COSC1284 Programming Techniques
Semester 1 (2019)
Assignment 2

Due:	**Sunday 26 May, 2019** **11:59 p.m. AEST** **Late submissions accepted until** **Wednesday 31 May, 2019** **11:59 p.m. AEST**
Assignment Type:	**Individual (Group work is not permitted)**
Total Marks:	**150 marks (15%)**

Background information

For this assignment you need to write an object-oriented console application in the Java programming language which adheres to basic object-oriented programming principles shown below:

a) Your code should follow good object-oriented principles such as: encapsulation, composition, cohesion.

b) Setting the visibility of all instance variables to private.

c) Using getter and setter methods only where needed and appropriate with consideration given to scope (visibility).

d) Avoiding the use of static variables and methods.
Only using static attributes and methods when no other solution can be found.
You should not need to use the static keyword in this assignment.

c) Encapsulating both the data for a class and the methods that work with and/or manipulate that data within the same class.

d) Using superclass methods to retrieve and/or manipulate superclass properties from within subclass methods. Avoiding direct access to class attributes.

e) Taking advantage of polymorphism wherever possible when invoking methods upon objects that have been created and avoiding unnecessary casting.

This assignment builds on the previous assignment.

You may use your implementation of the assignment to continue with this one. If you use your own implementation you will need to update any issues to ensure it meets the expectations in this specification.

Alternatively, you may use the solution provided for the previous assignment as the basis for starting on this one.

Overview

The staff at **MiRide** are so happy with the first version of the program that they have asked you to upgrade it with a whole list of new features.

A user will be able to:

- Search for available cars for a given time and date.
- Search on the basis of either a standard or silver service car.
- Display cars sorted in order of their registration numbers.

You will be addressing these requirements by implementing a series of classes designed to meet these needs. The classes need to be designed according to object oriented principles.

Assignment 1 – Getting Started

This assignment is divided up into several stages, each of which corresponds to the concepts discussed during the course as shown below:

• Stage 1 - SilverServiceCar class	(Classes - Inheritance)	(30 marks)
• Stage 2 - MiRideSystem	(Updated)	(30 marks)
• Stage 3 - Exception Handling	(Exception Handling)	(30 marks)
• Stage 4 - Persistence to file	(File Handling)	(30 marks)
• Code Quality	(General Concepts)	(30 marks)

The assignment is designed to increase in complexity with earlier stages being easy and the later stages being more difficult. In addition, the specification will be more prescriptive in the earlier stages. The later stages of the assignment are less prescriptive and will require you to be more independent and resolve design issues on your own.

You are not permitted to use streams from Java 1.8.
You must use an array to store any collections of data in your program.
You must not use ArrayLists, Maps or any other data structures within the Java API or from 3rd party libraries.
Failing to comply with these restrictions will incur a 25% marking penalty.

Disclaimer:
While the scenario described is based upon the concept of an ride sharing booking system, the specification for this task is intended to represent a simplified version of such a system and thus is not meant to be a 100% accurate simulation of the any ride sharing booking system.

Stage 1 of the Assignment begins on the next page ...

Stage 1 - SilverService Heirarchy (Design / Implementation) (30 marks)

Make a backup of your program before continuing

This stage requires the implementing of a new class called **SilverServiceCar**

You may need to make changes to your Car class

You will be required to think carefully about the concepts you have learned in the course and make decisions in regards to modifying the classes to meet the stated requirements.

When you have finished this stage, your program should operate exactly as it did before but you will have re-organised and re-written parts of the code.

The **Car** and **SilverServiceCar** class share common attributes and behaviours as they are both **'bookable'** items. However, there will be additional attributes and behaviours that belong only to SilverService cars.

A **SilverServiceCar is a 'Car'**. + additional attributes & methods.

In object-oriented programming whenever we see an '**is a**' relationship. We model this relationship by implementing a class heirarchy.

A) Create the SilverServiceCar class as a sub-class of Car

Create a new class called **SilverServiceCar.**
The name of the class must be SilverServiceCar.

You will need to make sure that you maintain good object-oriented design by keeping the scope of the attributes to the most restrictive scope you can.

You will need to make your own decisions about the constructor and additional attributes required for this class.

You should implement appropriately scoped getters and setters only where they are required.

The constructor must have the following signature:

```
public SilverServiceCar(String regNo, String make, String model,
                  String driverName, int passengerCapacity,
                                    double bookingFee, String[] refreshments)
```

Business Rules:

1. The charges and services for a SilverService car are determined by the owner/driver of the car. In other words one SilverService car may have a different standard booking fee from another SilverService car.

   Note: these charges are set at the time the Car is created and does not change with each booking.

   a. The booking fee for a SilverService car is a fixed fee but must be greater than or equal to $3.00.
   b. SilverService cars offer a set list of refreshments determined by the owner/driver.
   c. The trip fee is calculated as kilometers travelled multiplied by 40% of the booking fee

B) Override the method for booking

You should now implement the overidden **booking** method in the **SilverService** class.

Business Rules:

1. All the business rules for a booking a car also apply for a SilverService car with the following additions/modifications.

2. SilverService cars can ONLY be booked up to 3 days in advance.

C) Update and override the method for getting details

You should now override the **getDetails** method to ensure that the full state of the object is returned to enable printing the details to the console. These details should now include any current bookings.

You should be careful not to violate object-oriented principles and refrain from having repeated code that exists in both the super and sub classes in the heirarchy.

Not all variations provided due to space constraints.

Sample output for a car with a mixture of bookings current and past.

```
Reg No:         ROV465
Make & Model:   Honda Rover
Driver Name:    Jonathon Ryss Meyers
Capacity:       7
Standard Fee: : $3.50

Refreshments Available
Item 1          Mints
Item 2          Orange Juice
Item 3          Chocolate Bar

CURRENT BOOKINGS
                ________________________________________

                id:                 ROV465NIGLAW23042019
                Booking Fee:        $3.50
                Pick Up Date:       23/04/2019
                Name:               Nigella Lawson
                Passengers:         3
                Travelled:          N/A
                Trip Fee:           N/A
                Car Id:             ROV465
PAST BOOKINGS
                ________________________________________

                id:                 ROV465NIGLAW21042019
                Booking Fee:        $3.50
                Pick Up Date:       21/04/2019
                Name:               Nigella Lawson
                Passengers:         3
                Travelled:          75.00
                Trip Fee:           105.00
                Car Id:             ROV465
```

D) Override the toString method

You should now override the toString method to ensure that the full state of the object is represented. These details should now include any current bookings.

You should be careful not to violate object-oriented principles and refrain from having repeated code that exists in both the super and sub classes in the heirarchy.

Standard Car Examples

Not Booked	HON865:Honda:Accord Euro:Henry Cavill:5:YES:1.5
Booked	BMW255:BMW:323:Barbara Streisand:4:YES:1.5|BMW255CRACOC23042019:1.5:23042019:Craig Cocker:3:0.0:0.0:BMW255
Completed	LAM314:Lamborghini:Urus:Robbie Williams:7:YES:1.5|LAM314RODCOC23042019:1.5:23042019:Rodney Cocker:3:**75.0**:**33.75**:LAM314

Silver Service Car Examples

Not Booked	NIS591:Nissan:Rover:Jonathon Ryss Meyers:7:YES:3.5:Item 1 Mints:Item 2 Orange Juice:Item 3 Chocolate Bar
Booked	JAG117:Jaguar:Rover:Jonathon Ryss Meyers:7:YES:3.5Item 1 Mints:Item 2 Orange Juice:Item 3 Chocolate Bar|JAG117NIGLAW23042019:3.5:23042019:Nigella Lawson:3:**0.0**:**0.0**:JAG117
Completed	FER659:Ferrarri:Rover:Jonathon Ryss Meyers:7:YES:3.5Item 1 Mints:Item 2 Orange Juice:Item 3 Chocolate Bar|FER659NIGLAW23042019:3.5:23042019:Nigella Lawson:3:**75.0**:**105.0**:FER659

After refactoring your program it should still be operating the same as it did at then end of the assignment 1 with the additions detailed above.

Do not continue with the next stage until you have fully tested your program to ensure it is still operating correctly after your refactoring work.

Stage 2 - MiRideSystem class (updated) (30 marks)

Make a backup of your program before continuing

YOU MUST COMPLETE STAGE 1 BEFORE ATTEMPTING STAGE 2

IF STAGE 1 HAS NOT BEEN COMPLETED, STAGE 2 WILL NOT BE ASSESSED

The next stage of this task is to update the **MiRidesSystem** application class implementation described in stage 2 so that it incorporates the ability to work with **SilverServiceCars**.

A description of the functions that need to be implemented for each of these features is provided below.

A) Add Car (Updated)

The add item function of your menu is now being used for adding either a **Car** or a **SilverServiceCar**.

Update your add function to now prompt the user to indicate if they wish to create a **Car** or a **SilverServiceCar.** Based on the user's choice it will now prompt for the appropriate input to finish creating the correct type of object.

```
Example 1:

Enter Registration No:          SIM194
Enter Make:                     Honda
Enter Model:                    Accord Euro
Enter Driver's Name:            Henry Cavill
Enter Passenger Capacity:       2
Enter Service Type (SD/SS)      SD

New Car added successfully for registion number: SIM194.


Example 2:

Enter registration number:      MAT412
Enter Make:                     Honda
Enter Model:                    Accord
Enter Driver Name:              Matt Bomer
Enter number of passengers:     4
Enter Service Type (SD/SS):     SS
Enter Standard Fee:             3.50
Enter List of Refreshments:     Mints,Orange Juice,Chocolate Bar

New Car added successfully for registion number: MAT412
```

The objects should be stored in the next available (empty) position in the same array used to store both Car and SilverService cars.

Stage 2 of the Assignment continues on the next page ...

B) Search for available cars (date & time)

This feature should begin by prompting the user to enter a date and a vehicle type i.e. Standard or SilverService. All cars that match the required details should have their full details printed to the console.

If no matching **Cars** were found then a suitable error message should be displayed to the screen.

```
Example 1:

Enter Type (SD/SS):       SD
Date:                     07/03/2019


"Error - No cars were found on this date."
```

If one or more cars were found then the full details should be printed to the console.

```
Example 1:

Reg No:           NIS591
Make & Model:     Nissan Rover
Driver Name:      Jonathon Ryss Meyers
Capacity:         7
Standard Fee:     3.5

Refreshments Available
Item 1 Mints
Item 2 Orange Juice
Item 3 Chocolate Bar

Example 2:

Reg No:          ROV465
Make & Model:    Jaguar Rover
Driver Name:     Jonathon Ryss Meyers
Capacity:        7
Standard Fee: :  $3.5

Refreshments Available
Item 1           Mints
Item 2           Orange Juice
Item 3           Chocolate Bar

PAST BOOKINGS
                 ________________________________________

                 id:                     ROV465NIGLAW23042019
                 Booking Fee:            $3.50
                 Pick Up Date:           23/04/2019
                 Name:                   Nigella Lawson
                 Passengers:             3
                 Travelled:              75.00km
                 Trip Fee:               $105.00
                 Car Id:                 ROV465
```

C) Display all cars (Registration - Ascending/Descending)

You should update the display all cars feature by prompting the user to enter either 'A' for ascending or 'D' for descending.

NOTE: You must implement your own sorting logic. You are not permitted to use any built in functions or libraries to achieve the sort. Failure to implement your own sorting will result in a 15 mark penalty.

All cars in the order specified should have their full details printed to the console

```
Example 1:

Enter Type (SD/SS):          SD

Enter Sort Order (A/D):      D

"Error - No standard cars were found on this date."
```

If one or more cars were found then the full details should be printed to the console.

```
Example 2:

Enter Type (SD/SS):        SD
Enter Sort Order (A/D):    D

The following cars are available
______________________________________________________________
Reg No:          LEX356
Make & Model:    Lexus M1
Driver Name:     Angela Landsbury
Capacity:        3
Standard Fee: :  1.5
Available:       YES

______________________________________________________________
Reg No:          LAM314
Make & Model:    Lamborghini Urus
Driver Name:     Robbie Williams
Capacity:        7
Standard Fee: :  1.5
Available:       YES
PAST BOOKINGS
                 _________________________________________

                 id:                    LAM314RODCOC23042019
                 Booking Fee:           $1.50
                 Pick Up Date:          23/04/2019
                 Name:                  Rodney Cocker
                 Passengers:            3
                 Travelled:             75.00
                 Trip Fee:              33.75
                 Car Id:                LAM314
```

D) Seed Data

Update your seed method so that it now pre-populates the collection with an additional 6 SilverServiceCars. The seeded data must contain the following variety of SilverServiceCars.

SilverService Cars

- Two silver service cars that **HAVE NOT been booked**
- Two silver service cars that **HAVE BEEN been booked, but the bookings have not been completed**
- Two silver service cars that **HAVE BEEN** booked, and **the bookings have been completed**

If this option is not selected at runtime, then the collection of items should be empty.

If the collection of items already has data then, this method should not overwrite the current data and an error message should be displayed to the user.

Stage 3 - Exception handling (30 marks)

Make a backup of your program before continuing

YOU MUST COMPLETE STAGE 2 BEFORE ATTEMPTING STAGE 3

IF STAGE 2 HAS NOT BEEN COMPLETED, STAGE 3 WILL NOT BE ASSESSED

It has been identified that there are potential issues with the program that cannot be prevented but must be dealt with at the time the program is running.

You should comply with these requests by making the following changes to your program.

NOTE: When you implement exceptions into your program you will need to modify the other classes in your program to work with the exeptions.

Exceptions will be generated and thrown in the **Car**, and **SilverServiceCar** classes.

Exceptions will be caught and handled in the **Menu and/or MiRideSystem** classes.

A) Creating the Exception Classes (Invalid Booking)

Define your own custom Exception subclass type called `InvalidBooking`, which represents an issue that occurs when a booking problem is found. For example:

- trying to book on a date prior the current day.
- trying to book a car on a day that it is already booked.
- trying to book when five current bookings exist.

This `InvalidDate` type should allow an error message to be specified when a new `InvalidDate` object is created. No additional details need to be recorded for this new `InvalidDate` type.

Your exception class should be named **InvalidDate** and it should be a sub-class of the Java **Exception** class.

B) Creating the Exception Classes (Invalid Refreshments)

Define your own custom Exception subclass type called **`InvalidRefreshements`**, which represents an issue that occurs when attempting create refreshments that are invalid.

For example:

- supplying a list of refreshments that contains duplicate items
- providing less than three items in the refreshment list.

Your exception class should be named **`InvalidId`** and it should be a sub-class of the Java **Exception** class.

C) Generating Exceptions

You will need to look for areas in your program where you wish to handle various errors such as :

- invalid dates (custom exception)
- invalid id's (custom exception)
- invalid numeric input (in built exception)

Then you should determine where in your program these errors occur and generate appropriate exceptions and then throw them back to the Menu class for handling.

D) Handling Exceptions

This various custom exceptions should then be allowed to propagate back up to the **MiRideSystem** class (ie. the exception **should not be caught locally** within the **Car** or **SilverService** classes).

Any exceptions will need to be caught and handled in an appropriate manner in the **MiRideSystem** class (by displaying the error message contained within the various exception objects that have been propagated up from the relevant method call).

Stage 4 - Persistence (File I/O) (30 marks)

Make a backup of your program before continuing

YOU MUST COMPLETE STAGE 4 BEFORE ATTEMPTING STAGE 4

IF STAGE 3 HAS NOT BEEN COMPLETED, STAGE 4 WILL NOT BE ASSESSED

DO NOT NEED TO RESTORE BOOKINGS ONLY THE STATE OF THE CAR ITSELF
This section of the program is designed to challenge and extend your ability to take a problem and workout how to solve it.

It is expected that **you will design your own solution** for this feature.

Your tutor can comment on your solution, but will not tell you how to write this section of the assignment.

Your program should incorporate file handling functionality so that it writes the details for each **Car** object currently in the **MiRideSystem** out to file when the program terminates
(i.e. when the user selects the "Exit" option in the menu).

The data that was previously written out to file should then be read back in automatically when the program is started up again and restore the program to the state it was in when the program exited.

This feature should write the data out to two files. One is the main data file, the other will be a duplicate copy to act as a backup of the data.

If the booking data file is not found in the local folder then the program should:

1.) First check for the presence of a backup file and if found use this backup file for loading the data.

2.) If loading from the backup file, display a message indicating that the data was loaded from a backup file.

3.) If no backup file is found, display a message indicating that no **Booking** data was loaded and continue on to the program menu without reconstructing any objects.

The format that you write the data out in is entirely at your own discretion as long as it is done to a text file.
Hint: You could use the toString method :-)

You must not use object serialisers to accomplish this task.

One aspect of this task is to record any changes that have been made during the previous run of the **MiRideSystem** application, so your file handling functionality must be able to handle the writing out and reading in of all details for both types of **Cars** in such a way that the state of the **MiRide** system at the point where the program was last exited is reconstructed in full when the program is started up again.

You can make any changes or include any additional methods that you deem necessary to the **Car** and **SilverServiceCar** classes to facilitate the writing out and reading in of the program state.

Note that the program design described earlier in the specification does not fully support what is required in this stage, so part of this task is identifying what aspects need to be considered when designing / implementing a solution.

Remember that you will still be required to adhere to basic object-oriented programming principles (eg. encapsulation, information hiding, etc) when designing your solution for this aspect of the program.

IMPORTANT NOTES:

These file reading / writing features are advanced functionality.

Marks will only be awarded for this section if the following conditions are met.

A) You must have completed all the previous stages before attempting this section.

B) Non-functional code or code which needs to be commented out in order to get the rest of the program to compile / run will receive zero marks in this section, as will code that just writes data out without any real intent of structuring that data to facilitate object persistence.

C) Also note that **you are not permitted to use automatic serialisation or binary files** when implementing your file handling mechanism - you must use a **PrintWriter** for writing data out to a text file and a **BufferedReader** or **Scanner** when reading the data back in from the same text file that data was written out to previously.

Coding Style (30 marks)

Your program should demonstrate appropriate coding style, which includes:

- **Formatting**

  Indentation levels of 3 or 4 spaces used to indent or a single tab provided the tab space is not too large - you can set up your IDE/editor to automatically replace tabs with levels of 3 or 4 spaces.

  A new level of indentation added for each new class/method/ control structure used.

  Indentation should return to the previous level of at the end of a class/method/control structure (before the closing brace if one is being used). Going back to the previous level of indentation at the end of a class/method/control structure (before the closing brace if one is being used)

  Block braces should be aligned and positioned consistently in relation to the method/control structure they are opening/closing.

  Lines of code not exceeding 80 characters in length - lines which will exceed this limit are split into two or more segments where required (this is a guideline - it's ok to stray beyond this by a small amount occasionally, but try to avoid doing so by more than 5-6 characters or doing so on a consistent basis).

  Expressions are well spaced out and source is spaced out into logically related segments

- **Good Coding Conventions**

  Identifiers themselves should be meaningful without being overly explicit (long) – you should avoid using abbreviations in identifiers as much as possible (an exception to this rule is a “generic” loop counter used in a for-loop).

  Use of appropriate identifiers wherever possible to improve code readability.

  All identifiers should adhere to the naming conventions discussed in the course notes such as ‘camel case’ for variables, and all upper case for constants etc.

  Complex expressions are assigned to meaningful variable names prior to being used to enhance code readability and debugging.

- **Commenting**

Note: the examples provided below do not reflect actual code you should have in your assignment but are used for demonstration purposes only.

Class Level commenting

Each file in your program should have a comment at the top listing your name, student number and a brief (1-2 line) description of the contents of the file (generally stating the purpose of the class)

```
/*
 * Class:          Booking
 * Description:    The class represents a single booking record for
 *                 any object that can be booked.
 * Author:         [student name] - [student number]
 */
```

Method Level commenting

At least one method in each class should have an algorithm written in pseudocode as a multi-line comment. You don't need to specify test cases for every algorithm, only if it is appropriate.

```
/*
 * ALGORITHM
 * BEGIN
 *       GET age of dog
 *       COMPUTE human years equivalent
 *             IF dog is two years old or younger
 *                   COMPUTE human years is dog's age multiplied by eleven
 *             ELSE
 *                   COMPUTE human years is equal to age minus two then multiplied by five
 *                   COMPUTER human years is equal to human years plus twenty two
 *       DISPLAY age
 * END
 *
 * TEST
 *       AGE is equal to 0, computation should not be performed
 *       AGE is equal to 1, human years is equal to 11
 *       AGE is equal to 2, human years is equal to 22
 *       AGE is equal to 3, human years is equal to 27
 */
```

Code block commenting

At least one method in each class should provide in line commenting to explain the purpose of a series of multiple statements, or a code block such as a loop or branching statement.

```
/* calculate the total value of the room rental
 * prior to applying surcharge.*/
int standardRate = 200;
int numberOfNights = 4;
int totalBeforeSurcharge = standardRate * numberOfNights;
```

Commented out code

Any incomplete or non-functional code should be removed from your implementation prior to your final submission.

Code that is not being used in your final implementation will attract marking penalties.

Examples of bad commenting

An example of a bad comment and/or coding style:

```
// declare an int value for storing the basic nightly rate for a room
int standardRoomRate = 200;

// declare and assign an initial account balance.
double x = 500.0;
```

What to Submit

You are being assessed on your ability to write a program in an object-oriented manner in this assignment. Writing the program in a procedural style is not permitted and will be considered inadmissable receiving a zero grade.

You should stick to using the standard Java API (version 1.8) when implementing your program. Note the restrictions noted earlier in this document in relation to the Java Collection Framework, Streams and the use of 3rd party libraries.

You should export your entire eclipse project to a zip archive and submit the resulting zip file - do this from within eclipse while it is running, not by trying to copy or move files around in the eclipse workspace directly, as you may corrupt your entire workspace if you do something wrong.

All students are also advised to check the contents of their zip files by opening them and viewing the files contained within before submitting to make sure they have done it correctly and that the correct (latest) version of the source code file is present to avoid any unpleasant surprises later on.

You should also download a copy of your submission from Canvas after submission, so that you can double check that you have submitted the correct version.

Technical issues that result in the loss of your work or submitting an incorrect version **WILL NOT** be considered a basis for extensions or re-marking.

We are obliged to accept each submission in the form it is sent to us in, so make sure you submit the corrrect final version of your program!

The name of both your Eclipse project and your zip archive should be your student number followed by an underscore, then append A2. For example:

Project Name: s123456_A2
Zip Archive: s123456_A2.zip