Course: COSC2081 - Programming 1 - Semester 2020B
Group: Weapon Masters
Members:

1. Truong Duc Khai: s3818074
2. Huynh Vinh Tin: s3805632

Table of contents

I. Introduction

This app is a CRM system that will help increase the efficiency and the clarity of the business through letting the user see a more general view of the data that they collected, letting them have a better understanding about their leads and the interactions between the sales people and leads. The app includes features such as CRUD (create, read, update, delete) on leads and interactions, report and statistics on the data collected.

II. Tools and technologies

1. IntelliJ IDEA Community

Intellij IDEA Community is a specialized IDE for Java, it provides a lot of code snippets and code suggestions that can help us to develop the system faster and make our code more clean, preventing syntax errors and redundant code.

2. GitHub

Used to create a space for us to share and update our project, keeping track of each other's progress, preventing situations like we are both working on the same aspect which is a waste of time and effort.

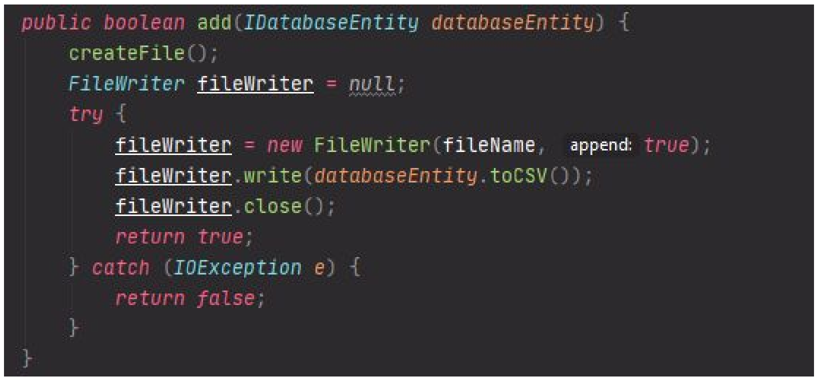
III. Steps

1. Building a database system and entity classes.

From the assignment instructions, we can figure out that there are 2 main entities in this system: Lead and Interaction. The CSV file will act as a database to store records of these 2 entities.

Class	**Functionality**
Lead	Entity class. Provide all the needed fields to be inserted into a CSV file. Can convert its instance into CSV string and vice versa.
Interaction	
Database	Act as a CSV reader, which can perform create, read, update and delete (CRUD) operations on CSV files. Constructor takes in the CSV file name it will operate on.

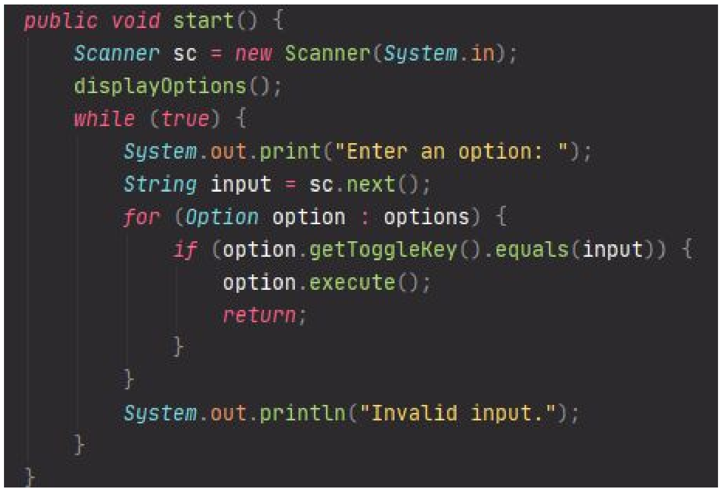
We also create an interface called IDatabaseEntity with the method toCSV(), which will be implemented by Lead and Interaction classes, forcing them to specify how their data should be written in CSV format, because we do not want the database to be used by non-entity classes.



```java
public boolean add(IDatabaseEntity databaseEntity) {
    createFile();
    FileWriter fileWriter = null;
    try {
        fileWriter = new FileWriter(fileName,  append: true);
        fileWriter.write(databaseEntity.toCSV());
        fileWriter.close();
        return true;
    } catch (IOException e) {
        return false;
    }
}
```

2. Building menu navigation.

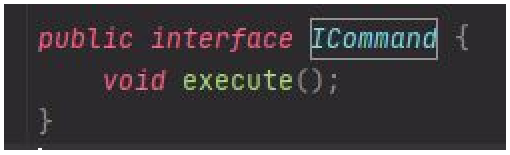
At first, we used nested switch case to handle different operations based on user input. Later, we find out that switch-case makes our code hard to read. We find out that we can encapsulate the logic of an option menu, which is displaying all the options for the user to select, then ask for the user input until it is valid, into a class called OptionMenu.
An option menu usually has a list of options for user to select, which is why we also decide to create a class called Option to represent the properties of an menu option: label, toggle key and a command, which is a callback function that is executed when the user types in an input that matches the toggle key.



```java
public void start() {
    Scanner sc = new Scanner(System.in);
    displayOptions();
    while (true) {
        System.out.print("Enter an option: ");
        String input = sc.next();
        for (Option option : options) {
            if (option.getToggleKey().equals(input)) {
                option.execute();
                return;
            }
        }
        System.out.println("Invalid input.");
    }
}
```

Implementation of OptionMenu main method start(), which displays all options, asks for user input and then executes the function.

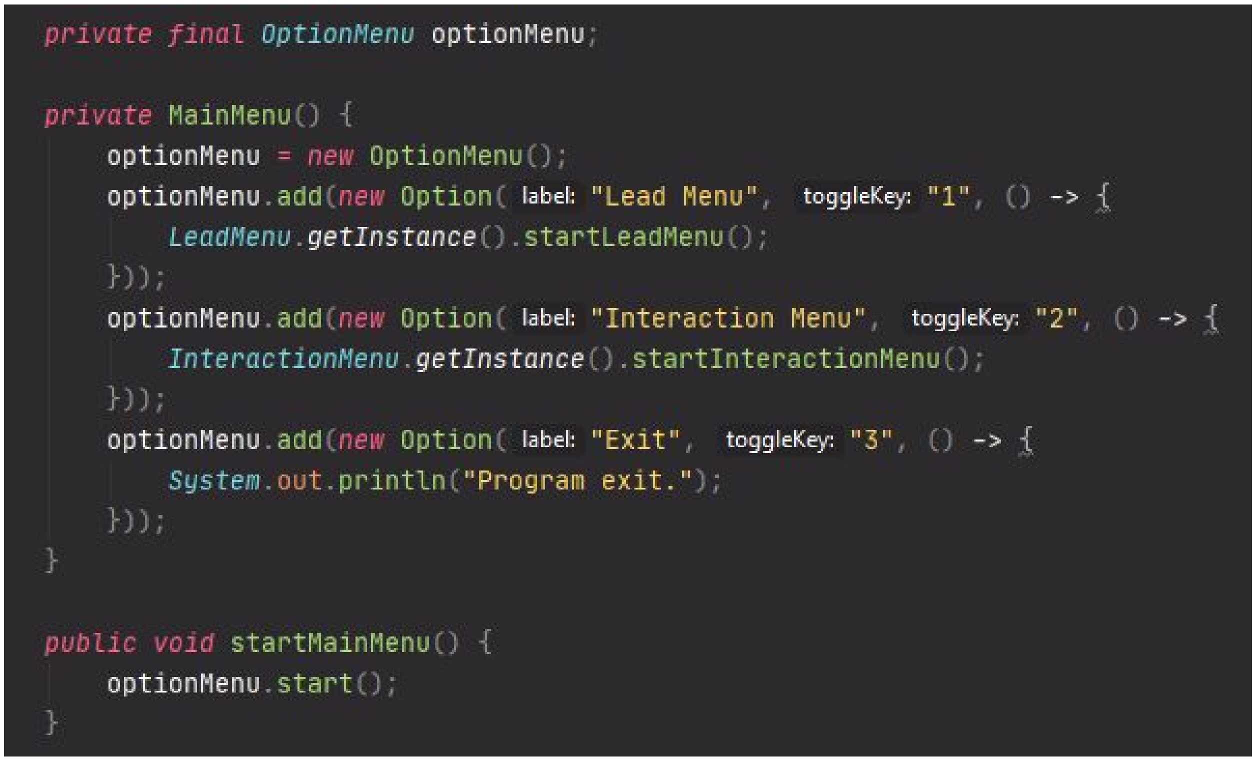
To create a callback function, we create an interface called ICommand, then make it a parameter in the Option constructor.



```java
public interface ICommand {
    void execute();
}
```

Because this interface only has 1 function, the IDE will recognize it as a functional interface and suggest to change the argument into a lambda function.

We also create 3 classes: MainMenu, LeadMenu, InteractionMenu which will hold their respective user interface and related functions. We also make all of them singletons because each menu only needs 1 instance.

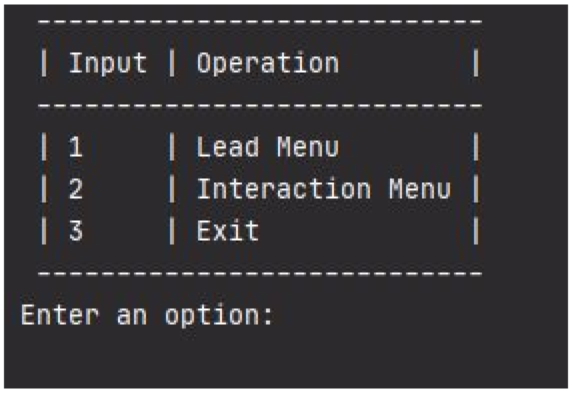


```java
private final OptionMenu optionMenu;

private MainMenu() {
    optionMenu = new OptionMenu();
    optionMenu.add(new Option( label: "Lead Menu",  toggleKey: "1", () -> {
        LeadMenu.getInstance().startLeadMenu();
    }));
    optionMenu.add(new Option( label: "Interaction Menu",  toggleKey: "2", () -> {
        InteractionMenu.getInstance().startInteractionMenu();
    }));
    optionMenu.add(new Option( label: "Exit",  toggleKey: "3", () -> {
        System.out.println("Program exit.");
    }));
}

public void startMainMenu() {
    optionMenu.start();
}
```

Implementation of OptionMenu inside MainMenu



```
 ----------------------------
 | Input | Operation        |
 ----------------------------
 | 1     | Lead Menu        |
 | 2     | Interaction Menu |
 | 3     | Exit             |
 ----------------------------
Enter an option:
```

Here is the interface that is displayed when the option menu is started.

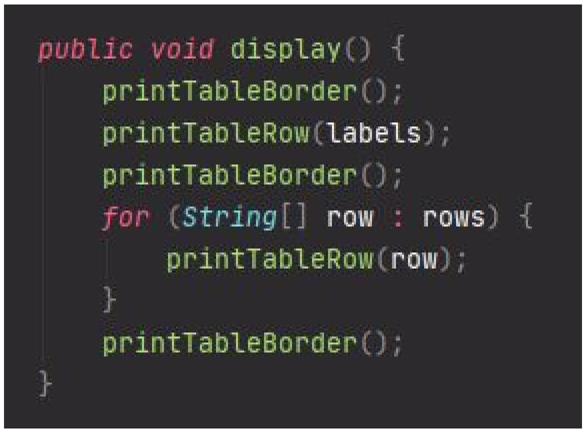
Selecting LeadMenu will navigate into a menu with functionalities that are related to lead. There is also a Back option inside that menu to navigate back to MainMenu. After finishing a task inside LeadMenu, the system will wait for the user to press Enter before navigating back to LeadMenu again. The same could be said for InteractionMenu.
Selecting Exit will just exit the menu loop and stop the program.

3. Implementing CRUD functionalities.

a) View all

Retrieve all rows inside the CSV files through Database class.

Each entity class will provide a static method called fromCSV(), which will create an instance of that class from a CSV string, which is a row in the CSV file.
We want the data to be printed in a table format. At first, we put all the implementation details of table formatting inside the menu, which makes the code not readable and violates the single-responsibility principle. For that reason, we want to abstract the process of table formatting into a class called TableFormatter.



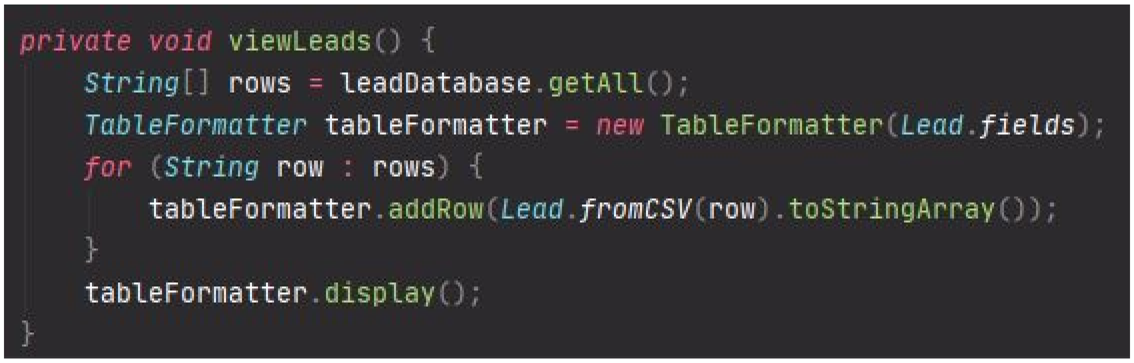
```java
public void display() {
    printTableBorder();
    printTableRow(labels);
    printTableBorder();
    for (String[] row : rows) {
        printTableRow(row);
    }
    printTableBorder();
}
```

Main method of TableFormatter, which prints 2-dimensional data in a table format.

The TableFormatter class has 2 main properties: labels and rows.

- labels: names of all columns, which is an array of strings provided by entity class.
- rows: an ArrayList of arrays of strings, which contains the data to display. Each entity class will also provide a static method called toStringArray() to convert its instance into a string array that can be added into the rows ArrayList.

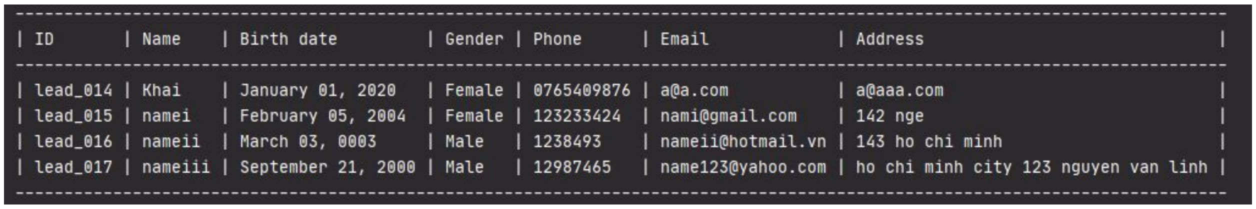
The client of TableFormatter only needs to care about data insertion because TableFormatter has already hidden all implementation details of formatting a table and calculating the column length to fit the data.



```java
private void viewLeads() {
    String[] rows = leadDatabase.getAll();
    TableFormatter tableFormatter = new TableFormatter(Lead.fields);
    for (String row : rows) {
        tableFormatter.addRow(Lead.fromCSV(row).toStringArray());
    }
    tableFormatter.display();
}
```

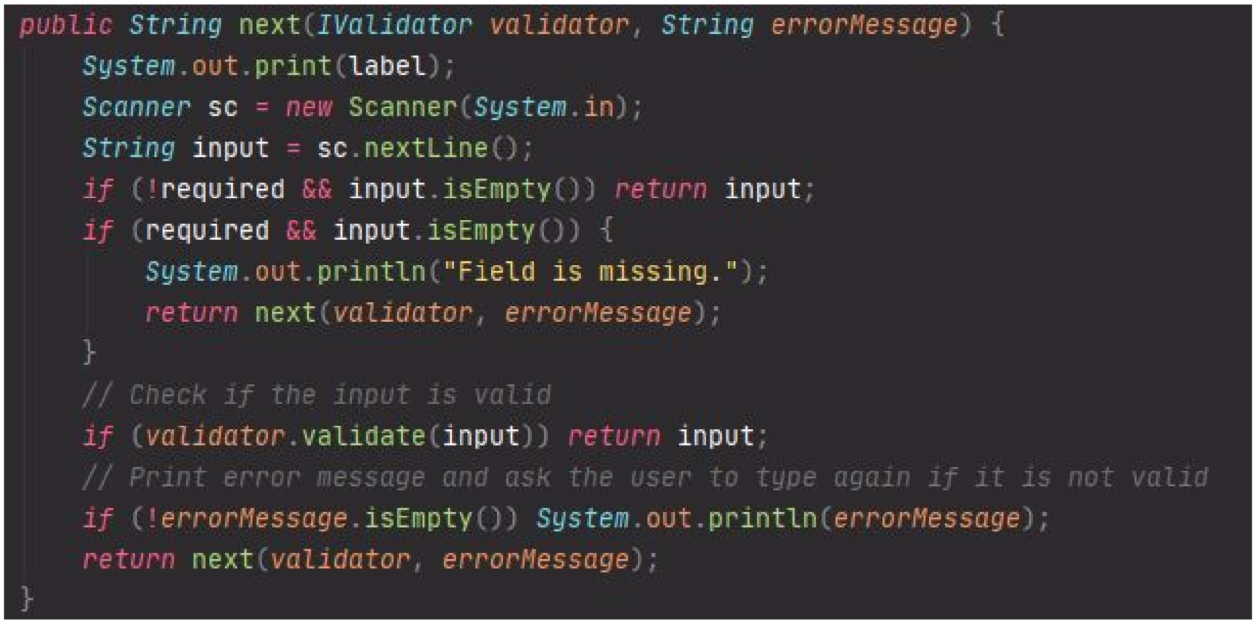
Implementation of the option to view all leads using TableFormatter.

First, we instantiate an instance of TableFormatter with all the column names. Then we add every row in the CSV files into the table formatter and finally call the method display().


```
-------------------------------------------------------------------------------------------------------------------------------------
| ID       | Name    | Birth date         | Gender | Phone      | Email              | Address                                    |
-------------------------------------------------------------------------------------------------------------------------------------
| lead_014 | Khai    | January 01, 2020   | Female | 0765409876 | a@a.com            | a@aaa.com                                  |
| lead_015 | namei   | February 05, 2004  | Female | 123233424  | nami@gmail.com     | 142 nge                                    |
| lead_016 | nameii  | March 03, 0003     | Male   | 1238493    | nameii@hotmail.vn  | 143 ho chi minh                            |
| lead_017 | nameiii | September 21, 2000 | Male   | 12987465   | name123@yahoo.com  | ho chi minh city 123 nguyen van linh       |
-------------------------------------------------------------------------------------------------------------------------------------
```

b) Add

- Retrieve the last ID in the CSV files, with this last ID, we can use it to create the next ID for the entity that the user is about to input.
- Receive input for all the fields: at first we validate the input manually using while loop, however, we found out that we can encapsulate the logic of asking the user to type input until it is correctly validated.
- We create a class called InputField, which has properties such as label (the content of input), required (allow user to skip a field in update method) and a String function next() to receive input with parameters: validator (a lambda function that takes in a string and return a boolean value for its validity), errorMessage (the message to print when the input is not valid).


```java
public String next(IValidator validator, String errorMessage) {
    System.out.print(label);
    Scanner sc = new Scanner(System.in);
    String input = sc.nextLine();
    if (!required && input.isEmpty()) return input;
    if (required && input.isEmpty()) {
        System.out.println("Field is missing.");
        return next(validator, errorMessage);
    }
    // Check if the input is valid
    if (validator.validate(input)) return input;
    // Print error message and ask the user to type again if it is not valid
    if (!errorMessage.isEmpty()) System.out.println(errorMessage);
    return next(validator, errorMessage);
}
```

- We ask for all input fields using the next() method of InputField. Then we create an object by passing all of those inputs into its constructor.
- By the toCSV() method that is implemented by the entity class, we call the method add() of Database and pass in the object to save its data into the CSV file.

c) Delete

- We prompt the user for the ID that they want to delete. The program will then read and rewrite all the rows except for the row with that ID . The user can cancel the prompt by entering nothing.
- Cascade delete interactions when deleting leads: If the lead is not involved in any interaction, the deletion will be made automatically, otherwise it will show the warning with the number of interactions that lead is involved in, then ask the user to type y/n (yes/no) to confirm deletion.

d) Update

- The update function is very similar to the delete function, but rather than skipping the line, we write that line with the new user inputs.
- To make updating easier, when the user types in the existing ID, it will show the current data of that entity. The input field in update option is also similar to the add option, but the user can leave the field blank to skip the field if they do not want to modify the data. However, if they type something, the input must be validated, which is similar to the add option.

4. Report and statistics

- The CRM system can report the number leads in group ages. Mainly 4 group ages: 0-10, 10-20, 20-60,60+ which stand for children, teenager, adult, senior.
- The app can also give a report about all the interaction's potentials in a period of time.
- Last but not least, the app can also give a sum of interactions in a period of time, it will display all the month that is within the start and end date, even if there is no interaction.

5. Testing

- Unit testing: create a package called 'test' to store all the classes to run the unit test, which includes classes such as Database (to check if data is read correctly), validators (to check if validating functions correctly validates user input).
- Manual testing: testing menu navigation, check if all options inside the menu achieve their purpose, testing the validator with different types of inputs.

IV. Work distribution

Truong Duc Khai	Huynh Vinh Tin
- Create database and entity classes. - Come up with the design for classes to optimize code reuse. - Create functions that will be implemented by Tin. - Report and statistics: view interactions by month and potential. - Write the report. - Present in the video.	- Implementing menu options: add, view update, delete. - Build menu navigation. - Create validator classes. - Writing tests for validators. - Report and statistics: view leads by age. - Write the report. - Present in the video.

V. Result

We manage to finish all the features required in the assignment instructions

+ Build an option menu for the user to choose a task.
+ Users can view, add, delete and update leads and interactions.
+ Warning and cascading deletion when deleting a lead.
+ Validate all the inputs that the user types in, minimizing errors for the system. Showing appropriate error messages for the user to correct the input.

+ View number of leads by group age.
+ View interactions in a period of time introduced by the user.
+ View interactions by potential.

We also have some features to improve the quality of the system and user experience:

+ Wait the user to press enter after a task is finished before showing the option menu immediately.
+ Split menus by functionalities: Lead Menu, Interaction Menu, Main Menu, with options to navigate to other menus.
+ Data and menu options are printed in table format, making the user interface cleaner.
+ Showing current information of an entity in the update option, allowing the user to skip fields that they do not want to change.

VI. Challenges

- As this is the first time we program in pairs and work on the same project, conflict when pushing the code in GitHub repo is inevitable. Therefore, we have to create branches for each person to work on different features before pushing to master branch. To avoid working on the same functionality, which can waste our time and cause conflicts, we have to agree on a workflow and communicate with each other frequently to catch up on how others are doing.
- This is also the first time we build a fully functional console application in Java, which involves a different programming paradigm compared to projects we have done in the past such as game, web, etc. Therefore, we have to come up with a good OOP design for the system to make the code reusable and maintainable. We have to encapsulate what remains the same in the system and differentiate them from what varies.
- As our expectation for this project is very high, we would not only write code that can only run, but also readable and maintainable. We also want to make the system feel nice to use as a user, which takes us a lot of time to discuss the correct way to implement a feature. This is what makes us learn a lot when doing this project.

VII. Demo video

https://youtu.be/2fsN3iOnHDw