**Halcion Motors Car Rental System Documentation**

This document provides a detailed overview of the Car Rental System, designed using Object-Oriented Programming (OOP) principles in Java. The system models key entities involved in a car rental business, including cars, customers, rental transactions, and the rental agency itself.

**1. System Overview**

The Halcion Motors Car Rental System is a console-based application that allows for the management of a car fleet, customer registration, and the processing of car rental and return transactions. It emphasizes modularity and maintainability through well-defined classes and encapsulation.

**Key Features:**

* **Car Management:** Add, remove, find, and list cars, including their availability status.
* **Customer Management:** Register, find, and list customers.
* **Rental Transactions:** Record car rentals, calculate rental costs, and process car returns.
* **Data Encapsulation:** Each entity's data and behaviour are self-contained within its respective class.

**2. Core Classes and Their Responsibilities**

The system is composed of four main classes: Car, Customer, RentalTransaction, and RentalAgency.

**2.1. Car Class**

**Purpose:** Represents an individual vehicle available for rent.

**Attributes:**

* make (String): The manufacturer of the car (e.g., "Toyota", "Honda").
* model (String): The specific model of the car (e.g., "Camry", "Civic").
* year (int): The manufacturing year.
* licensePlate (String): A unique identifier for the car. This is crucial for distinguishing vehicles.
* dailyRentalRate (double): The cost to rent the car for one day.
* isAvailable (boolean): A flag indicating whether the car is currently available (true) or rented (false).

**Methods:**

* **Constructor:** Car(String make, String model, int year, String licensePlate, double dailyRentalRate)
  + Initializes a new Car object. isAvailable is set to true by default upon creation.
* **Getters:** getMake(), getModel(), getYear(), getLicensePlate(), getDailyRentalRate(), isAvailable()
  + Provide read-only access to the car's attributes.
* **rentCar():**
  + Changes the isAvailable status from true to false if the car is currently available.
  + Returns true on success, false if the car is already rented.
* **returnCar():**
  + Changes the isAvailable status from false to true if the car is currently rented.
  + Returns true on success, false if the car is already available.
* **toString():**
  + Provides a string representation of the Car object for easy printing and debugging.

**2.2. Customer Class**

**Purpose:** Represents a client who rents cars from the agency.

**Attributes:**

* customerId (String): A unique identifier for the customer.
* name (String): The full name of the customer.
* contactInfo (String): Contact details, such as an email address or phone number.

**Methods:**

* **Constructor:** Customer(String customerId, String name, String contactInfo)
  + Initializes a new Customer object.
* **Getters:** getCustomerId(), getName(), getContactInfo()
  + Provide read-only access to the customer's attributes.
* **toString():**
  + Provides a string representation of the Customer object.

**2.3. RentalTransaction Class**

**Purpose:** Records the details of a single instance of a car rental.

**Attributes:**

* transactionId (String): A unique identifier for the transaction.
* rentedCar (Car): A reference to the Car object involved in this transaction.
* customer (Customer): A reference to the Customer object who made the rental.
* rentalStartDate (LocalDate): The date when the rental period begins. (Uses java.time.LocalDate for robust date handling).
* rentalEndDate (LocalDate): The date when the rental period is expected to end.
* totalCost (double): The calculated cost of the rental.

**Methods:**

* **Constructor:** RentalTransaction(String transactionId, Car rentedCar, Customer customer, LocalDate rentalStartDate, LocalDate rentalEndDate)
  + Initializes a new RentalTransaction object.
  + Automatically calculates totalCost upon creation.
* **calculateTotalCost() (private):**
  + Calculates the total cost by multiplying the rentedCar's dailyRentalRate by the number of days in the rental period. The duration includes both the start and end dates.
* **Getters:** getTransactionId(), getRentedCar(), getCustomer(), getRentalStartDate(), getRentalEndDate(), getTotalCost()
  + Provide read-only access to the transaction's attributes.
* **toString():**
  + Provides a string representation of the RentalTransaction object.

**2.4. RentalAgency Class (Halcion Motors)**

**Purpose:** Acts as the central hub of the car rental system, managing the fleet, customers, and orchestrating rental activities.

**Attributes:**

* agencyName (String): The name of the rental agency (e.g., "Halcion Motors").
* cars (List): A collection to store all cars managed by the agency.
* customers (List): A collection to store all registered customers.
* transactions (List): A collection to store all historical and active rental transactions.
* nextTransactionId (int): A simple counter to generate unique transaction IDs.

**Methods:**

* **Constructor:** RentalAgency(String agencyName)
  + Initializes the agency with its name and empty lists for cars, customers, and transactions.
* **addCar(Car car):**
  + Adds a Car object to the agency's fleet. Prevents adding cars with duplicate license plates.
* **removeCar(String licensePlate):**
  + Removes a car by its license plate. A car can only be removed if it's currently available.
* **findCar(String licensePlate):**
  + Searches for a car by its license plate. Returns an Optional<Car>.
* **getAvailableCars():**
  + Returns a list of Car objects that are currently available for rent.
* **getAllCars():**
  + Returns a list of all Car objects in the fleet.
* **registerCustomer(Customer customer):**
  + Registers a Customer object with the agency. Prevents registering customers with duplicate IDs.
* **findCustomer(String customerId):**
  + Searches for a customer by their ID. Returns an Optional<Customer>.
* **getAllCustomers():**
  + Returns a list of all registered Customer objects.
* **rentCar(String licensePlate, String customerId, int durationDays):**
  + The core rental logic.
  + Finds the specified car and customer.
  + Checks if the car is available.
  + If conditions are met, it marks the car as rented, creates a new RentalTransaction, and adds it to the transactions list.
  + Returns an Optional<RentalTransaction>.
* **returnCar(String licensePlate):**
  + Processes the return of a car.
  + Finds the car and marks it as available again.
* **getAllTransactions():**
  + Returns a list of all recorded RentalTransaction objects.
* **getAgencyName():**
  + Returns the name of the agency.

**3. Main Class (Demonstration)**

The Main.java file serves as a demonstration of how to use the RentalAgency class and its associated entities. It includes:

* Initialization of the Halcion Motors agency.
* Adding various cars to the fleet.
* Registering several customers.
* Performing successful and unsuccessful car rental attempts.
* Displaying available cars before and after rentals.
* Displaying all recorded transactions.
* Demonstrating car return functionality.
* Demonstrating car removal.

This class is for testing the system's flow and interaction, rather than being part of the core business logic.

**4. Test Suite (JUnit)**

A comprehensive suite of JUnit 5 test classes (CarTest.java, CustomerTest.java, RentalAgencyTest.java) has been provided to ensure the correctness and robustness of the system.

**Test Coverage:**

* **CarTest:**
  + Verifies constructor initialization and getter methods.
  + Tests the rentCar() method for successful rental and scenarios where the car is already rented.
  + Tests the returnCar() method for successful return and scenarios where the car is already available.
  + Verifies the toString() representation.
* **CustomerTest:**
  + Verifies constructor initialization and getter methods.
  + Verifies the toString() representation.
* **RentalAgencyTest:**
  + Tests addCar() for successful addition and duplicate license plate scenarios.
  + Tests removeCar() for successful removal, car not found, and attempting to remove a rented car.
  + Tests findCar() for existing and non-existing cars.
  + Tests getAvailableCars() and getAllCars().
  + Tests registerCustomer() for successful registration and duplicate ID scenarios.
  + Tests findCustomer() for existing and non-existing customers.
  + Tests getAllCustomers().
  + Tests rentCar() for successful rentals, and various failure scenarios (car not found, customer not found, car not available, invalid duration).
  + Tests returnCar() for successful return, car not found, and attempting to return an already available car.
  + Tests getAllTransactions().

Each test method is annotated with @Test and uses assertions (e.g., assertEquals, assertTrue, assertFalse, assertNotNull) to validate expected behaviors. The @BeforeEach annotation ensures a fresh state for the agency, cars, and customers before every test, preventing side effects between tests.

**5. How to Compile and Run**

To compile and run this Java project:

1. **Save the files:** Save each .java file into a directory (e.g., CarRentalSystem).
2. **Install JUnit (for testing):** If you are using a build tool like Maven or Gradle, add the JUnit 5 dependencies as shown in the previous section. If compiling manually, you'll need to download the JUnit JARs and include them in your classpath during compilation and execution.
3. **Compile:** Open a terminal or command prompt, navigate to the directory containing your .java files, and compile them:
4. javac \*.java

If using JUnit, ensure JUnit JARs are in your classpath.

1. **Run the Main application:**
2. java Main
3. **Run the Tests (using JUnit Runner or IDE):** If using an IDE like IntelliJ IDEA or Eclipse, you can right-click on the test files and select "Run Tests." If running from the command line (after compiling with JUnit in classpath):
4. java -jar <path\_to\_junit\_platform\_console\_standalone.jar> --scan-classpath

(You might need to download junit-platform-console-standalone.jar for command-line execution).

**6. Design Principles Applied**

* **Encapsulation:** Each class (Car, Customer, RentalTransaction, RentalAgency) encapsulates its data (attributes) and behavior (methods) within itself. Attributes are private, and access is provided via public getter and setter (or state-modifying) methods.
* **Modularity:** The system is broken down into small, self-contained units (classes), making it easier to understand, develop, and maintain.
* **Separation of Concerns:** Each class has a single, well-defined responsibility (e.g., Car manages car properties, Customer manages customer properties, RentalAgency orchestrates the rental business logic).
* **Abstraction:** Users of the RentalAgency class don't need to know the internal details of how cars or customers are stored or how transactions are calculated; they just interact with the public methods.
* **Polymorphism (Potential Future Extension):** While not heavily used in this basic version, polymorphism could be introduced for different car types (e.g., LuxuryCar, EconomyCar extending a base Car class) or different rental policies.
* **Composition:** The RentalAgency class "has-a" relationship with Car, Customer, and RentalTransaction objects through its List attributes.

This structured approach ensures a robust, extensible, and easy-to-manage car rental system.