COSC1284 Programming Techniques
Semester 1 (2019)
Assignment 1

Due:	**12.04.2019 (Friday)** **9 am AEST** **i.e. end of week 6** **Late submissions accepted until** **17.04.2019 with penalty** **9 am AEST**
Assignment Type:	**Individual (Group work is not permitted)**
Total Marks:	**100 marks (10%)**

Background information

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

a) Follow good object-oriented principles such as : encapsulation, composition, cohesion.

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

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

d) Avoid the use of static variables and methods. Only use static attributes and methods when no other solution can be found.

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

f) Use superclass methods to retrieve and/or manipulate superclass properties from within subclass methods. Avoid accessing variables directly.

Overview

For this assignment you need to write a console application in the Java programming language which simulates booking a new ride sharing service called **MiRide**.

The staff at **MiRide** will need to be able to maintain the details for bookings that are available for **MiRide's** customers. Some general requirements are listed below as an overview. Additional requirements will be stated in the individual sections of the assignment.

A user will be able to:

- search for available cars for a given time and date.
- select a car and make a booking for that car.
- make payment at the conclusion of the trip.

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 - Car class	(Classes)	(25 marks)
• Stage 2 - Booking class	(Classes)	(25 marks)
• Stage 3 - Menu & MiRideSystem	(Menu and work with Arrays of objects)	(25 marks)
• Code Quality	(General Concepts)	(25 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.

Note:

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 50% marking penalty.

Disclaimer:

While the scenario described is based upon the concept of a ride sharing booking system, the specification for this task is intended to represent a simplified version of such a system and is not meant to be a verbatim simulation of the any such ride sharing booking system(s).

Stage 1 - Car (Design / Implementation) (25 marks)

Some specific instructions are given below to get you started with the basics, but you will need to use the knowledge and skills you are developing in the course to further develop these classes and solve any problems you encounter as required.

You need to make sure that any modifications/changes you make adhere to good object-oriented concepts and design.

Each **Car** will have associated **Booking** objects. You will need to create both classes before proceeding ahead. Once you have created these classes, you can continue with filling out the implementation of the Car class as mentioned below:

A) Create the Car Class (5.0 marks)

The **Car** class represents a single car.

The following attributes should be defined in your **Car** class.

You should **NOT** define any other instance variables unless there is a good reason to do so. You should think very carefully before adding any additional instance variables.

Instance variable	Type	
regNo	String	
make	String	
model	String	
driverName	String	
passengerCapacity	int	
available	boolean	
currentBookings	[Booking]	Note: you must use an array
pastBookings	[Booking]	Note: you must use an array

B) Constructor (5.0 marks)

```
public Car(String regNo, String make,
String model, String driverName, int passengerCapacity)
```

You should **NOT** have a default constructor.

The constructor is responsible for initialising the object into a valid state at the time of construction.

You should **NOT** modify the definition of this constructor.

Business Rules:

1. The registration number should be exactly six characters that consist of an initial 3 alphabetical characters, followed by 3 numeric characters.
2. The make, model and driver name must handle being able to have multiple words.
3. Passenger capacity must be greater than 0 and less than 10.

C) Implement a method for booking the car (5.0 marks)

```
public boolean book(String firstName, String lastName, DateTime required,
int numPassengers)
```

This method will be responsible for updating the state of the car object to indicate if it is currently booked or available for booking. The method should ensure that only valid bookings are created.

Business Rules:

1. For simplicity a car can only be booked once per day, so you need not to worry about the time of the booking or managing multiple bookings on the same day.
2. A car that has more than 5 current bookings will be considered as unavailable for further bookings until the number of current bookings has fallen below 5.
3. Each car may have zero or more associated bookings. Any bookings associated with a car are defined as either current or past bookings.
4. Bookings cannot be made for a past date.
5. The total number of passengers cannot exceed the passenger capacity.
6. A booking should be placed into the collection of current bookings for a car.
7. The standard booking fee for a car is $1.50.
8. The trip fee is calculated as kilometres travelled multiplied by 30% of the booking fee.

Please note: As you are yet to implement the booking class the full implementation of this method will have to be finished after you have created the booking class in the next stage of the assignment.

D) Implement a method for getting all the details of a Car object (5.0 marks)

```
public String getDetails()
```

This method should build and return a string. The returned string should be formatted in a human readable format as shown below. This method should not do the actual printing.

The description must include labels and values for all the instance variables as shown in the examples below. Note these are just a couple of sample outputs.

Sample output for a car that **IS** available.

```
RegNo:         SIM194
Make & Model:  Honda Accord Euro
Driver Name:   Henry Cavill
Capacity:      5
Available:     YES
```

Sample output for a car that **IS NOT** available.

```
RegNo:         SIM194
Make & Model:  Honda Accord Euro
Driver Name:   Henry Cavill
Capacity:      5
Available:     NO
```

E) Implement a toString method (5.0 marks)

```
public String toString()
```

This method should build a string and return it to the calling method. The returned string should be formatted in a pre-defined format designed to be read and processed by a computer.

Format for the toString method is a colon separated list of the instance variables as shown below:

```
regNo:make:model:driverName:passengerCapacity:available
```

Sample output for a car that **IS** available.

```
SIM194:Honda:Accord Euro:Henry Cavill:5:YES
```

Sample output for a car **IS NOT** available.

```
SIM194:Honda:Accord Euro:Henry Cavill:5:NO
```

Write some test code in the main method of your program to thoroughly test the class before moving on to the next section.

Backup your program once the testing is complete.

Remove any test code from your main method and move on to the next stage →

Stage 2 - Booking (Design / Implementation) (25 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

A) Create the Booking Class (6.0 marks)

The following attributes should be defined in your **Booking** class.

You should define other instance variables **ONLY** if necessary and where appropriate.

You should think very carefully before adding any additional instance variables to make sure that you do not creating additional variables that are unnecessary.

You should also create additional constants where appropriate avoiding the use of any hard-coded values.

Instance variable	Type
id	String
bookingFee	double
pickUpDateTime	Date/Time
firstName	String
lastName	String
numPassengers	int
kilometersTravelled	double
tripFee	double
car	Car

B) Constructor (7.0 marks)

You should **NOT** have a default constructor.

You should **NOT** modify the definition of this constructor.

```
public Booking(String firstName, String lastName, DateTime required,
                     int numPassengers, Car car)
```

Business Rules:

1. The date of the booking must not be in the past or more than one week in the future.
2. The first name must have a minimum of three characters.
3. The last name must have a minimum of three characters.
4. Each booking must have only one car associated with the booking.
5. The id will be automatically generated and will consist of the following three parts each separated by an underscore:
   - the registration number of the car that has been booked.
   - a six-character representation of the name (first three characters of the first name + the first three characters of the last name.
   - an eight-character representation of the date the car has been booked.

Example:

REGNO	+	Passenger	+	DATE
SIM194	+	Matt Bomer	+	5th March, 2019
SIM194	+	MATBOM	+	05032019

Booking ID: `SIM194_MATBOM_05032019`

C) Implement a method for getting all the details of a Booking object (6.0 marks)

```
public String getDetails()
```

This method should build and return the string. The returned string should be formatted in a human readable format as shown below. This method should not do the actual printing.

The description must include labels and values for all the instance variables as shown in the examples below. Note these are just a couple of sample outputs.

Sample output for a car record that represents a standard car that **HAS BEEN** booked and **HAS NOT BEEN** completed.

```
id:           SIM194_MATBOM_05032019
Booking Fee:  $1.50
Pick up Date: 05/03/2019
Name:         Matt Bomer
Passengers:   2
Travelled:    N/A
Trip Fee:     N/A
Car Id:       SIM194
```

Sample output for a car record that represents a standard car that **HAS BEEN** booked and **HAS BEEN** completed.

```
id:           SIM194_MATBOM_05032019
Booking Fee:  $1.50
Pick up Date: 05/03/2019
Name:         Matt Bomer
Passengers:   2
Travelled:    43km
Trip Fee:     19.35
Car Id:       SIM194
```

D) Implement a toString method (6.0 marks)

```
public String toString()
```

This method should build and return it to the calling method. The returned string should be formatted in a pre-defined format designed to be read and processed by a computer.

Format for the toString method is a colon separated list of the instance variables as shown below:

```
id:booking fee:date:name:passengers:distance:trip fee:car id
```

Sample output for a booking record that represents a silver service car that **HAS** been booked but **HAS NOT** been completed.

```
SIM194_MATBRO_05032019:1.5:05032019:Matt Bomer:2:0.0:0.0:SIM194
```

Sample output for a car record that represents a standard car that **HAS BEEN** booked and **HAS BEEN** completed.

```
SIM194_MATBRO_05032019:1.5:05032019:Matt Bomer:2:43.0:19.35:SIM194
```

Write some test code in the main method of your program to thoroughly test the class before moving on to the next section.

Backup your program once testing is complete.

Remove any test code from your main method and move on to the next stage→

Stage 3 - Menu & MiRidesApplication class (basic functionality) (25 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

Here, you will begin with the implementation of the **MiRidesApplication** application class, which will use an array of **Car** references called '**cars**' to store and manage all the cars that are added to the system by the user and the bookings associated with those cars.

Note:

You are not permitted to use streams from Java 1.8.

You must use an array to store any collections of objects.

You are not permitted to use inbuilt or 3rd party collections such as ArrayLists or Maps.

Failing to comply with these restrictions will incur a 50% marking penalty.

1.) You should:

A) create a class called **'MiRidesApplication'** for managing all the cars and bookings

B) create a class to present a menu to the user for interacting with the program.

2.) The menu options that must be presented to the user are:

```
*** MiRides System Menu ***

Create Car                      CC
Book Car                        BC
Complete Booking                CB
Display ALL Cars                DA
Search Specific Car             SS
Search available cars           SA
Seed Data                       SD
Exit Program                    EX
```

You must implement the menu as shown. Do not change the options or the inputs for selecting the menu options.

(Note: the letters on the right represent what the user must type to use that feature.
In other words, to add a new Car, the user must input 'CC'.

The solution should be case insensitive, that is the user should be able to enter either ‘cc’ or ‘CC’)

Your task is to work on this initial **MiRidesApplication** class by implementing the features shown in the menu. A description of the functionality that needs to be implemented for each of these features is provided below.

A) Create two classes required and implement Create Car method (5.0 marks)

This first feature **'Create Car'** should prompt the user to enter all relevant details for a **Car**.

The Menu class and the MiRides class together should check data to ensure that incorrect data is not used to create the Car objects. You will need to ensure that the program does not violate any of the requirements. For example:

> The Menu class should perform data validation to ensure that incorrect data is picked up at the point of entry such as a registration number that is not six characters and in the correct format. If this check fails, then should display an appropriate error message to the console and the program should go back to the menu immediately without creating or storing a new **Car** object in the array.
>
> The MiRidesApplication class should check to ensure that the registration entered is not already in the system. If this check fails then should display an appropriate error message to the console and the program should go back to the menu immediately without creating or storing a new **Car** object in the array.

If the Car object is created then it should be added to the next (empty) spot in the array instance that you are using to store the **Car** objects.

You are not permitted to implement auto generated user inputs such as registration numbers, but must validate those entered by the user.

Example Outputs

```
Example 1:

Enter Registration No:            SIM194
Enter Make:                       Honda
Enter Model:                      Accord Euro
Enter Driver's Name:              Henry Cavill
Enter Passenger Capacity:         2

New Car added successfully for registion number: SIM194.


Example 2:

Enter Registration No:            SIM194
"Error - already exists in the system."


Example 3:

Enter Registration No:            RICCAC
"Error - Registration number is invalid."
```

B) Book Car **(4.0 marks)**

This feature should begin by prompting the user to enter the details need to find an available car.

```
Enter Date Required:              06/03/2019
```

Once the user has entered the required date, the feature should then attempt to locate a corresponding **Car** object(s) within the array of **Car** references described above.

If a matching **Car** with the specified detail was not found then a suitable error message should be displayed to the screen.

```
Example 1:

Enter Date Required:              06/03/2019

Error - No cars are available on this date
```

If one or more cars are found, then it should display the list of cars available and allow the user to select one of them to complete the booking.

Once the car has been selected, then it should prompt the user for a first name, last name and number of passengers to complete the booking.

```
Example 2:

Enter Date Required:              05/03/2019

The following cars are available.

1.  SIM194
2.  RIC847

Please select the number next to the car you wish to book: 1

Enter First Name:                 Matt
Enter Last Name:                  Bomer
Enter Number of Passengers:       2

Thank you for your booking. Henry Cavill will pick you up on 05/03/2019.
Your booking reference is: SIM194_MATBOM_05032019.
```

C) Complete Booking (4.0 marks)

NOTE: This feature was not dealt with earlier when you created the Car and Booking classes. You will need to think about what changes will be required in these classes to enable your to add this feature to your menu.

This feature should begin by prompting the user to enter:

- the registration number or date of the booking
- first name and last name of the person who made the booking.

Once the user has entered the details, the function should then attempt to locate the corresponding **Car** or **Booking** object within the system.

If a matching **Booking** was not found, then a suitable error message should be displayed to the screen.

```
Example 1:

Enter Registration or Booking Date: RIF847
Enter first name:    Matt
Enter last name:     Bomer
Error - The booking could not be located.


Example 2:

Enter Registration or Booking Date: 06/03/2019
Enter first name:    Matt
Enter last name:     Bomer
Error - The booking could not be located.


Example 3:

Enter Registration or Booking Date: 05/03/2019
Enter first name:    Henry
Enter last name:     Cavill
Error - The booking could not be located.
```

If the booking was successfully located, then the booking should be completed, and a suitable message displayed to the user on the console.

```
Example 4:

Enter Registration or Booking Date: 05/03/2019
Enter first name:   Matt
Enter last name:    Bomer
Enter kilometers:   43

Thank you for riding with MiRide.  We hope you enjoyed your trip.

$20.85 has been deducted from your account.
```

Note: this is the total of the trip fee and the booking fee.

D) Display specific car (4.0 marks)

This feature should begin by prompting the user to enter the registration number of the car. Once the user has entered the details, the function should then attempt to locate the corresponding **Car** object within the array of **Car** references described above.

If a matching **Car** was not found, then a suitable error message should be displayed to the screen.

```
Example 1:

Enter Registration No:     RIF847
Error - The car could not be located.
```

If the car was found, then the full booking details should be printed to the console.

```
Example 1:

Enter Registration No:    SIM194
RegNo:                    SIM194
Make & Model:             Honda Accord Euro
Driver Name:              Henry Cavill
Capacity:                 5
Available:                NO
```

E) Display All Cars (4.0 marks)

This feature should simply display the details of all the cars in the system. You should ensure that your program does not crash if there are no cars currently in the system.

F) Menu System & Seed Data method **(4.0 marks)**

You should implement a method called '**seedData'** that pre-populates the system with a range of cars and booking objects. When the seed data menu item is selected 6 hard coded cars will be instantiated and added to the collection with the booking states as indicated below.

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

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

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

You must ensure that your program does not crash under any circumstances, so you will need to thoroughly test your program before final submission.

Coding Style (25 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 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.

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 correct 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 A1. For example:

Project Name:	s123456_A1
Zip Archive:	s123456_A1.zip

Your submission will be penalised heavily if you fail to meet any of the above guidelines.

Submission details

This assignment will be marked out of a total of **100 marks** and contributes **10%** towards your result for this course.

Due Date

This due date of the assignment is listed on the first page of this specification. You must submit your final version to Canvas prior to this date/time to avoid incurring late penalties.

Please refer to the extensions document for information on valid extension requests.