**Due Date** – Monday, September 29, by 11:59 pm.

**Submission**

(1) Zip the project folder and submit the zipped file to Canvas.
   (a) Windows – right-click the project folder and select **Compress to Zip file**
   (b) Mac/Linux – right-click the project folder and **select Compress**

(2) The zip file must include the following grading items.
- **The source folder "src",** containing the Java files (*.java) [**100 points**]
  (a) MUST create a Java package to hold the source files, or **lose 2 points**
  (b) MUST use all lowercase letters for the package name, or **lose 2 points**
  (c) MUST include the **@author** tag in the comment block on top of EVERY Java class and put the name(s) who implemented the Java class, or **lose 2 points**
- **Test specification**, including the test cases designed to test the `isValid()` method of the `Date` class and the `compareTo()` method of the `Vehicle` class. [**17 points**]
- Testbed **`main()`** in the following Java classes implementing the test cases in the Test Specification.
  (a) Date class [**12 points**]
  (b) Vehicle class [**6 points**]
- The **Javadoc folder** "**doc",** to include all the ***.html** files generated by Javadoc. [**5 points**]

(3) The submission button on Canvas will disappear after **September 29, 11:59 pm**. Do not wait until the last minute to submit the project. You are responsible for ensuring the project is well-received in Canvas by the due date and time. **You get 0 points** if you do not have a submission on Canvas. **Projects sent through emails or other methods will not be accepted.**

**Project Description**

The RU operation department is managing a fleet of vehicles available for employees' business trips. Your team is tasked with developing a software system to manage the operations of the fleet. The activities include adding/removing the information of a vehicle to/from the fleet, making/canceling the reservation of a vehicle for a business trip, and returning the vehicle when completing a trip. Only eligible employees (with the departments) listed below can book a vehicle.

Patel (Computer Science)
Lim (Electrical Engineering)
Zimnes (Computer Science)
Harper (Electrical Engineering)
Kaur (Information Technology and Informatics)
Taylor (Math)
Ramesh (Math)
Ceravolo (Business Analytics and Information Technology)

The RU operation department has given you the list of requirements for the software system your team will develop.

1. The system shall maintain the information of the fleet that the department manages. Each vehicle is uniquely identified by the license plate number.
2. For each vehicle, the system shall keep track of the license plate number, date obtained, make, and current mileage on the odometer. There are four different makes in the fleet: FORD, CHEVY, TOYOTA, and HONDA.
3. The system shall maintain a list of bookings (reservations) for the upcoming business trips.

4. For each booking, the system shall keep track of the beginning and ending dates of the trip, the vehicle being booked, and the employee who booked it.
5. The system shall maintain a list of completed business trips. For each trip, the system shall keep track of the original booking details, beginning mileage, and ending mileage of the trip.
6. The system shall be able to process the vehicle activities, including adding or removing vehicles, booking or canceling a vehicle for a trip, and returning a vehicle when a trip is completed.
7. The system shall only allow eligible employees to book a vehicle for a business trip.
8. The system shall not allow a beginning date of a booking that is more than 3 months away from today.
9. The system shall not allow the duration of a booking that is more than 7 days.
10. When returning a vehicle, the system shall remove the original booking from the list of bookings and add the booking details to the list of completed trips.
11. The system shall not allow a vehicle return in which the booking end date is not the earliest ending date in the list of bookings.
12. When returning a vehicle, the new mileage on the odometer must be provided. The system shall not accept a mileage that is less than or equal to the original mileage on the odometer.
13. The system shall be able to display the list of vehicles ordered by the car make and the obtained date.
14. The system shall be able to display the list of bookings ordered by the license plate number and the beginning date, or ordered by the department and the employee who booked the vehicle.
15. The system shall be able to display the list of completed trips ordered by the booking end date.

The software system shall be an interactive system where the users enter command lines on the terminal, and the system immediately outputs the results on the terminal. That is, when the user enters a command line and hits the enter key, the system reads the data entered, processes the data, and immediately displays the results.

A command line represents a vehicle activity that begins with a command followed by additional data tokens delimited by one or more spaces. Commands are in uppercase letter(s) and **case-sensitive**, which means the commands in lowercase letters are invalid. However, the data tokens are NOT case-sensitive; for example, "TAYLOR" and "taylor" are considered equivalent. Below is a list of commands the software system must support.

- **A** command: to **add a vehicle** to the fleet. After the A command, the user shall enter the vehicle information in the following sequence: the license plate number, date obtained, make, and current reading on the odometer. Below is an example of an activity for adding a new vehicle. You can assume that the user will always enter enough data tokens if a valid command is entered. However, there might be more than one space delimiting the data tokens.

  ```
  A 58719D 1/31/2017 TOYOTA 87120
  ```

  The above command line adds a new vehicle to the fleet. The date shall be given in the **mm/dd/yyyy** format. To prevent garbage data from entering the system and ensure the quality of software, your software must check if the data tokens are valid in the order of the data tokens being read from left to right, as follows. You can assume the license plate number is always correct for this project.

  1. The date obtained should be a valid calendar date.
  2. The date obtained should not be today or a future date.
  3. The make should be one of the four makes that the system manages.
  4. The mileage should not be 0 or negative.
  5. The vehicle being added should not be in the fleet already.
- **D** command: to **remove a vehicle** from the fleet; for example, the command line: `D 58718D` is to delete the vehicle with the license plate `58718D` from the fleet. The license plate number given may not exist in the fleet. Note that if the list of bookings contains the bookings of this vehicle, the system shall not process the command line to delete the vehicle from the system.

- **B** command: to **book a vehicle** for a business trip. For example, the command line below makes a reservation of the vehicle `58718D` beginning 10/25/2025 and ending 10/25/2025, booked by the employee `KAUR`.

  ```
  B 10/25/2025 10/25/2025 58718D KAUR
  ```

  The system shall check the conditions below in the order of the data tokens entered above, from left to right, before the reservation can be made.

  1. The beginning date should be a valid calendar date, be today or a future date.
  2. The beginning date should not be a date that is beyond 3 months from today.
  3. The ending date should be a valid calendar date, be the same as the beginning date, or a date after the beginning date.
  4. The duration from the beginning date to the ending date should not exceed 7 days.
  5. The license plate number does not exist in the fleet.
  6. The vehicle associated with the license plate number is not available for the dates entered.
  7. The employee is not eligible to book a vehicle.
  8. The employee has an existing booking conflicting with the dates entered.
- **C** command: to **cancel an existing booking**. For example, the booking below, identified by the license plate number, the beginning and ending dates, will be removed from the list of bookings.

  ```
  C 11/21/2025 11/25/2025 58718D
  ```

  Since the system checks the necessary conditions when it adds the bookings, the system shall only check if the booking entered above exists in the list of bookings.
- **R** command: to **return the vehicle** when a trip is completed. For example, the command line below returns a vehicle, including the ending date, license plate, and the new mileage. Note that, for simplicity of the project, we **assume the returning date is the earliest ending date** in the list of bookings. Thus, the date below is the ending date of the booking. If the ending date entered is not the earliest ending date in the list of bookings, the system should not process the command line.

  ```
  R 8/24/2025 58719D 87170
  ```

  The system checks if the data tokens are valid in the order entered above, from left to right, as listed below.

  1. The booking associated with the ending date and vehicle exists in the list of bookings.
  2. The ending date should be the earliest ending date in the list of bookings.
  3. The new mileage should not be 0 or negative; it should be positive and greater than the original mileage when it was booked.
- **P commands,** to display the information, sorted with different keys. If the keys for sorting are the same, the order of display for those lines does not matter.
  - ✓ **PF** command: to display the list of vehicles, ordered by make, then by date obtained.
  - ✓ **PR** command: to display the list of bookings, ordered by license plate, then by beginning dates.
  - ✓ **PD** command: to display the list of bookings, ordered by department, then by employee.
  - ✓ **PT** command: to display the list of completed trips, ordered by ending date.
- **Q** command to stop running the software system.

**Project Requirement**

1. You MUST follow the Coding Standard posted on Canvas under Modules/Week #1. **You will lose points** if you violate the rules listed in the coding standard.

2. You are required to follow the Academic Integrity Policy. See the **Additional Note #14** in the syllabus posted on Canvas. If your team uses a repository hosted on a public website, you MUST **set the repository to private**. Setting it to public is considered a violation of the academic integrity policy. The consequences of violation of the Academic Integrity Policy are: **(i) all parties involved receive 0 (zero) on the project, (ii) the violation is reported, and (iii) a record of this violation is in your file.**
3. Test cases for grading are included in the file **Project1TestCases.txt.** The data for the test cases are fictional for testing purposes. DO NOT change the spacing or sequence in the file; use it as is. The graders will run your project by copying all the test cases in **Project1TestCases.txt** and pasting them to the terminal. Enter the test cases from the file in the same order to test your project.
4. The associated output generated from the test cases is in **Project1Output.txt**. You must match the output provided. Your project should be able to ignore the empty lines between the test cases in **Project1TestCases.txt**. You MUST use the **Scanner class** to read the command lines from the terminal (**System.in**); DO NOT read it as an external file, or you will **lose 5 points**. The graders will compare your output with the expected output in **Project1Output.txt**. You will **lose 2 points** for each output not matching the expected output, OR for each exception, causing your project to terminate abnormally.
5. Each source file (.java file) can only include one public Java class; the file name is the same as the Java class name, or **lose 2 points for each violation**.
6. Your program MUST handle invalid commands, or **lose 2 points** for each bad command not handled.
7. You CANNOT import any Java library classes, EXCEPT the **Scanner**, **StringTokenizer, Calendar,** and **DecimalFormat** classes. **You will lose 5 points** for EACH additional Java library class imported, with a **maximum of losing 10 points**. You CANNOT use the Java library class **ArrayList** anywhere in the project, or use any Java library classes from the Java Collections, or **you will get 0 points for this project**!
8. Be specific when importing Java library classes; DO NOT import unnecessary classes or the whole package. For example, `import java.util.*;` this will import all classes in the `java.util` package. You will **lose 2 points** for using the asterisk "*" to include all the Java classes in the `java.util` package, or other Java packages, with a **maximum of losing 4 points.**
9. You can ONLY use **System.in** or **System.out** statements in the user interface class `Frontend.java`, the testbed `main()` methods and the **print()** methods. You **lose 2 points** for using **System.out** in other places, with a **maximum of losing 10 points**. This is to adhere to the MVC (Model-View-Controller) design pattern.
10. You MUST implement the Java classes below, or **lose 5 points** for each class missing or NOT used. You must always add the **@Override** tag for overriding methods, or **lose 2 points** for each violation.

(a) **Vehicle class**

```
public class Vehicle implements Comparable<Vehicle> {
    private String plate;     //license plate number
    private Date   obtained; //Date class described in the next page
    private Make   make;      //Make is an enum class
    private int    mileage;  //current reading on the odometer
}
```

- You should add necessary constants, constructors, and methods. However, you CANNOT change or add the instance variables, or **lose 2 points** for each violation.
- Must override **equals()**, **toString()** and the **comapreTo()** methods, with the **@Override** tags, or **lose 2 points** for each violation. The **equals()** method returns true if two vehicles have the same license plate; it returns false otherwise. The **toString()** method returns a textual representation of a vehicle as follows, including all the values stored in the instance variables. For example,

```
67155S:CHEVY:3/1/2010 [mileage:101230]
```

(b) **Date class**

```
public class Date implements Comparable<Date> {
    private int year;
    private int month;
    private int day;
    public  boolean isValid() {} //check if a date is a valid calendar date
}
```

- You can add necessary constants, constructors, and methods; however, you CANNOT change or add instance variables, or **lose 2 points** for each violation.
- You MUST override **equals()**, **toString()** and the **comapreTo()** methods, with the **@Override** tags, or **lose 2 points** for each violation.
- The **isValid()** method checks if a date is a valid calendar date. There are months that have 31 days or 30 days; however, February has 28 days in a non-leap year, and 29 days in a leap year. DO NOT use magic numbers in your code for the months, days, and years. You can use the constants defined in the **Calendar** class or define your own constant names. For example,

```
public static final int QUADRENNIAL = 4;
public static final int CENTENNIAL = 100;
public static final int QUATERCENTENNIAL = 400;
```

To determine whether a year is a leap year, you may define a **isLeap()** method; follow these steps:

Step 1. If the year is evenly divisible by 4, go to step 2. Otherwise, go to step 5.
Step 2. If the year is evenly divisible by 100, go to step 3. Otherwise, go to step 4.
Step 3. If the year is evenly divisible by 400, go to step 4. Otherwise, go to step 5.
Step 4. The year is a leap year.
Step 5. The year is not a leap year.

(c) **Make class.** This is an enum class that defines the four car makes of the vehicles.

(d) **Fleet class.** This is a resizable array-based implementation of a linear data structure to hold the list of vehicle objects. The initial capacity of the list is 4 and automatically grows in capacity by 4 if the list is full. The list does not decrease in capacity. You must implement the API specified below.

```
public class Fleet {
    private static final int CAPACITY  = 4; //initial capacity
    private static final int NOT_FOUND = -1;
    private Vehicle[] fleet;
    private int       size; //current number of vehicles in the fleet

    private int     find(Vehicle vehicle) {}   //search the given vehicle
    private void    grow() {}                  //resize the array
    public  void    add(Vehicle vehicle) {}    //add to end of array
    public  void    remove(Vehicle vehicle) {} //overwrite with last element
    public  boolean contains(Vehicle vehicle) {}
    public  void    printByMake() {} //ordered by make, then date obtained
}
```

- You can add necessary constants, constructors, and methods. However, you CANNOT change or add instance variables, or **lose 2 points** for each violation.
- You CANNOT change the method signatures of the methods listed above, or **lose 2 points** for each violation. Note that you CAN overload the methods if necessary.
- **System.out** allowed ONLY in the **printByMake()** method or private helper print() methods, or **lose 2 points** for each violation.

- The **find()** method searches for a vehicle in the list and returns the index if it is found; it returns `NOT_FOUND` if it is not in the list.
- You must use an **in-place sorting algorithm** to implement the sorting, such as selection sort or insertion sort. That is, the order of the objects in the array will be rearranged after the sorting without using an additional array. You CANNOT use **Arrays.sort()** or **System.arraycopy()** or any other Java library classes or utilities for sorting. You must write the code yourself to sort the list. You will **lose 10 points** for the violation.

(e) **Booking class.** This class holds the information of a single booking (reservation).

```
public class Booking {
    private Date     begin;
    private Date     end;
    private Employee employee; //Employee is a enum class
    private Vehicle  vehicle;
}
```

- You should add necessary constants, constructors, and methods. However, you CANNOT change or add the instance variables, or **lose 2 points** for each violation.
- Must override **equals()** and **toString()** methods, with the **@Override** tags, or **lose 2 points** for each violation. The **equals()** method returns true if two bookings have the same vehicle, beginning, and ending dates; it returns false otherwise. The **toString()** method returns a textual representation of a booking, including all the values stored in the instance variables. For example,

```
67359S:FORD:11/1/2019 [mileage:59644] [beginning 10/31/2025 ending 11/2/2025:KAUR]
```

(f) **Employee class** is an enum class defining the last names of the employees with an additional attribute department: `private Department dept;`

(g) **Department class** is an enum class defining the department names.

(h) **Reservation class.** This is a resizable array-based implementation of a linear data structure to hold the list of bookings. The initial capacity of the list is 4 and automatically grows in capacity by 4 if the list is full. The list does not decrease in capacity. You must implement the API specified below.

```
public class Reservation {
    private Booking[] bookings;
    private int       size;

    private int     find(Booking booking) {}   //search the given booking
    private void    grow() {}                  //resize the array
    public  void    add(Booking booking) {}    //add to end of array
    public  void    remove(Booking booking) {} //overwrite with last element
    public  boolean contains(Booking booking) {}
    public  void    printByVehicle(){} //ordered by plate then beginning date
    public  void    printByDept() {} //ordered by department then by employee
}
```

- You can add necessary constants, constructors, and methods. However, you CANNOT change or add instance variables, or **lose 2 points** for each violation.
- You CANNOT change the method signatures of the methods listed above, or **lose 2 points** for each violation. Note that you CAN overload the methods if necessary.
- **System.out** statements are allowed ONLY in the **print()** methods, or **lose 2 points** for each violation.

- The **find()** method searches for a booking in the list and returns the index if it is found; it returns `NOT_FOUND` if it is not in the list.
- You must use an in-place sorting algorithm to implement the sorting, such as selection sort or insertion sort. That is, the order of the objects in the array will be rearranged after the sorting without using an additional array. You CANNOT use **Arrays.sort()** or **System.arraycopy()** or any other Java library classes or utilities for sorting. You must write the code yourself to sort the list. You will **lose 10 points** for the violation.

(i) **Trip class.** This class holds the information of a completed trip.

```
public class Trip {
    private Booking booking;
    private int beginMileage;
    private int endMileage;
}
```

- You should add necessary constants, constructors, and methods. However, you CANNOT change or add the instance variables, or **lose 2 points** for each violation.
- Must override **equals()** and **toString()** methods, with the **@Override** tags, or **lose 2 points** for each violation. The **equals()** method returns true if two trips have the same booking details; it returns false otherwise. The **toString()** method returns a textual representation of a trip, including all the values stored in the instance variables. For example,

```
58718D 10/15/2025 ~ 10/15/2025 original mileage: 64390 current mileage: 64500 mileage used: 110
```

(j) **TripList class.** This class implements **a circular linked list** to store the list of completed trips. You cannot add or change the instance variable and must implement the API specified below, or **lose 2 points** for each violation. You should define the Node class for a node in the linked list.

```
public class TripList {
    private Node last; //the reference to the last node of the linked list.

    public void add(Trip trip) {}
    public void print() {} //print the list ordered by the ending date
}
```

(k) **Frontend class**

- This is the user interface class to process the command lines entered on the terminal. An instance of this class can process a single command line or multiple command lines at a time, including the empty lines. **You will lose 10 points** if it cannot process multiple command lines.
- When your software starts running, it shall display `"Vehicle Management System is running."`. Next, it will continuously read and process the command lines until the "Q" command is entered. If the Q command is entered, display `"Vehicle Management System is terminated."` Then, the software stops normally. You will **lose 2 points** for each violation.
- You must define a **run()** method with a while loop to continuously read the transactions until a "Q" command is entered. Each iteration of the loop processes a command line. You will **lose 5 points** if the **run()** method is missing. You MUST keep this method **under 40 lines** for readability, or you will **lose 3 points**. You can define necessary instance variables and private helper methods.

(l) **RunProject1 class.** This is a driver class that runs your software. The graders will run this class to grade your project.

```
public class RunProject1 {
    public static void main(String[] args) {
        new Frontend().run(); //run the software
    }
}
```

11. **Test Specification.** Design test cases to test your code. Organize your test cases by using the table template from the Coding Standard, **or** you will **get 0 points for this part.** Use a document editor, such as Microsoft Word or Google Docs. Do not copy and paste your code into this document, or you will **get 0 points for this part**. Include this document in your project folder. **This part is worth 17 points total**.
    (a) **Date class** – design **four** invalid and **two** valid test cases for testing the **isValid()** method; you will **lose 12 points** if this is not done. You **lose 2 points** for each test case missing.
    (b) **Vehicle class** – design test cases to test the **compareTo()** method; **one** test case returns -1, **one** test case returns 1, and **one** test case returns 0; you will **lose 5 points** if this is not done. You **lose 1.5 points** for each test case missing.
12. **Unit Testing** – every Java class can include a **main()** method as the driver to test the public methods within the class. This is to unit test the Java class in isolation and ensure the API functions properly. The main() is also called "**testbed main()"**. You MUST implement the test cases designed in the Test Specification above. **This part is worth 18 points total.**
    (a) **Date class** – 6 test cases, **lose 2 points** for each test case missing; you **lose 12 points** if the main() is missing.
    (b) **Vehicle class** – 3 test cases, **lose 2 points** for each test case missing; you **lose 6 points** if the main() is missing.
13. You must generate the Javadoc after you properly comment your code. For grading purposes, set the scope to 'private" so your Javadoc will include the documentation for the private instance variables, constructors, private helper methods, and public methods.

    **Generate the Javadoc** in a single folder and include it in the zip file to be submitted to Canvas. You are responsible for double-checking your Javadoc folder after you generate it. Submitting an empty folder will **result in 0 points for this part**. Open the Javadoc folder and look for the **index.html** file. Double-click the file and check every link to ensure all comments are NOT EMPTY. You will lose points if any description in the Javadoc is empty. You will **lose 5 points** for not including the Javadoc.