journey the fuel in the tank may be a negative amount of Litres. As noted above, a car cannot be driven when its tank is empty (or less than empty!). This means a journey cannot be started when the tank has 0 or less Litres available. More fuel will have to be added before starting the journey.
- All calculations are for whole Litres and whole Kilometres. Use integer arithmetic in calculations. Any journeys that would in the real world require the consumption of a partial Litre, require the consumption of a whole additional Litre in this model of driving. So, for example, a small car will consume 1 whole Litre for a journey of 1, 10, 15, 19 or 20 Kilometres.

You must provide an appropriate hierarchy for cars. Your car rental class issues a car of the appropriate type when requested (and according to the rules set out in Section 2).

Car registration number

A car registration number has two components. The first component is two letters followed by two digits. The second component is three letters. For example:

- NG57 HXE

You must provide access to each component and an appropriate string representation of the registration number.

Registration numbers are unique. You must guarantee that no two cars have the same registration number.

Persons/Customers

A person (or customer) has a name and a date of birth.

A name consists of the first name and last name of a person.

You should use `java.util.Date` for the date of birth. You must not use deprecated methods of the `Date` class. So, in your test classes, you should use `java.util.Calendar` to construct a date of birth. You can assume default time zone and locale.

You must provide methods to access a person's name and date of birth.

Two people are considered to be the same if they have the same name and the same date of birth.

Driving licence

A driving licence has a unique number, a date of issue (represented by the `java.util.Date` class), and an indication whether the licence is a full driving licence or not.

The licence number has three components. The first component is the concatenation of the initial of the first name of the driver with the initial of the last name of the driver. The second component is the year of issue of the licence. The third component is an arbitrary serial number. For example, the string representation of the licence number for a licence issued to Mark Smith in 1990 would have the form:

- MS-1990-10

where the 10 is a serial number that, with the initials and year, guarantees the uniqueness of the licence number as a whole.

You must decide whether to define a separate class for the licence number. Whether or not you do, you must provide access to the individual components of the licence number and to its string representation.

You must guarantee the uniqueness of licence numbers.

Your driving licence must provide methods to access the licence number, the date of issue of the licence and whether it is a full licence or not.

---

4. Deliverables

Your solution should include your interfaces and classes that comprise the implementation of the system and types outlined in Sections 2 and 3. You must annotate your code with appropriate Javadocs. In addition, you should provide separate unit test classes that demonstrate thorough testing of your solution.

Also, write a document (MS Word, or PDF) that explains the structure of your overall design. This document should have a picture containing UML diagrams of all your classes and interfaces and their relationships depicted by annotated arrows (extends, implements, uses). For each class and interface, the UML diagram should show the class/interface name, fields and methods (where appropriate). Also, each diagram should indicate whether it is a class, abstract class or an interface.

You must submit your solution through NESS as a single zip archive that contains your Java source code files and your document.

---

5. Assessment & mark allocation

Marks will be allocated for:

- Overall structure (e.g. interfaces, classes and their relationships), **15%**
- Correct implementation of rules specified in Sections 2 and 3, as well as for choice and use of maps and collections, **40%**.
- Following good practice guidance: maintenance of invariants and defensive programming, use of immutability, appropriate overriding of `Object` methods, use of Javadoc comments, **25%**.
- Evidence of testing by implementation of appropriate test classes that test the normal case, boundary conditions, and exceptional cases. **20%**.

---

6. Style guidelines

Adopt a consistent style, do not violate naming conventions (when to use upper/lower case letters in names) and make appropriate use of whitespace (indentation and other spacing).

---

7. Further notes and hints

Break the coursework down into separate tasks. Start with the simpler classes first (e.g. for car registration number, person and driving licence). Unit test classes as you work through the coursework.

For each class you implement you should consider:

1. whether to override `Object` methods (`equals`, `toString` etc.),
2. whether to use an interface-based hierarchy, and
3. whether the class should be immutable.

For any questions, please email: Ellis.Solaiman@ncl.ac.uk