ASTANA IT UNIVERSITY 2025

# CAR RENTAL SYSTEM

PROJECT OF THE MMEM
RENTAL GROUP FROM
THE SE-2409:

# OVERVIEW TO THE CAR RENTAL SYSTEM

Car Rental System is a Java-based application designed to manage car rentals. It includes functionalities like car management, rental tracking, customer and staff management, overdue fine calculation, and more. The system aims to streamline the workflow for both customers and administrators, ensuring a smooth experience for renting cars and managing rentals.

# FEATURES

## CAR MANAGEMENT 🚗

**Add Cars:** Admins can add new cars to the system by providing details such as: Car ID (unique identifier); Model Brand; Availability status (Available, Rented); Rental price per day

**Update Cars:** Admins and staff can update the car details, such as availability status or rental price.

**Delete Cars:** Remove cars from the system that are no longer available or in service.

**Search Cars by ID:** Users and admins can search for cars using their unique IDs for quick access to their details.

```
6. Remove Car
7. Manage Rentals
8. View All Cars
0. Exit
8
Viewing All Cars:
Car ID: 2, Brand: Toyota, Model: RAV4, Price per day: 65,00, Available: true, Category: SUV
Car ID: 4, Brand: Chevrolet, Model: Silverado, Price per day: 70,00, Available: true, Category: TRUCK
Car ID: 6, Brand: Tesla, Model: Model S, Price per day: 120,00, Available: true, Category: ELECTRIC
Car ID: 7, Brand: Toyota, Model: Prius, Price per day: 60,00, Available: true, Category: HYBRID
Car ID: 8, Brand: Jeep, Model: Wrangler, Price per day: 85,00, Available: true, Category: SUV
Car ID: 9, Brand: Mini, Model: Cooper, Price per day: 48,00, Available: true, Category: HATCHBACK
Car ID: 10, Brand: Ram, Model: 1500, Price per day: 75,00, Available: true, Category: TRUCK
Car ID: 5, Brand: Porsche, Model: 911, Price per day: 150000,00, Available: false, Category: SPORTS_CAR
Car ID: 3, Brand: Volkswagen, Model: Golf, Price per day: 50,00, Available: false, Category: HATCHBACK
Car ID: 1, Brand: Ford, Model: Focus, Price per day: 55,00, Available: false, Category: SEDAN
```

# FEATURES

## CUSTOMER MANAGEMENT 👥

**Add Customers:** Admins can register new customers with: ID (unique identifier); Full Name; Email Address ; Password (for login)

**Login/Sign-Up:** Customers can sign up and log in to manage their rentals, view available cars, and make reservations.

## STAFF MANAGEMENT 👨‍💼👩‍💼

**Add Staff:** Admins can add staff members with: Name; Surname; Password (for login)

**Staff Actions:** Staff can handle car rental requests, manage car availability, and track rental statuses.

```
Select your role:
1. Admin
2. Manager
3. Customer
3
Enter your ID: 7
Enter your Password: password5
Login successful!
Customer Menu:
1. Start rental process
2. View available cars
0. Exit
```

```
C:\Users\Acer\.jdks\openjdk-23.0.2\b
Select your role:
1. Admin
2. Manager
3. Customer
1
Enter your ID: 3
Enter your Password: password1
Login successful!

===== ADMIN MENU =====
1. Add Manager
2. Remove Manager
3. View Managers
4. View Customers
5. Add Car
6. Remove Car
7. Manage Rentals
8. View All Cars
0. Exit
```

```
Select your role:
1. Admin
2. Manager
3. Customer
2
Enter your ID: 4
Enter your Password: password2
Login successful!

===== MANAGER MENU =====
1. Approve/Reject Rentals
2. View Rented Cars
3. Manage Car Status
4. View Customers
5. Add Customer
6. Remove Customer
7. View All Cars
0. Exit
```

# FEATURES

```
List of Rentals:
1. Rental ID: 6, Car ID: 7, User ID: 9, Start Date: 2025-12-28, End Date: 2025-12-31, Status: PENDING
2. Rental ID: 4, Car ID: 7, User ID: 6, Start Date: 2025-09-01, End Date: 2025-09-10, Status: REJECTED
3. Rental ID: 8, Car ID: 7, User ID: 2, Start Date: 2026-12-12, End Date: 2026-12-13, Status: PENDING
4. Rental ID: 3, Car ID: 8, User ID: 5, Start Date: 2025-07-08, End Date: 2025-07-25, Status: Rejected
5. Rental ID: 7, Car ID: 7, User ID: 1, Start Date: 2026-08-09, End Date: 2026-08-12, Status: Rejected
Enter rental ID to approve/reject (or 0 to return):
```

## RENTAL MANAGEMENT 🔄

**Track Rentals:** Admins and staff can view all active rentals, including the customer's details, car rented, and due dates.

**Rent Cars:** Customers can rent available cars by selecting a car from the catalog. The system records the rental date, due date, and payment details.

**Return Cars:** Customers can return cars, and the system automatically updates the car's availability, tracks rental duration, and calculates overdue fines if applicable.

**Due Date Calculation:** The system automatically calculates the due date based on the library's policy (e.g., 7 days from the rental date). If a car is returned late, the system applies an overdue fine.

## OVERDUE FINES 💰

**Late Fees:** If a car is returned after the due date, the system calculates overdue fines based on the number of late days and applies it to the customer's account.

VISION STATEMENT

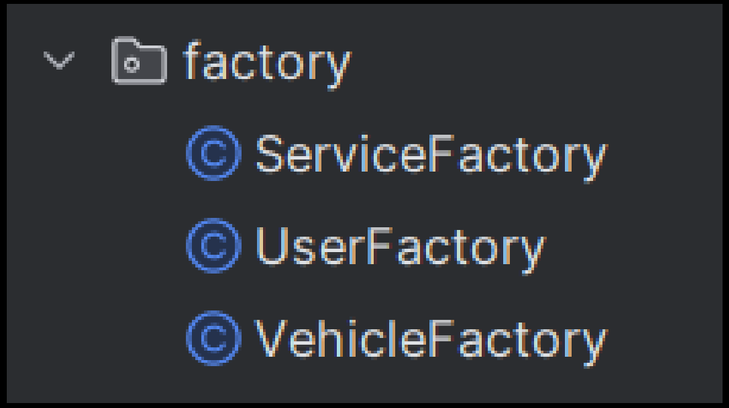


# APPLICATION OF DESIGN PATTERNS

# SINGLETON PATTERN

## APPLIED IN SERVICEFACTORY, USERFACTORY, AND VEHICLEFACTORY

- Each of these classes ensures that only one instance of the factory exists throughout the application's lifecycle.
- This is achieved using a private static instance and the getInstance() method, which creates the object only once when it is first called.




```
private static ServiceFactory instance;
```

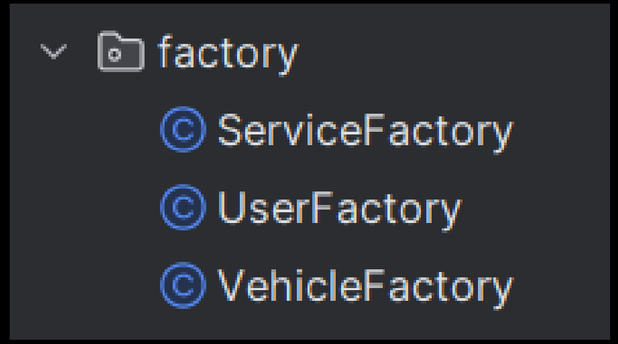
```
public static ServiceFactory getInstance()
    if (instance == null) {
        instance = new ServiceFactory();
    }
    return instance;
}
```

## WHY USE IT?

- Saves resources: Services are created once and reused.
- Centralized control over object management.

# FACTORY

✅ Centralizes object creation – no need to write new Class() everywhere.
✅ Simplifies maintenance – if the object constructor changes, only the factory needs to be updated.
✅ Improves scalability – new objects can be added without modifying the main code.




## SERVICE FACTORY (SINGLETON + FACTORY)

- Manages services (CarService, UserService, RentalService).
- Creates services once and returns the existing instance.

```
public CarService getCarService() { return carService; }
public UserService getUserService() { return userService; }
public RentalService getRentalService() { return rentalService; }
```

## USER FACTORY (FACTORY METHOD)

- Creates User objects, ensuring consistent instantiation.

## VEHICLE FACTORY (FACTORY METHOD)

- Creates Car objects, hiding the instantiation logic.

## EXAMPLE 1 : USING SWITCH-CASE WITH ->

- The -> operator is used in switch-case, making the syntax more compact than traditional case X: { ... break; }.
- This feature, known as "switch expressions", follows a lambda-like style.

```
public void handleChoice(int choice) {
    try {
        switch (choice) {
            case 1 -> rentalController.manageRentals(new Scanner(System.in));
            case 2 -> rentalController.viewRentedCars();
            case 3 -> carController.manageCarStatus(new Scanner(System.in));
            case 4 -> userController.displayCustomers();
            case 5 -> userController.addUser(new Scanner(System.in), role: "CUSTOMER");
            case 6 -> userController.removeUser(new Scanner(System.in), role: "CUSTOMER");
            case 7 -> carController.viewAllCars();
            case 0 -> System.out.println("Exiting Manager Menu...");
            default -> System.out.println("Invalid choice, please try again.");
        }
    } catch (SQLException e) {
        System.out.println("Database error: " + e.getMessage());
    }
}
```

✅ Advantage:

- More concise – removes the need for break; statements.
- Easier to read – each action is defined in a single line.

# APPLICATION OF LAMBDA EXPRESSIONS

## EXAMPLE 2 : USING FOREACH FOR ITERATING OVER RENTALS

- allCars.forEach(System.out::println); is a shorter alternative to a traditional for loop.
- Instead of manually iterating over the list, it applies the System.out::println method to each item.

```
public void removeCar(Scanner scanner) throws SQLException {  2 usage
    System.out.println("Available Cars:");
    List<Car> allCars = carService.getAllCars();
    if (allCars.isEmpty()) {
        System.out.println("No cars available.");
    } else {
        allCars.forEach(System.out::println); // Lambda expression
        System.out.print("Enter Car ID to remove: ");
        int carId = scanner.nextInt();
        scanner.nextLine();

        carService.removeCar(carId);
        System.out.println("Car successfully removed.");
    }
}
```

✅ Advantage:

- More readable and concise than for loops.
- Uses method references, making the code cleaner.

# APPLICATION OF LAMBDA EXPRESSIONS

VISION STATEMENT

Single Responsibility Principle

Open/Closed Principle

Liskov Substitution Principle

Interface Segregation Principle

Dependency Inversion Principle

# SOLID PRINCIPLES

# S – SINGLE RESPONSIBILITY PRINCIPLE

## "A CLASS SHOULD HAVE ONLY ONE REASON TO CHANGE."

Application in Code:

- CarService, UserService, RentalService – each class is responsible for only one entity (cars, users, rentals).
- CarController, UserController, RentalController – these classes are responsible only for user interaction.

```
public class CarService {  16 usages   Ekosik7 +2
    private IDB db;  8 usages

    public CarService(IDB db) { this.db = db; }
```

✅ Why does this code follow SRP?

- The CarService class is responsible only for handling cars (retrieving the list, deleting cars).
- It does not include methods related to users (User) or rentals (Rental), making the code flexible and easy to maintain.

# O – OPEN/CLOSED PRINCIPLE

## "CLASSES SHOULD BE OPEN FOR EXTENSION BUT CLOSED FOR MODIFICATION."

Application in Code:

- If a new car category needs to be added, a new VehicleCategory can be created without modifying the existing code.

```
public interface IDB {  8 usages  1 implementation  Ekosik7 +
    Connection getConnection() throws SQLException;  2
    void close() throws SQLException;  no usages  1 implem
}
```

Now PostgreDB implements this interface:

```
public Connection getConnection() throws SQLException {
    if (connection == null || connection.isClosed()) {
        connection = DriverManager.getConnection(
                url: "jdbc:postgresql://" + host + "/" + database, user, password
        );
    }
    return connection;
}
```

✅ Why does this follow OCP?
If you want to use MySQL in the future, you can simply create a new class without modifying the existing code.

# L – LISKOV SUBSTITUTION PRINCIPLE

***"SUBCLASS OBJECTS SHOULD REPLACE PARENT CLASS OBJECTS WITHOUT CHANGING THE PROGRAM'S BEHAVIOR."***

Application in Code:

- Inheritance Vehicle → Car – a Vehicle can be used instead of Car, and the code will continue to work correctly.

✅ Why does this follow LSP?
Car extends Vehicle while preserving its functionality.
You can use a Car object wherever a Vehicle is expected without breaking the code.

# I – INTERFACE SEGREGATION PRINCIPLE

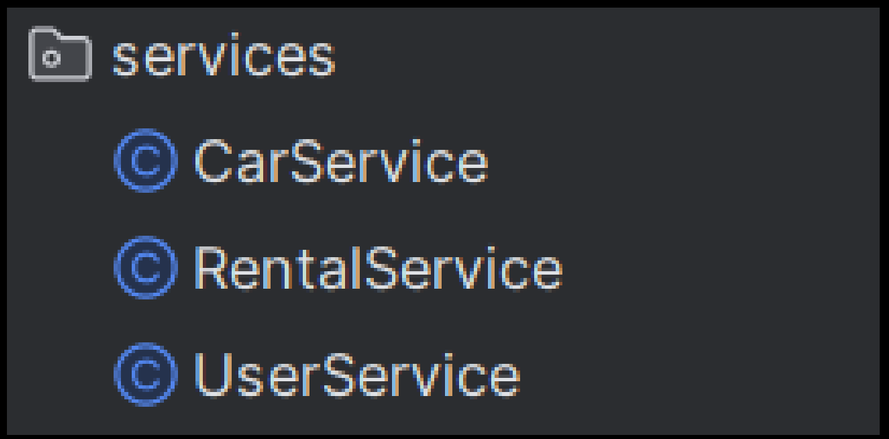
***"CLIENTS SHOULD NOT BE FORCED TO DEPEND ON INTERFACES THEY DO NOT USE."***

Application in Code:

- Service classes (CarService, UserService, RentalService) are separated and depend only on the necessary methods.

✅ You didn't create a single service for everything but instead divided it into multiple specialized services.

- UserService works only with users, without unnecessary methods related to cars or rentals.
- CarService does not include user management methods – each class performs only its specific task.



# D – DEPENDENCY INVERSION PRINCIPLE

## "HIGH-LEVEL MODULES SHOULD NOT DEPEND ON LOW-LEVEL MODULES. BOTH SHOULD DEPEND ON ABSTRACTIONS."

Application in Code:

- Controllers (CarController, UserController) depend on services (CarService) rather than a specific database.

- ServiceFactory does not depend on a specific database but uses the Database interface.
- If you want to use MySQLDB in the future, you only need to pass it to ServiceFactory without modifying the service code.

```
private ServiceFactory() {  1 usage  Ekosik7 *
    PostgreDB db = new PostgreDB(); // ✅ Dependency on an abstraction, not a concrete class
    carService = new CarService(db);
    userService = new UserService(db);
    rentalService = new RentalService(db);
}
```


THANK YOU