**Name: Mohamed Ahmed Mohamed Abdelhalim Attia**

**Project Report**

1. **Used Data Structures/Algorithms**

**Analysis of Authentication.java:**

**Data Structures:**

1. **ArrayLists**:
   * **PassengersList**, **ManagersList**, **DriversList**, **AvailableTripsList**, **VehiclesList**: These lists store different types of users, trips, and vehicles. They are used for managing entities in the system.
2. **HashMap**:
   * **userCredentials**: This map is used to store username-password pairs for authentication purposes.

**Algorithms:**

* **Login and Registration System**: Utilizes basic conditional checks and loops to handle user input and process authentication.
* **User Operations**:
  + Managers can review, add, or remove trips, assign drivers, and generate reports.
  + Drivers can view assigned trips and basic profile information.

These functionalities show an application of simple control flow mechanisms and data management using collections. No specific sorting or searching algorithms are mentioned in this file.

Now, let's examine the **Main.java** file to continue our analysis. ​​

**Analysis of Main.java:**

This file contains the main entry point for the program and is quite straightforward:

* **Function**: Initializes an instance of the **Authentication** class and calls the **LaunchSystem** method to start the application.

There are no specific data structures or algorithms used in this file; it simply serves to kick off the authentication process defined in **Authentication.java**.

Next, I'll review the **Trip.java** file to provide further insights into the data structures and algorithms used. ​​

**Analysis of Trip.java:**

**Data Structures:**

This class **Trip** acts as a container for information about a trip within the transportation system. It uses the following attributes to manage this information:

* **Attributes**:
  + **type**: Type of the trip (e.g., internal or external).
  + **Source**: Starting location of the trip.
  + **destination**: Ending location of the trip.
  + **isRoundTrip**: Boolean to indicate if the trip is a round trip.
  + **numberOfStops**: Integer to store the number of stops on the trip.
  + **availableSeats**: Integer to manage the number of seats available.
  + **price**: Double to represent the price of the trip.

**Algorithms:**

* **Method Implementations**:
  + **Getters and Setters**: Methods to get and set the properties of a trip.
  + **displayTripDetails()**: Method to print out the details of the trip, which includes type, source, destination, whether it's a round trip, number of stops, available seats, and price.

This class is primarily used for data encapsulation with getters and setters providing controlled access to the trip's data. No complex algorithms are used here, just straightforward data manipulation.

Next, let's examine the **User.java** file to continue identifying the data structures and algorithms used. ​​

**Analysis of User.java:**

**Data Structures:**

This file defines a hierarchy of user types in the transportation system, each represented by a class:

1. **User (Base Class)**:
   * Attributes: **username**, **password**, and **id**.
2. **Passenger (Derived from User)**:
   * Attributes: **ticketType**, **tripDetails**, and **numberOfBookedTickets**.
   * Methods include managing trip selections and ticket bookings.
3. **Driver and Manager (Derived from User, not fully displayed)**:
   * Expected to manage specific functionalities like trip assignment for drivers and operational management for managers.

**Algorithms:**

* **Inheritance**: Utilizes inheritance where **Passenger**, **Driver**, and **Manager** classes extend the **User** class.
* **Trip Selection**: Includes methods for selecting trips and handling user inputs with error checking.
* **Dynamic User Interaction**: Methods allow dynamic interaction based on user type, such as managing bookings for passengers or trip assignments for managers.

The classes here primarily manage user information and interactions, with methods structured to handle tasks specific to each user type. There are uses of loops and conditional statements to facilitate user interaction and data validation.

Finally, let's examine the **Vehicle.java** file to complete our analysis of the data structures and algorithms used. ​​

**Analysis of Vehicle.java:**

**Data Structures:**

The **Vehicle** class encapsulates data about vehicles used in the transportation system. Key components include:

* **Attributes**:
  + **type**: Enumerated type (**VehicleType**) defining possible vehicle types such as BUS, MINIBUS, and LIMOUSINE.
  + **capacity**: Integer representing the maximum number of passengers the vehicle can hold.
  + **licensePlate**: String for the vehicle's license plate number.

**Algorithms:**

* **Data Encapsulation**: Similar to the **Trip** class, this class uses getters and setters to encapsulate and manage vehicle data.
* **Information Display**: The **displayInfo()** method outputs details about the vehicle, including its type, capacity, and license plate.

**Summary of Data Structures and Algorithms Used:**

* **Data Structures**: Arrays, ArrayLists, Enums, and HashMaps are utilized to manage users, trips, vehicles, and authentication data.
* **Algorithms**:
  + **Control Structures**: Loops and conditionals for user interaction, data management, and error handling.
  + **Inheritance and Encapsulation**: Object-oriented principles are applied extensively to manage different types of data and operations, especially in user and vehicle management.
  + **Dynamic User Role Management**: Depending on whether the user is a passenger, driver, or manager, the system offers different functionalities, highlighting polymorphism and method overriding in practice.

1. **Any assumptions or Decisions**

**Assumptions:**

1. **User Input Validation**: The code might assume that users enter valid data. If extensive error handling isn't implemented, this could be an assumption that users will follow the instructions and inputs correctly without trying to break the system.
2. **Uniqueness of Usernames for all the users of the system**: when registering a new user or when the manager adds a new employee to the system, username entered should be unique.
3. **Uniqueness of Vehicle Plate number**: when the manager adds a new Vehicle to the system, plate number entered should be unique.
4. **Persistent Data Storage**: The use of a CSV file (**UsersDataBase.csv**) for storing user data suggests an assumption that the data size is manageable and does not require the use of more complex database systems.

**Decisions:**

1. **Data Structure Choices**: The decision to use ArrayLists and HashMaps for data management impacts performance, especially in search operations, which are O(n) for ArrayLists and O(1) for HashMaps.
2. **Error Handling**: Decisions regarding how to handle errors and exceptions, particularly input validation and file operations, affect the robustness and user experience of the system.
3. **File Storage for Persistence**: Choosing to use flat files (CSV) for data persistence over a database system is a significant decision, affecting data integrity, access speed, and scalability.
4. **A description about each method you are using and how does it work,**

**1. launchSystem() - Authentication.java**

* **Description**: This method serves as the entry point for the user interface of the transportation system. It loads user data, initializes available trips, and enters a loop to handle user actions such as register, login, and exit.
* **How it works**: The method begins by loading user data from a file and initializing lists of trips. It then uses a loop to continuously display the main menu and process user inputs through conditional checks, directing the flow to registration, login, or exiting the program.

**2. register(Scanner scanner) - Authentication.java**

* **Description**: Facilitates the registration of a new user by collecting username and password and storing them in the system.
* **How it works**: The method prompts the user to enter a username and password, checks for uniqueness, and if the credentials are not already taken, adds the new user to the appropriate list based on their role and updates the user credentials map.

**3. login(Scanner scanner) - Authentication.java**

* **Description**: Handles the login functionality, allowing users to access their specific profiles and functionalities based on their role.
* **How it works**: This method asks for username and password, validates them against stored credentials, and if successful, redirects the user to either passenger, driver, or manager functionalities depending on the user type.

**4. displayTripDetails() - Trip.java**

* **Description**: Prints detailed information about a trip to the console.
* **How it works**: The method accesses the trip’s attributes like type, source, destination, and others, and prints them in a formatted manner to provide clear information to the user.

**5. selectTrip(Scanner scanner) - Passenger.java**

* **Description**: Allows a passenger to select a trip from a list of available trips.
* **How it works**: It presents a list of trips and asks the user to choose one by entering the corresponding number. It performs validation on the input and displays the selected trip details using **displayTripDetails()**.

**6. assignDriver(Scanner scanner) - Manager.java**

* **Description**: Assigns a driver to a specific trip, used by managers.
* **How it works**: The method first displays all available trips and asks the manager to select one. After a trip is selected, it displays all available drivers and lets the manager assign a driver to the selected trip by updating the trip’s assigned driver attribute.

**7. displayInfo() - Vehicle.java**

* **Description**: Outputs the information of a vehicle.
* **How it works**: Retrieves the vehicle’s type, capacity, and license plate information and prints these details in a user-friendly format.

**8. generateReport() - Manager.java**

* **Description**: Generates a comprehensive report that includes details about vehicles, employees, and trips managed by a particular manager.
* **How it works**: This method aggregates data from various lists and prints out formatted information about each category, helping managers get an overview of operational aspects.

**9. addTrip(Scanner scanner) - Manager.java**

* **Description**: Allows managers to add new trips to the system.
* **How it works**: This method prompts the manager for details about a new trip, such as type, source, destination, round trip status, number of stops, available seats, and price. It then creates a new **Trip** object and adds it to the list of available trips.

**10. removeTrip(Scanner scanner) - Manager.java**

* **Description**: Enables managers to remove existing trips from the system.
* **How it works**: Managers are presented with a list of current trips. They select a trip to remove by entering its index, and the method validates the input and removes the trip from the list if the index is valid.

**11. addVehicle(Scanner scanner) - Manager.java**

* **Description**: Facilitates the addition of new vehicles to the fleet.
* **How it works**: The manager inputs the type, capacity, and license plate for the new vehicle. A new **Vehicle** object is created and added to the list of vehicles.

**12. addEmployee(Scanner scanner) - Manager.java**

* **Description**: Allows the manager to add new employees to the system.
* **How it works**: The manager enters the employee's details, including the type (driver or manager), name, and other relevant credentials. Depending on the type, a new **Driver** or **Manager** object is instantiated and added to the respective list.

**13. viewAssignedTrips() - Driver.java**

* **Description**: Displays the trips assigned to a specific driver.
* **How it works**: This method iterates through a list of trips and prints details of those that are assigned to the driver calling the method.

**14. accessBasicInfo() - User.java (and its subclasses)**

* **Description**: Outputs basic information about a user, such as ID and username.
* **How it works**: This method prints the basic identity attributes of any user, whether a passenger, driver, or manager, using the inherited properties from the **User** class.

**15. generateReport() - Manager.java**

* **Description**: Compiles and prints a detailed report of all system entities managed by the manager.
* **How it works**: The method collects and formats data about trips, vehicles, and other employees, providing a comprehensive overview for managerial review and decision-making.

**16. loadUserData() - Authentication.java**

* **Description**: Loads user data from a CSV file into the system at startup.
* **How it works**: This method reads the CSV file line by line, parsing user details and populating the respective lists (Passengers, Managers, Drivers) based on user types identified in the data.

**17. writeUserDataToFile() - Authentication.java**

* **Description**: Saves current user data back to the CSV file, typically called during system shutdown or after registration/login to ensure data persistence.
* **How it works**: It iterates over the user lists, formats the data into CSV format, and writes it to the file.

1. **State all test cases you tried while making your tests for each method.**

Here's a detailed list of test cases for the methods we discussed:

**1. launchSystem()**

* **Test Case**: Verify that the system prompts correctly for registration, login, and exit options.
* **Error Handling**: Check the system's response to invalid options.

**2. register(Scanner scanner)**

* **Normal Case**: Enter valid username and password for registration.
* **Edge Case**: Attempt registration with an already existing username.
* **Error Case**: Input empty or invalid data for username or password.

**3. login(Scanner scanner)**

* **Normal Case**: Enter correct username and password.
* **Edge Case**: Enter a username that does not exist.
* **Error Case**: Input incorrect password for an existing username.

**4. displayTripDetails()**

* **Normal Case**: Display details of a trip with all attributes filled.
* **Edge Case**: Display a trip with minimum or no bookings (testing the display of available seats).

**5. selectTrip(Scanner scanner)**

* **Normal Case**: Select a valid trip number from the list.
* **Edge Case**: Attempt to select a trip number that does not exist.
* **Error Case**: Enter non-numeric input when selecting a trip.

**6. assignDriver(Scanner scanner)**

* **Normal Case**: Assign a driver to a valid trip.
* **Edge Case**: Try to assign a driver to a trip that already has a driver.
* **Error Case**: Input an invalid trip number or driver number.

**7. displayInfo()**

* **Normal Case**: Display all vehicle details correctly.
* **Edge Case**: Display a newly added vehicle with default settings (if applicable).

**8. addTrip(Scanner scanner)**

* **Normal Case**: Add a trip with all required fields.
* **Edge Case**: Add a trip with the same details as an existing trip (if duplicates are not handled).
* **Error Case**: Leave required fields empty or input invalid data types (e.g., string for price).

**9. removeTrip(Scanner scanner)**

* **Normal Case**: Remove an existing trip by specifying its index.
* **Edge Case**: Attempt to remove a trip using an index that is out of range.
* **Error Case**: Enter a non-integer value for the trip index.

**10. addVehicle(Scanner scanner)**

* **Normal Case**: Correctly add a vehicle by specifying all required details.
* **Edge Case**: Add a vehicle with a license plate that already exists.
* **Error Case**: Input invalid data such as a non-integer for capacity.

**11. addEmployee(Scanner scanner)**

* **Normal Case**: Successfully add a new employee with valid details.
* **Edge Case**: Add an employee with a username that already exists.
* **Error Case**: Provide incomplete or incorrect data types for any field.

**12. viewAssignedTrips()**

* **Normal Case**: Display all trips assigned to a driver.
* **Edge Case**: Display the assigned trips for a new driver with no trips assigned yet.

**13. accessBasicInfo()**

* **Normal Case**: Display basic user info correctly.
* **Error Case**: Attempt to display info for a user that has been removed or not properly initialized.

**14. generateReport()**

* **Normal Case**: Generate a complete report covering all categories.
* **Edge Case**: Generate a report for a manager with no trips, vehicles, or employees under management.

**15. loadUserData()**

* **Normal Case**: Load all user data correctly on system startup.
* **Error Case**: Handle a scenario where the CSV file is corrupted or empty.

**16. writeUserDataToFile()**

* **Normal Case**: Ensure data is written correctly to the file.
* **Error Case**: Test response to file write errors, such as permissions issues.

These test cases aim to cover the functionality of your system comprehensively

1. **Screenshots for the program that shows every functionality, with different scenarios and test cases**

Screenshots hereeeeeeeeeeee

1. **A description about how you used the files to store and load the state**

**File Usage Overview:**

* **Primary File**: **UsersDataBase.csv**
* **Purpose**: To store user data, including credentials and roles, which allows for persistent user profiles across system runs.

**Storing State to File:**

**1. Data Formatting:**

* Each user's data is formatted as a CSV (Comma-Separated Values) string before being written to the file. Typical data fields stored include username, password, user type (Passenger, Driver, Manager), and any specific attributes relevant to their role.

**2. Writing Data:**

* **Method Used**: **writeUserDataToFile()**
* **Functionality**:
  + At system shutdown or after significant changes (like registration or role update), this method is called.
  + It iterates through all user lists (Passengers, Drivers, Managers).
  + For each user, their data is converted into a CSV format string.
  + These strings are written to **UsersDataBase.csv**, overwriting the previous content to update it with the current system state.

**3. Handling Complexity:**

* Simple file operations are employed, meaning the system is designed for scenarios with manageable user counts and data volumes that do not require the complexity of a database system.

**Loading State from File:**

**1. Reading Data:**

* **Method Used**: **loadUserData()**
* **Functionality**:
  + When the system starts, this method is invoked to initialize the user environment.
  + It reads **UsersDataBase.csv**, parsing each line to extract user data.
  + Based on the role specified in the data, it reconstructs the user objects (Passenger, Driver, Manager) and populates the respective lists in the system.

**2. Data Parsing:**

* Each line from the CSV file is split using commas as delimiters.
* The fields are used to create new user instances with attributes set according to the values from the file.

**3. Error Handling:**

* Basic error handling includes checks for file existence and readability.
* Additional checks might include validation of data format to ensure robustness against corrupted or malformed data entries.

**Benefits and Limitations:**

* **Benefits**:
  + Easy to implement and manage without needing complex database systems.
  + Sufficient for small to medium-sized user bases where performance is not a critical issue.
* **Limitations**:
  + Scalability issues with larger data sets, as every read or write operation requires processing the entire file.
  + Lack of advanced querying, indexing, and concurrent access management compared to database systems.

1. **A description about the kinds of exceptions you are handling in each part.**

Here's an overview of how exceptions are managed across different parts of the system:

**1. Authentication and User Management:**

**Login and Registration:**

* **Input Mismatch Exception**: During login and registration, there's potential for **InputMismatchException** if non-integer inputs are provided where integers are expected (e.g., choosing menu options). This is handled by catching the exception and prompting the user to try again with correct input.
* **File Not Found Exception**: When loading or saving user data to **UsersDataBase.csv**, **FileNotFoundException** might occur if the file doesn't exist or isn't accessible. This is handled by either creating the file if it's not found or informing the user about the issue.

**2. Trip and Vehicle Management:**

**Adding/Removing Trips and Vehicles:**

* **Array Index Out of Bounds Exception**: When accessing trips or vehicles from lists using indices provided by user input, there’s a risk of **IndexOutOfBoundsException**. This is managed by validating user inputs against the list sizes before accessing them.
* **Input Mismatch Exception**: Similar to user management, when expecting numeric inputs for things like setting prices or capacities and receiving incorrect formats, **InputMismatchException** is caught and handled appropriately.

**3. Data Loading and Saving:**

**File Operations:**

* **IOException**: General **IOExceptions** can occur during file read/write operations. These exceptions are caught and handled by notifying the user of the failure and potentially retrying the operation or exiting gracefully.
* **ParseException**: When parsing data from the CSV file, there might be formatting errors leading to **ParseExceptions**. These are handled by skipping the malformed lines or alerting the user about the data issues.

**4. General System Operations:**

**System Errors and Runtime Exceptions:**

* **NullPointerException**: This could occur if there are attempts to access or manipulate null objects, especially after loading data from files where not all required fields are guaranteed to be non-null. Such exceptions are handled by checking for null before use.
* **IllegalArgumentException**: This is handled when setting properties of objects with invalid arguments, such as negative numbers for prices or capacities, ensuring the system maintains logical integrity.

**5. User Interaction and Input Handling:**

**Scanner and Input Handling:**

* **NoSuchElementException**: This might occur if **Scanner** objects try to read input but encounter unexpected end of input (e.g., if a user abruptly disconnects or closes an input stream). It’s managed by ensuring all input operations are wrapped in try-catch blocks to handle unexpected input termination.

1. **State performance enhancements you did, and how did you achieve that. What are the complexities you saved after using it and comparing it with the previously used ones.**

**Changing userCredentials from ArrayList to HashMap:**

**Before Enhancement:**

* **Data Structure Used**: **ArrayList**
* **Complexity**: The time complexity for searching an **ArrayList** to authenticate a user (check username and password) was 𝑂(𝑛)*O*(*n*), where 𝑛*n* is the number of users. This was because each user's credentials needed to be checked sequentially until a match was found or the list was exhausted.

**After Enhancement:**

* **Data Structure Used**: **HashMap**
* **Complexity**: The time complexity for user authentication improved to 𝑂(1)*O*(1) on average, due to the direct key-value access provided by **HashMap**.
* **How Achieved**: Usernames were used as keys, and passwords as values in the **HashMap**. This allowed direct access to the password using the username as the key, eliminating the need for iterating through a list.

**Impact:**

* **Performance Gain**: This change significantly reduced the time required for user login, especially noticeable as the number of users grew. It enhanced the system's scalability and responsiveness.