



Object Oriented Programming Final Project

Lecturer: Jude Joseph Lamug Martinez MCS

Class L2AC

Student ID: 2702255981

Student Name: Rafael Anderson

Introduction

1. Background

In today's world, almost everything is involved with digital systems. From ordering foods in a restaurant to booking travel accommodations, digital systems have become very involved, aiming to enhance user experience, making processes more efficient and user-friendly. The usage of digital systems is crucial to help businesses remain competitive and meet customer's expectations.

Car rental businesses in particular, can benefit by implementing these digital systems. Usually, customers are required to meet employees face to face, do paperworks, check if the cars are available, which may lead to an inefficient process with errors.

2. Solution

The solution to solve these inefficiencies and error prone processes is to implement a digital system. I came up with a digital system, allowing both employees and customers to use it. Where users are able to sign in or login, making it easier for them to access the program if they have logged in.

This digital system will be extremely useful for the businesses, benefiting both the business and the customers, making all the processes significantly faster than before. For this project, this program will be a car rental management system for a car rental shop called FleetFlex.

Libraries Used

1. Java Swing

Java Swing is used to create the graphical user interface for the program. It provides a user-friendly design, allowing users to navigate through the program easily.

2. Java Util

Java Util is used to store all of the necessary data needed for the program to function. Where arraylist is used to store cars, employees, and customers. While the hash tables are used to store a key value pair of user, employee and user, customer.

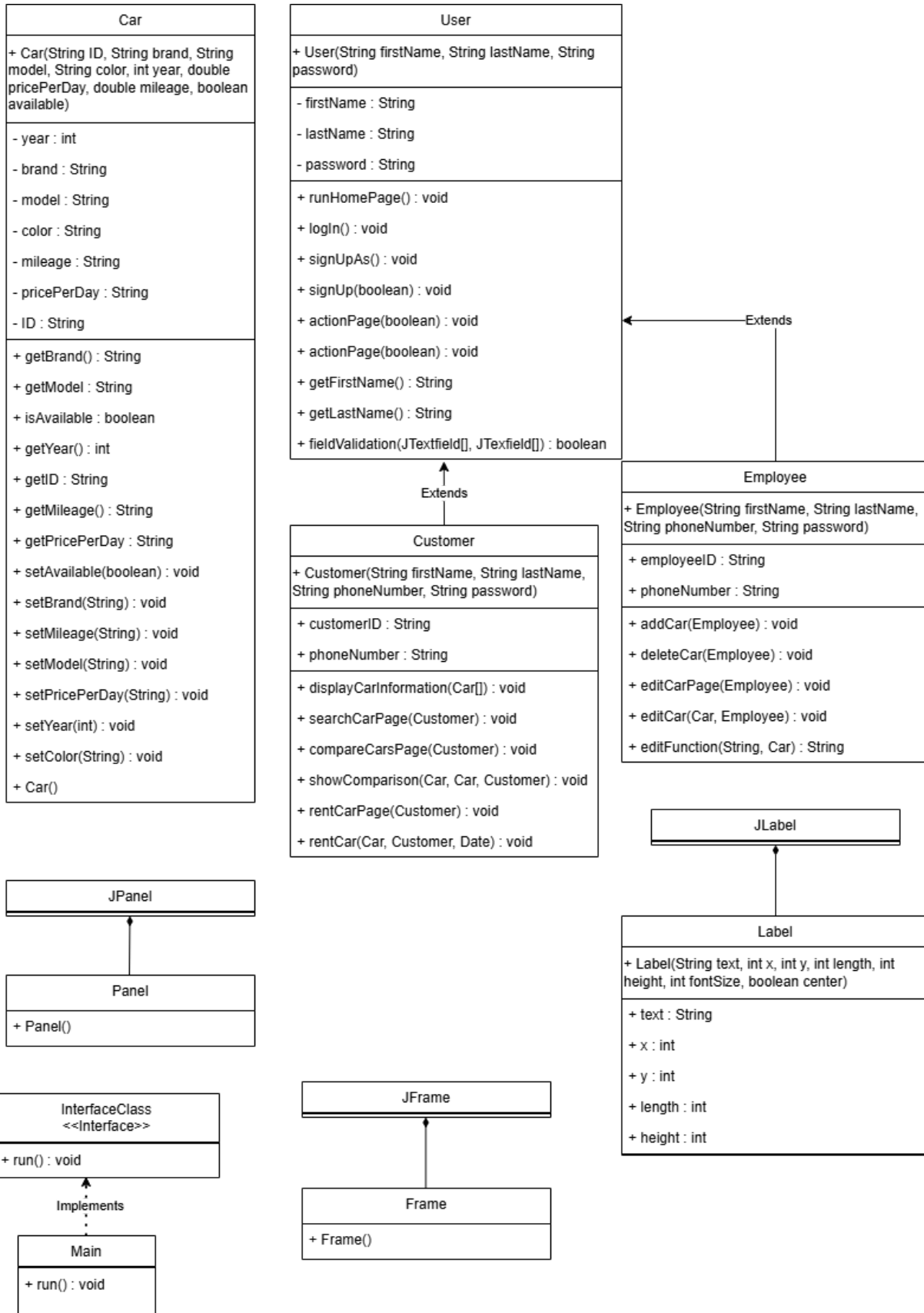
In addition to that, java util random is also used to generate a random integer, that is used in the generation of the car ID.

3. Java awt

In this program, java awt is used mainly in designs. For example, it is used to create a color for the background, set a new font, etc.

Solution Design

1. UML Diagram



Algorithms

1. **Field Validation:** It checks whether all of the fields are filled in pages that require the users to enter data into the text fields. It is also used to check if the data entered in the password and confirm password fields are equivalent or not.

```
public static boolean fieldValidation(JTextField[] textFields, JTextField[] numberFields){  
  
    boolean reminder = false;  
  
    // Makes sure all fields are filled  
    for (JTextField textField : textFields){  
        if (textField.getText().isEmpty() && !reminder) {  
            reminder = true;  
            JOptionPane.showMessageDialog( parentComponent: null, message: "Please fill in all of the fields");  
            break;  
        }  
    }  
  
    // Checks whether the values are in integer for fields that require integer inputs, may be null because some validation do  
    if (numberFields != null) {  
        for (JTextField numberField : numberFields) {  
            try {  
                int value = Integer.parseInt(numberField.getText());  
            } catch (NumberFormatException a) {  
                if (!reminder) JOptionPane.showMessageDialog( parentComponent: null, message: "Please enter valid values");  
                reminder = true;  
                break;  
            }  
        }  
    }  
}
```

For field validations, there is a method that handles it by taking 2 text fields as its parameters, a textfield and a number field. The first text field is used to ensure that all of the text fields are already filled. While the number field is used to ensure that the data entered in that field is of the integer data type. It is checked by using try and catch, by converting the string into an integer. If there is an error in the parsing process, the reminder value will be set to true. Lastly it returns the reminder value. Where a true reminder value represents a warning / an error, and a false reminder value represents no errors.

```

char[] phoneNumberArray = phoneNumberTextField.getText().toCharArray();

// Checks whether the phone number entered is in the correct format
for (char c : phoneNumberArray){

    // Checks whether each position in the string is a digit
    if (!Character.isDigit(c)){
        reminder = true;
        JOptionPane.showMessageDialog(parentComponent: null, message: "Please enter a valid phone number");
    }
}

// Ensure that the length of the phone number is valid
if (phoneNumberArray.length < 9){
    reminder = true;
    JOptionPane.showMessageDialog(parentComponent: null, message: "Please enter a valid phone number");
}

```

In addition to that, there is also a validation done towards the phone number text field by converting the string to a character array, where it checks if each position in the string entered is a digit by going through the character array of the string and the length of the phone number. If the inputs are invalid, the reminder will be set to true.

2. **User Authentication / Login:** The user authentication checks whether the user is valid or not. For example, when the user has successfully signed up and wants to log in, it checks if the account is already registered or not. It checks by comparing the first name, last name, and the password entered.

```

Employee employee1 = new Employee(firstNameString, lastNameString, phoneNumberString, passwordString);
User user = new User(firstNameString, lastNameString, passwordString);
Main.employeeList.add(employee1); // Add to the employee list
Main.employeeHashtable.put(user, employee1); // Add to the employee hashtable used in login process

```

```

Customer customer = new Customer(firstNameString, lastNameString, phoneNumberString, passwordString); // Creates new Customer
User user = new User(firstNameString, lastNameString, passwordString); // User
Main.customerList.add(customer); // Add to the customer list
Main.customerHashtable.put(user, customer); // Add to the customer hashtable

```

When a user signs up as an employee or customer, a user object is created along with the employee or customer objects. A hash table is used to store the user object as the key and the customer or employee object as the value.

```
// Goes through the users of customers
for (User users : Main.customerHashtable.keySet()) {

    // Checks if the user is equals to any of the registered user
    if (user.firstName.equals(users.firstName) && user.lastName.equals(users.lastName) && user.password.equals(users.password)) {

        userFound = true;
        Customer customer = Main.customerHashtable.get(users); // Get the customer value from the user key
        actionPage(customer); // Goes to the customer action page
        frame.dispose();
    }
}
```

```
// Goes through the users of employees
for (User users : Main.employeeHashtable.keySet()) {

    // Checks if the user is equals to any of the registered user
    if (user.firstName.equals(users.firstName) && user.lastName.equals(users.lastName) && user.password.equals(users.password)) {

        userFound = true;
        Employee employee = Main.employeeHashtable.get(users); // Get the employee value from the user key
        actionPage(employee); // Goes to the employee action page
        frame.dispose();
    }
}

// Displays error message if user is not found
if (!userFound) JOptionPane.showMessageDialog( parentComponent: null, message: "User invalid, please check for misspellings.");
```

It goes through the user key in the hash tables, it compares the user properties between the the user object created from the data entered in the textfields of the login page. If they are equal, it will redirect the user to the action page either as a customer or employee. But if the user is not found, the system will display an error message.

3. **Adding Car:** Employees are prompted to enter the necessary information regarding a car like brand, model, etc. After all of the necessary information is obtained, the car will be added to the arraylist containing the list of cars.

```
// Get attributes
String brand = carBrandTextField.getText().substring(0,1).toUpperCase() + carBrandTextField.getText().substring(beginIndex: 1).toLowerCase();
String model = carModelTextField.getText().substring(0,1).toUpperCase() + carModelTextField.getText().substring(beginIndex: 1).toLowerCase();
String colour = colorTextField.getText().substring(0,1).toUpperCase() + colorTextField.getText().substring(beginIndex: 1).toLowerCase();
int yearValue = Integer.parseInt(yearTextField.getText());
double pricePerDay = Double.parseDouble(priceTextField.getText());
double mileageValue = Double.parseDouble(mileageTextField.getText());

// Create new car and add it to the car list
Car car = new Car(randomID, brand, model, colour, yearValue, pricePerDay, mileageValue, available: true);
Main.carList.add(car);

// Show a message stating that the car is added successfully
JOptionPane.showMessageDialog(parentComponent: null, message: "Car " + brand + " " + model + " has been successfully added!");
```

Based on the necessary textfields the user needs to enter regarding the car, the system will retrieve all of the data entered in each text field once the add button is clicked, and create a new car object using the data, then adding it to the list of cars. This is done by using an action listener on the button.

4. **Delete Car:** Employees are given a combo box, where they choose a car of their choice. If the employee clicks the delete button, the system will go through the arraylist containing the list of cars to find the targeted car and delete it.

```
// Creation of combo box
String[] carIDs = new String[Main.carList.size()];

int i = 0;
for (Car C : Main.carList){
    carIDs[i] = C.getID() + "/" + C.getBrand() + " " + C.getModel();
    i++;
}

// Combo box
JComboBox carChoice = new JComboBox(carIDs);
carChoice.setBounds(x: 190, y: 160, width: 150, height: 30);
```


For the deletion of cars, the system utilizes a combo box that allows the user to choose a car of their choice to delete.

```
// Get selected car
String carName = String.valueOf(carChoice.getSelectedItem());

// Goes through the car list to see if it exists
for (Car C : Main.carList){

    // If yes, will ask for confirmation
    if ((C.getID() + "/" + C.getBrand() + " " + C.getModel()).equalsIgnoreCase(carName)) {
        int confirm = JOptionPane.showConfirmDialog( parentComponent: null, message: "Confirm Delete Car " + C.getBrand()

        // If user confirms, will remove the car and displays a message
        if (confirm == 0){
            Main.carList.remove(C);
            JOptionPane.showMessageDialog( parentComponent: null, message: "Car " + C.getBrand() + " " + C.getModel() + "
            deleteCar(employee);
            frame.dispose();

        // Otherwise, will cancel the removal
        } else JOptionPane.showMessageDialog( parentComponent: null, message: "Removal Canceled");
    }
}
```

By using an action listener on the button, the system will get the selected item once the delete button is clicked. Then, it goes through the list of cars and compares them one by one to find the car of choice. Once found, the system will ask the user for confirmation first. If the user confirms, the car object will be removed from the list of cars, and a message will be displayed.

5. **Edit Car:** Employees need to choose a car first from the combo box. Then they need to choose a feature that they want to edit. After choosing a feature, they are prompted to enter an input that will change that specific feature of the car to the input the user entered.

```
// Brand Button
JButton brandButton = new JButton( text: "Brand");
brandButton.setBounds( x: 30, y: 150, width: 90, height: 50);

brandButton.addActionListener(e->{
    String brand = editFunction( feature: "Brand", car);
    if (brand != null) car.setBrand(brand);
});
```

In the edit page, there are all buttons representing the car feature. An example is the brand button, if clicked, it will run the editFunction with a parameter “Brand”, along with the car. The edit function will return a string that the user changed to, but if it is null, it means that there were no changes made.

```
// Get the feature string
String carFeature = switch (feature) {
    case "Brand" -> car.getBrand();
    case "Model" -> car.getModel();
    case "Year" -> String.valueOf(car.getYear());
    case "Price" -> car.getPricePerDay().substring( beginIndex: 1);
    case "Mileage" -> car.getMileage().substring(0, car.getMileage().length() - 3);
    case "Color" -> car.getColor();
    default -> null;
};
```

Based on the feature entered, the carFeature will represent that particular feature of the car.

```

boolean validInput = false;
String change = null;

while (!validInput) {
    // Get the new string
    change = JOptionPane.showInputDialog("Edit " + feature + " for Car " + car.getBrand() + " " + car.getModel() +
        "\n\nEnter a " + feature + ":");

    // If it is status, it will check whether the input is valid or not (Available / Unavailable)
    if (feature.equalsIgnoreCase( anotherString: "Status")) {
        if (change.equalsIgnoreCase( anotherString: "Available") || change.equalsIgnoreCase( anotherString: "Unavailable")){
            validInput = true;
        } else {
            JOptionPane.showMessageDialog( parentComponent: null, message: "Please enter Available or Unavailable");
        }
    }
}

```

There is a special case if the user wants to change the status feature of the car, where they are required to enter Available or unavailable, or it will raise an error otherwise.

```

// Ensure the features that require an integer input is in integer data type
else if (feature.equalsIgnoreCase( anotherString: "Year") || feature.equalsIgnoreCase( anotherString: "Price") || feature.equalsIgnoreCase( anotherString: "Mileage")) {

    // Converts string to integer, if there is an error, will display an error message
    try {
        int intValue = Integer.parseInt(change);
        validInput = true;
    } catch (NumberFormatException e){
        JOptionPane.showMessageDialog( parentComponent: null, message: "Please enter an integer");
    }
} else if (change == null) { // Makes sure to cancel properly
    validInput = true;
    proceedTheProgram = false; // Make it not execute the rest of the method
} else validInput = true; // Will not loop if it is not regarding the status

```

For other inputs that require an integer input such as year, price, mileage. It uses try and catch. It will catch whenever the intValue receives a parse result that is not an integer, requiring users to enter an integer value. If there is an error, it will display an error message. Finally, it will check whether the user clicks on the cancel button or not (change value = null), if it is, the proceedTheProgram will be set to false not executing the rest of the code in the method. For other inputs, it doesn't require any validation, making validInput true automatically.

```

if (proceedTheProgram) {
    // Ask for confirmation
    int confirmation = JOptionPane.showConfirmDialog(parentComponent: null, message: "Confirm change of " + feature + " from " + ca

    // If yes, it will return the string changed and display a success message
    if (confirmation == 0) {
        JOptionPane.showMessageDialog(parentComponent: null, message: "Car " + feature + " is successfully changed from " + carFeat
    } else change = null; // If not, it will return null leading to no changes made
    }
}

return change; // Returns change

```

The rest of the code is just confirming whether the user confirms the change or not. If not, it will return change as a null string, causing no changes made in the values.

6. **Show Cars:** Customers are able to show all of the cars the car rental shop offers, along with the complete information of the car. The algorithm takes in a list of cars that will be shown, and will display the information of cars present in the list.

```

public static void displayCarInformation(Car[] cars){

    UIManager.put("OptionPane.messageFont", new Font( name: "Monospaced", Font.BOLD, size: 15)); // Make all letters occupy equal sp

    // String builder is used as the information will be displayed using JOptionPane
    StringBuilder print = new StringBuilder("\nCar ID:      Car Brand:   Car Model:   Color:      Year:      Mileage:

    for (Car C : cars){

        String available;
        if (C.isAvailable()) available = "Available"; // So it will not display true or false
        else available = "Unavailable";

        print.append("\n");

        // List of features
        String[] lengths = new String[]{C.getID(), C.getBrand(), C.getModel(), C.getColor(), String.valueOf(C.getYear()), String.v

        // Goes through all the features and adds space depending on the length of the string to create a structured printing
        for (String s : lengths){
            print.append(s);
            print.append(" ".repeat( count: 14 - s.length()));
        }
    }
}

```

The system shows the information of all the cars available by executing the displayCarInformation and putting the list of all cars present in the system. The

method utilizes a string builder in which the car information is appended, separated with a conditional number of spaces depending on the length of the string, making the printing format structured. It uses a string builder because the result will be printed using JOptionPane.

The UIManager is used to make all of the elements / alphabets take the same amount of space, allowing structured printing.

7. **Search Cars:** Customers are able to search cars, by first choosing a feature they want to search by. After choosing a feature, they are prompted to enter a string that they want to search in that specific feature. After clicking the search button, all of the cars within the criteria will be displayed

```
brandButton.addActionListener( e -> {  
  
    boolean proceedExecution = true; // To provide appropriate action if user clicks the cancel button  
    boolean present = false;  
    String carBrand = JOptionPane.showInputDialog("Enter a Car Brand:"); // Takes in brand input  
    if (carBrand == null) {  
        proceedExecution = false;  
    } // Redirects user if user clicks the cancel button  
    ArrayList<Car> targetCars = new ArrayList<>();  
  
    if (proceedExecution){  
        // Compares the brand of cars with the input, if it's the same, add car to the targetCars list and set present to true  
        for (Car C : Main.carList) {  
            if (C.getBrand().equalsIgnoreCase(carBrand)) {  
                targetCars.add(C);  
                present = true;  
            }  
        }  
    }  
}
```

For example, the user chooses to search cars by brand. If the user clicks the cancel button, the system will not execute the rest of the program (displaying the cars). The system will take an input from the user, and go through the list of cars to find cars fitting the criteria set by the user. Cars that fit the criteria will be added to the targetCar list, which will be used as a parameter in the displayCarInformation method. The present is set to true, so it will display an appropriate message if no cars fit the criteria set by the user.

```
// Display carInformation of the target cars
if (present) displayCarInformation(targetCars.toArray(new Car[0]));

// If none is found, display error message
else JOptionPane.showMessageDialog( parentComponent: null, message: "There are no Cars with the brand " + carBrand);
```

If cars that fit the criteria set by the user are present, the system will execute the `displayCarInformation` by using the `targetCar` list as the parameters. But if none of the cars fit the criteria set by the user, the system will display an error message.

8. **Compare Cars:** Customers are able to compare one car with another. By choosing both cars on a combo box, a comparison table will be displayed using `JTable`, presenting the data of both cars side by side.

```
compareButton.addActionListener(e -> {

    Car carNo1 = null;
    Car carNo2 = null;

    // Get selected cars
    String carName1 = String.valueOf(carChoice1.getSelectedItem());
    String carName2 = String.valueOf(carChoice2.getSelectedItem());

    // Displays an error message if cars are equivalent
    if (carName1.equalsIgnoreCase(carName2)) JOptionPane.showMessageDialog( parentComponent: null, message: "Please Choose Diff
    else {
        // Get the cars
        for (Car C : Main.carList){
            if ((C.getID() + "/" + C.getBrand() + " " + C.getModel()).equalsIgnoreCase(carName1)) carNo1 = C;
            if ((C.getID() + "/" + C.getBrand() + " " + C.getModel()).equalsIgnoreCase(carName2)) carNo2 = C;
        }

        // Show the comparison
        showComparison(carNo1, carNo2, customer);
        frame.dispose();
    }
});
```

The compare car feature utilizes 2 combo boxes. When the compare button is clicked, it finds the 2 cars selected and compares them if they are equal or not. If yes, an error message will be displayed asking the user to choose 2 different cars. Otherwise, the `showComparison` method is executed, with the cars and the customer object as the parameters.

```
String[] columns = {"", "Car 1", "Car 2"}; // Column Header

// Data in each row
String[][] data = {
    {"Car ID", car1.getID(), car2.getID()},
    {"Car Brand", car1.getBrand(), car2.getBrand()},
    {"Car Model", car1.getModel(), car2.getModel()},
    {"Color", car1.getColor(), car2.getColor()},
    {"Year", String.valueOf(car1.getYear()), String.valueOf(car2.getYear())},
    {"Price Per Day", car1.getPricePerDay(), car2.getPricePerDay()},
    {"Mileage", car1.getMileage(), car2.getMileage()}
};

// Create table and initializing its properties
JTable table = new JTable(data, columns);
table.setBounds(x: 396, y: 200, width: 470, height: 280);
table.setRowHeight(40);
table.setBorder(BorderFactory.createLineBorder(Color.BLACK, thickness: 3));
```

Based on the information of the 2 cars obtained, a JTable is initialized containing the information of both cars, setting the row height, size, and the coordinates. Where the table will be added to the frame later.

9. **Rent Cars:** Lastly, customers are able to rent cars, by choosing a car from the combo box. And entering the date and the amount of rental days. The system will check whether the car is available or not, if it's unavailable, the process will be canceled. The system will ask the customer for confirmation, if the user confirms, the car will be rented and the availability will be set to false.

```

for (Car C : Main.carList){

    // Car information, with space separating to make it structured
    Label carID = new Label(C.getID(), x: 100, y, length: 850, height: 100, fontSize: 15, center: false);
    Label carInformation = new Label(text: C.getBrand() + " " + C.getModel(), x: 250, y, length: 500, height: 100, fontSize: 15, center: false);
    JButton button = new JButton(text: "Rent"); button.setBounds(x: 420, y: y + 38, width: 80, height: 25); // Add a rent button for each car

    button.addActionListener( e-> {

        // If the car is currently unavailable, will display an error message
        if (!C.isAvailable()) JOptionPane.showMessageDialog( parentComponent: null, message: "The car " + C.getBrand() + " " + C.getModel() + " is currently unavailable");
    }
}

```

It goes through the list of cars, and prints the car information, along with a rent button. If the user clicks on it, it checks whether the car is available or not. If not, it will display an error message.

```

else {

    int days = 0;
    Date dateObtained = null;
    boolean invalidDays = false;

    // Get a valid number of rental days
    while (!invalidDays) {

        // Checks whether the data entered is an integer data type and greater than 0
        try {
            days = Integer.parseInt(JOptionPane.showInputDialog("Enter the amount of days the rental will be (Max 30 Days)"));
            if (days > 0 && days <= 30) invalidDays = true;
        } catch (NumberFormatException a) {
            JOptionPane.showMessageDialog( parentComponent: null, message: "Please Enter a valid Days Value");
        }
    }
}

```

If the car selected is available, the system will prompt the user to enter the number of days they would like to rent. It will continue to loop until the number of days is valid based on both the value and the data type, by using the try and catch to verify it is an integer data type. Which is done by parsing it into an integer, and will continue looping if there is a parsing error.

```

// Get Valid Date
boolean validDate = false;
SimpleDateFormat formatter = new SimpleDateFormat( pattern: "dd/MM/yyyy"); // Create Date format
formatter.setLenient(false); // Ensure values in the date entered are valid
Date now = Date.from(Instant.now()); // Get current time

```


A date formatter is initialized, and set to lenient to ensure that the values entered are valid. The date today is obtained, to compare and ensure that the date entered by the user is after today.

```
// Loops until date entered is valid
while (!validDate) {
    try {
        // Get date from user
        String dates = JOptionPane.showInputDialog("Enter a date for the rental in the form of (dd/MM/yyyy)");
        dateObtained = formatter.parse(dates); // Parse date

        // Ensures that the date is after today
        if (now.after(dateObtained)) {
            JOptionPane.showMessageDialog(parentComponent: null, message: "Please enter a date after today");
        } else validDate = true;
    } catch (ParseException a) { // If there is a parse error, will display an error message and continue loop
        JOptionPane.showMessageDialog(parentComponent: null, message: "Please enter a valid date");
    }
}

rentCar(C, customer, days, dateObtained); // If there are no more errors, car will be rented
frame.dispose();
```

While the date obtained is invalid, the system will continue to loop until it receives a valid date input that is after today with a correct format. It checks if there is any parse error when parsing the string using the formatter defined previously. After that validation, it checks whether the date entered by the user is after the date today. If all conditions are fulfilled, the rentCar method will be executed.

```

cancel.addActionListener( e-> {
    User.actionPage(customer);
    frame.dispose();
});

// Confirm button
JButton confirm = new JButton( text: "Confirm");
confirm.setBounds( x: 235, y: 330, width: 90, height: 30);

// Will display that the process was successful, and make the car.available to false
confirm.addActionListener( e-> {
    car.setAvailable(false);
    JOptionPane.showMessageDialog( parentComponent: null, message: "Car Rented Successfully!");
    User.actionPage(customer);
    frame.dispose();
});

```

Lastly, it asks the user for confirmation. If the user confirms, the car's availability will be set to false and the system redirects the user back to the actionPage. Else, the system will just redirect the user back to the action page without causing any changes.

Solution Scheme

Classes and Objects

1. **User:** Contains the common attributes of the customer and employee classes. It has some methods that the customer and employee objects can perform.
2. **Customer:** The customer class is a derived class from the user class with additional attributes such as customer ID and phone number. It contains all of the methods that customers are able to perform such as show, compare, search, and rent cars.

3. **Employee:** The employee class is another derived class from the user class. Also having extra attributes such as employee ID and phone number. It contains all of the methods employees can perform such as add, delete, and edit cars.
4. **Car:** The car class is a class to declare the cars stored in the system, containing attributes such as brand, model, ID, color, mileage, price, and availability. The class has basic setters and getters, allowing data to be displayed and edited.
5. **Panel:** The panel class is extended from JPanel, which makes it easier to initialize JPanels. By having this class, it doesn't require repetitive code lines.
6. **Frame:** The panel class is extended from JFrame, which makes it easier to initialize JFrames. By having this class, it doesn't require repetitive code lines.
7. **Label:** The panel class is extended from JLabel, which makes it easier to initialize JLabels. By having this class, it doesn't require repetitive code lines.
8. **InterfaceClass:** InterfaceClass is a self-made interface, in which the main class implements. It is only used to run the homePage().
9. **Main:** The Main class consists of data structures such as arraylist and hashtables. Which is used to store objects like User, Customer, Employee, and Car. It also runs the homePage when it is executed.

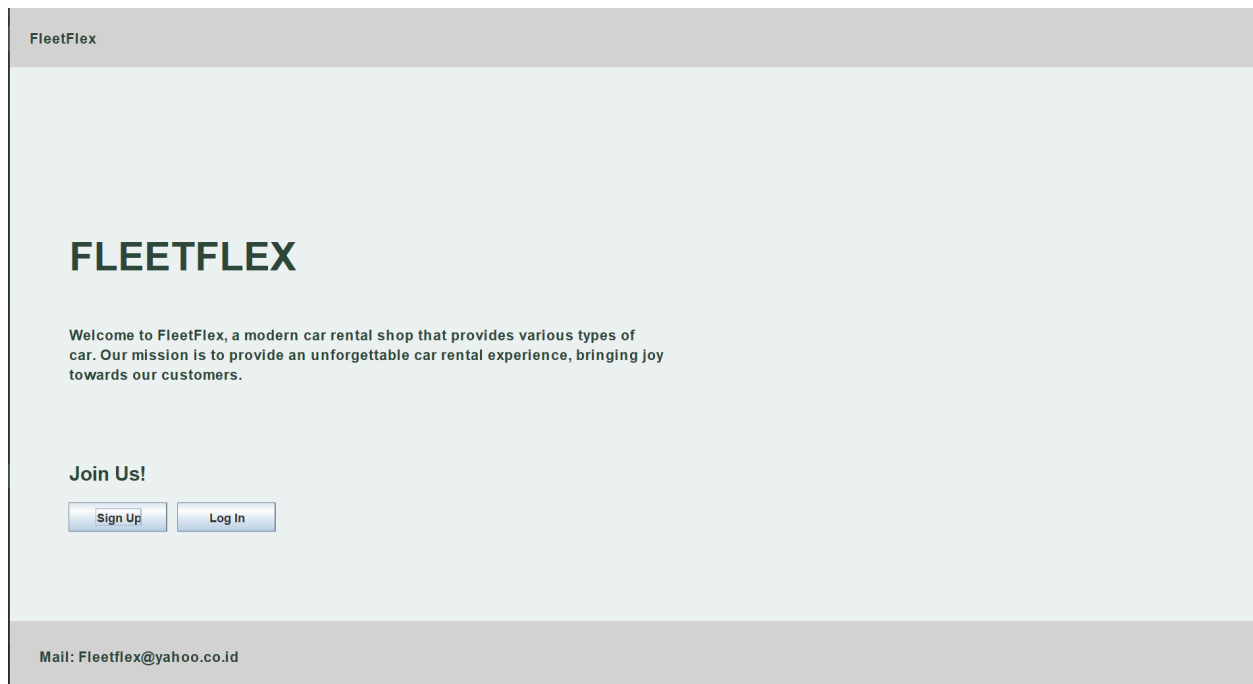
Data Structures Used

1. **Arraylist:** This program utilizes arraylists to store users, customers, employees, and cars. This allows users to go through the list and check if targeted objects are present or not.

- 2. Hash Tables:** Hash tables are mainly for the login process of users, it stores the User object as the key, and Customer or Employee object as the value. When users initially sign in as an employee or customer, a new User object is created with the first name, last name, and password inserted as parameters. Thus, when users login and enter their information, it checks for equivalent user objects in the hashtable keys. If it's equivalent, it gets the value which is either customer or employee, and runs the action page with it as the parameter.

Evidence of working program

Home Page:



Sign Up As:

Return

Sign Up As:

Employee

Customer

Return

Let's Sign Up!

First Name:

Rafael

Last Name:

Anderson

Phone Number:

081285293177

Password:

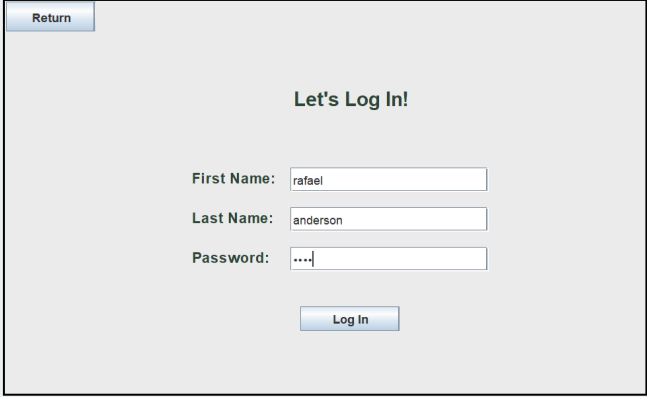
....

Confirm Password:

....

Sign Up

Login (After Signing Up)



A login form interface centered on a light gray background. The form is enclosed in a light gray box with a black border. In the top-left corner of the box is a blue button labeled "Return". The title "Let's Log In!" is centered at the top of the form area. Below the title are three input fields: "First Name:" with the value "rafael", "Last Name:" with the value "anderson", and "Password:" with the value "...". Each input field is a white rectangle with a thin gray border. Below the password field is a blue button labeled "Log In".

[Return](#)

Let's Log In!

First Name:

Last Name:

Password:

[Log In](#)

Employee HomePage

Welcome Rafael Anderson!
Employee ID: EMP0

Add Car

Delete Car

Edit Car

Show Cars

Customer HomePage

Welcome Rafael Anderson!
Customer ID: CUS0

Show All Cars

Search Cars

Compare Cars

Rent Car

Add Car

[Return](#)

Add Car

Car Brand:

Car Model:

Color:

Year:

Price/Hour:

Mileage:

[Add](#)

Message



Car Toyota Raze has been successfully added!

[OK](#)

Delete Car

Return

Delete Car

Car Choice

C11/Toyota Camry ▼

Delete

Select an Option ×

?

Confirm Delete Car Toyota Camry?

Yes

No

Cancel

Message ×

i

Car Toyota Camry is Removed!

OK

Edit Car

Return

Edit Car

Car Choice

C25/Nissan Altima ▼

Edit

Return

Edit Car's

Brand

Model

Year

Price

Mileage

Status

Color

Input

×

?

Edit Brand for Car Nissan Altima

Enter a Brand:

OK

Cancel

Select an Option

×

?

Confirm change of Brand from Nissan to Toyota?

Yes

No

Cancel

Show Cars

Message							
Car ID:	Car Brand:	Car Model:	Color:	Year:	Mileage:	Price/Day:	Status:
C42	Honda	Accord	Black	2020	240.0 KM	\$48.0	Available
C25	Toyota	Altima	Silver	2021	553.0 KM	\$50.0	Available
C11	Toyota	Corolla	Silver	2020	643.0 KM	\$40.0	Unavailable
C41	Toyota	RAV4	Blue	2021	250.0 KM	\$60.0	Available
C22	Honda	Civic	White	2019	100.0 KM	\$42.0	Available
C133	Honda	CRV	Black	2021	154.0 KM	\$58.0	Available
C24	Nissan	Sentra	Gray	2020	234.0 KM	\$38.0	Available
C45	Nissan	Rogue	Silver	2021	523.0 KM	\$55.0	Available
C216	Toyota	Highlander	Gray	2022	12.0 KM	\$70.0	Available
C54	Toyota	Raze	Black	2023	0.0 KM	\$40.0	Available

OK

Search Car

Return

Search Car By:

Brand**Year****Price****Mileage**

Input

? Enter a Car Brand:

Toyota

OK **Cancel**

Message							
	Car ID:	Car Brand:	Car Model:	Color:	Year:	Mileage:	Price/Day: Status:
	C25	Toyota	Altima	Silver	2021	553.0 KM	\$50.0 Available
	C11	Toyota	Corolla	Silver	2020	643.0 KM	\$40.0 Unavailable
	C41	Toyota	RAV4	Blue	2021	250.0 KM	\$60.0 Available
	C216	Toyota	Highlander	Gray	2022	12.0 KM	\$70.0 Available
	C54	Toyota	Raze	Black	2023	0.0 KM	\$40.0 Available
OK							

Compare Cars

Return

Compare Cars

Car 1:

C42/Honda Accord ▼

Car 2:

C54/Toyota Raze ▼


Compare

Comparison		
Car ID	C42	C54
Car Brand	Honda	Toyota
Car Model	Accord	Raze
Color	Black	Black
Year	2020	2023
Price Per Day	\$48.0	\$40.0
Mileage	240.0 KM	0.0 KM
Proceed		


Rent Car

Car ID:	Car Name:	
C42	Honda Accord	<input type="button" value="Rent"/>
C25	Toyota Altima	<input type="button" value="Rent"/>
C11	Toyota Corolla	<input type="button" value="Rent"/>
C41	Toyota RAV4	<input type="button" value="Rent"/>
C22	Honda Civic	<input type="button" value="Rent"/>
C133	Honda CRV	<input type="button" value="Rent"/>
C24	Nissan Sentra	<input type="button" value="Rent"/>
C45	Nissan Rogue	<input type="button" value="Rent"/>
C216	Toyota Highlander	<input type="button" value="Rent"/>
C62	Toyota Raze	<input type="button" value="Rent"/>

Input ×

 Enter the amount of days the rental will be (Max 30 Days):

Input ×

 Enter a date for the rental in the form of (dd/MM/yyyy)

Rent Information

Customer Name: Rafael Anderson

Car Name: Honda Accord

Date of Rent: Wed Dec 11 00:00:00 WIB...

Rent Period: 30 days

Price: \$1440.0

Cancel

Confirm

Message



Car Rented Successfully!

OK

Resources:

Github Link: [Rafael23082/OOP-FP: Object Oriented Programming Final Project \(github.com\)](https://github.com/Rafael23082/OOP-FP: Object Oriented Programming Final Project)