

Object Oriented Programming

Individual Project

Rent A Car

Name:	P.A.D.B.Dilshan
Index No:	UWU/CST/22/014
Course Code :	CST 124-2

Technologies used in the application:	3
Database Design	3
Tables:	3
Use Cases:.....	4
Classes used in the Application	5
ConnectSQL Class	6
SQLstatements Class	10
Owner Class.....	14
CarDetails Class.....	16
Customer Class	20
Booking Class	24
Payments Class	28
Calculation Logic	31
Report Class.....	32
StopProgrammeException Class	33
Run the program using .jar file	Error! Bookmark not defined.

Scenario : **Software solution for a Car Rental Agency.**

Description of the scenario I am going to address :

Car rental agency must manage their available cars for a better customer satisfaction. Customer should easily pick the car they want without a complex process. It's a must to see how much they will need to pay for the days they are booking the car. Thus, customers can easily get an idea of how much should they pay before renting a car. That way they can easily rent a car which fits their budget.

Technologies used in the application:

JAVA Programming

(using OOP programming paradigm)

MySQL for Database management

JDBC API for connect MySQL with JAVA application.

Database Design

Tables:

CarDetails Table

Columns:

1. Car_id
2. Brand_Name
3. Model_Name
4. Model_Year
5. Cost
6. isAvailable

Customers Table:

Columns:

1. Customer_id
2. Name
3. Address
4. Phone_number
5. NIC

Bookings Table:

Columns:

1. Booking_id
2. Booking_date
3. Due_date
4. Car_id
5. Customer_id

Payments Table:

Columns:

1. Payment_id
2. Booking_id
3. Total_Payment

Use Cases:

Use Case 1: Customer Booking a Car.

Actors: Customer

Description: The customer searches for available cars and books one.

Use Case 2: Owner Adding a New Car

Actors: Car Owner

Description: The owner adds new car details to the database.
Owner checks all transactions happened.

Classes used in the Application

1. Main

Runs the programme.
Small no of objects created in the main group, Only most wanted methods run on this main class.

2. ConnectSQL

Used to connect mySQL local server with Java Application
- Using JDBC API

3. SQLstatements

Used a separate class to handle mySQL DDL and DML.

4. Owner

Owner access is granted through a password. Making high security. Customer is restricted to access this section by a password.

5. CarDetails

Encapsulate all methods and details about cars in Car Agency.
25 cars will be automatically added to the database called carDetails.

6. Customer

Customer details will be collected through this class.

7. Booking

Letting customer select the car they want this class is relevant for make a booking.

8. Payments

Payment class is basically for calculation of the car rent based on the day count that customer wanna rent the selected car.

9. Report

This class is for owner's side. Owner can get a report of all transactions about the car bookings happened all the time.

10. StopProgrammeException

Custom class to do error handling for invalid inputs.

Explanations and Source Code

ConnectSQL Class

Used to connect mySQL database with JAVA Application.

```
package RentACar;

import java.sql.*;

9 usages  ▲ banuka2001
public class ConnectSQL {
    1 usage
    private static final String JDBC_URL = "jdbc:mysql://localhost:3306/";
    3 usages
    private static String userName;
    3 usages
    private static String password;
    5 usages
    private static Connection connection = null;

    1 usage
    SQLstatements sql = new SQLstatements();

    1 usage
    CarDetails carDetails = new CarDetails();
    1 usage  ▲ banuka2001
    public static void setCredentials(String user, String pass) {
        userName = user;
        password = pass;
    }
}
```

"jdbc:mysql://localhost:3306/" is related to the mySQL local host server.

Username and password is User's local sql server's username and password.

"Connection" is an interface which helps to keep connection with the JAVA application and mySql server.

Fisrt its declared as none to create an object of that and later an value is passed to it.

SQL statement class is where I wrote all Sql queries. (which make the programme abstract.)
A new object related to that class is created as "sql".

Class related to car details object is also declared.

Through a contructor user name and password which are gathered by user input is passed to the private variables declared inside the class. Which helps to not reveal the username and password to other classes.

```

public static void establishConnection() {
    try {
        Class.forName("com.mysql.cj.jdbc.Driver");
        System.out.println("JDBC Driver Registered!");

        connection = DriverManager.getConnection(JDBC_URL, userName, password);
        System.out.println("Connection successfully!");

        var sql = new SQLstatements();

        Statement statement = connection.createStatement();

        // Creating the database if it doesn't exist
        statement.executeUpdate(sql.setCreateDatabaseSQL());
        System.out.println("Database 'CarAgencyDB' checked/created.");

        // Connect to Rent a car database
        String jdbcURLWithDatabase = "jdbc:mysql://localhost:3306/CarAgencyDB";
        connection = DriverManager.getConnection(jdbcURLWithDatabase, userName, password);
        System.out.println("Connection to 'carRent' database established successfully!");

        ConnectSQL connectSQL = new ConnectSQL();
        connectSQL.checkAndRun();
    } catch (ClassNotFoundException | SQLException e) {
        e.printStackTrace();
    }
}

```

This method in “ConnctetSQL” class is declared as static, so its scope will go beyond that class.

Through using DriverManager, connection is established using username and password which are collected by user.

`Class.forName("com.mysql.cj.jdbc.Driver");`

- This method is a part of the “java.lang.class” class.
- com.mysql.cj.jdbc.Driver is fully qualified name of the JDBC driver class. This class contains necessary code to communicate with a MySQL database.

Statement is interface which allows to send SQL statements to the database and its need to obtain the results produced by those statements.

To create a statement object we need to use createStatement() method of a ‘connection’ class

I used statement object in order to execute sql commands directly.

executeUpdate() is a method used to update database. First I made sure that database named ‘CarAgencyDB’ is already created in the local sql server or not.

setCreateDatabase() is a method I made inside the SQLstatement class with DDL query to create the database.

Then a new connection with the created database is established.

Because the establishConnection() method is static we need to create an object of the same class its exists to use the other methods wrote inside that class.
with using object of that class checkAndRun() method will execute after building a connection to the created or already existed database.

Exception handling is done using try/catch method.

```
1 usage  ▸ banuka2001
public void checkAndRun () throws SQLException {
    if (checkCarDetailTable()) {
        System.out.println("Checked/Car details Table Already Exist in the database!");
    } else {
        carDetails.saveExistingCars();
    }
}
```

This method is used with a method which initiated in SQLstatement class.

checkAndRun() method will check if the Car Details Table is exist or not. If its exists then a user friendly message will be shown. This kind of messages are essential for debugging. if the “if” condition was not true then it will run another method which is in carDetails class which used to save the cars which are already in the agency.

Here is the method used for check if the car details table exist or not.

```
1 usage  ▸ banuka2001
private boolean checkCarDetailTable() throws SQLException {
    PreparedStatement preparedStatement = connection.prepareStatement(sql.checkTableCarDetail());
    preparedStatement.setString( parameterIndex: 1, x: "CarAgencyDB");
    preparedStatement.setString( parameterIndex: 2, x: "CarDetails");

    ResultSet resultSet = preparedStatement.executeQuery();

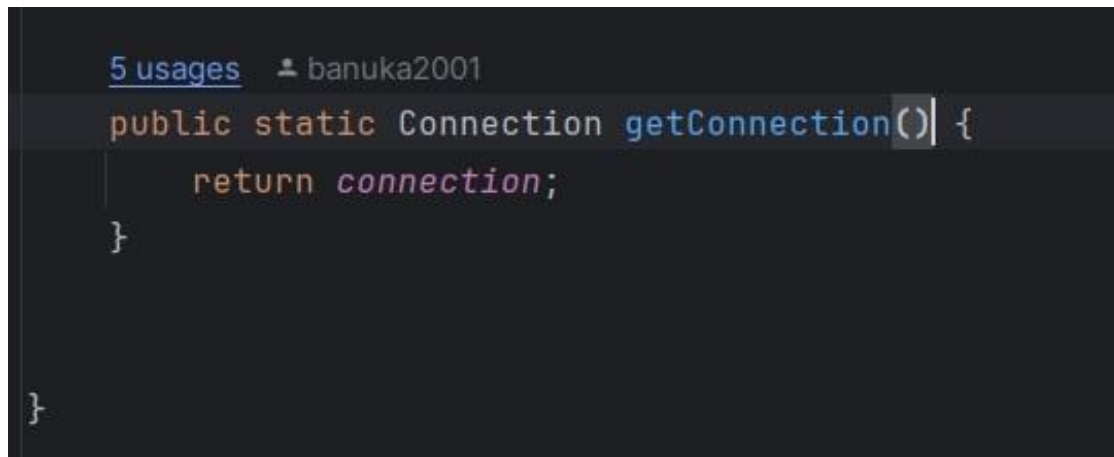
    return resultSet.next();
}
```

“PreparedStatement” is an interface which used for create SQL queries with placeholders (“?”) that can be filled with values at runtime.

using connector object that we have created ‘preparedStatment’ object is created.
This is the related sql query which I wrote in another class to keep a class encapsulated and abstracted.

```
public String checkTableCarDetail() {
    String checkTable = "SELECT 1 FROM information_schema.tables WHERE table_schema = ? AND table_name = ? LIMIT 1";
    return checkTable;
}
```


When using 'PreparedStatement' we need to handle Sql exception. Because if something went wrong this needs to throw a sql exception.
by 'throw' keyword used with the class head we can let this method handle the exception when we using the method.



```
5 usages  banuka2001
public static Connection getConnection() {
    return connection;
}

}
```

This getter is used to get connection object out of that class to use in other class in order to keep connection with MySql database.

SQLstatements Class

This class is used to encapsulate the SQL queries to the same place.

Let's breakdown the methods of this class.

```
package RentACar;

13 usages  📄 banuka2001
public class SQLstatements {

    1 usage  📄 banuka2001
    public String setCreateDatabaseSQL() {
        String createDatabaseSQL = "CREATE DATABASE IF NOT EXISTS CarAgencyDB";
        return createDatabaseSQL;
    }
}
```

This method is used to initiate sql statement related to database creation if CarAgencyDB not exists.

```
1 usage  📄 banuka2001
public String createCarDetails () {
    String createTableSQL = "CREATE TABLE IF NOT EXISTS CarDetails ( " +
        "    Car_id INT PRIMARY KEY AUTO_INCREMENT, " +
        "    Brand_Name VARCHAR(50), " +
        "    Model_Name VARCHAR(50), " +
        "    Model_Year INT, " +
        "    Cost DECIMAL(10, 2), " +
        "    isAvailable BOOLEAN DEFAULT TRUE " +
        ") ";
    return createTableSQL;
}

1 usage  📄 banuka2001
public String createCustomerDetails () {
    String createTableSQL = "CREATE TABLE IF NOT EXISTS CustomerDetails ( " +
        "    Customer_id INT PRIMARY KEY AUTO_INCREMENT, " +
        "    Name VARCHAR(100), " +
        "    Address VARCHAR(255), " +
        "    Phone_number CHAR(10), " +
        "    NIC VARCHAR(15) UNIQUE " +
        ") ";
    return createTableSQL;
}
```

```

1 usage  ± banuka2001
public String createBookingDetails () {
    String createTableSQL = "CREATE TABLE IF NOT EXISTS BookingDetails (" +
        "    Booking_id INT PRIMARY KEY AUTO_INCREMENT, " +
        "    Booking_date DATE, " +
        "    Due_date DATE, " +
        "    Car_id INT, " +
        "    Customer_id INT, " +
        "    FOREIGN KEY (Car_id) REFERENCES CarDetails(Car_id), " +
        "    FOREIGN KEY (Customer_id) REFERENCES CustomerDetails(Customer_id) " +
        ") ";
    return createTableSQL;
}

1 usage  ± banuka2001
public String createPaymentDetails () {
    String createTableSQL = "CREATE TABLE IF NOT EXISTS PaymentDetails ( " +
        "    Payment_id INT PRIMARY KEY AUTO_INCREMENT , " +
        "    Booking_id INT , " +
        "    Total_Payment DECIMAL(10, 2), " +
        "    FOREIGN KEY (Booking_id) REFERENCES BookingDetails(Booking_id) " +
        ") ";
    return createTableSQL;
}

```

These methods are for 4 Table creation sql statements.

The 4 tables are;

CarDetails, CustomerDetails, BookingDetails, PaymentDetails.

```

//Add table details statements
2 usages  ± banuka2001
public String setCarDetails() {
    String insertTableData = "INSERT INTO CarDetails (Brand_Name, Model_Name, Model_Year, Cost) VALUES (?, ?, ?, ?)";
    return insertTableData;
}

1 usage  ± banuka2001
public String setCustomerDetails () {
    String insertTableData = "INSERT INTO CustomerDetails (Name, Address, Phone_number, NIC) VALUES (?, ?, ?, ?)";
    return insertTableData;
}

1 usage  ± banuka2001
public String setBookingDetails () {
    String insertTableData = "INSERT INTO BookingDetails (Booking_date, Due_date, Car_id, Customer_id) VALUES (?, ?, ?, ?)";
    return insertTableData;
}

1 usage  ± banuka2001
public String setPaymentDetails () {
    String insertTableData = "INSERT INTO PaymentDetails (Booking_id, Total_Payment) VALUES (?, ?)";
    return insertTableData;
}

```

These methods are for inserting values for above created tables.

This have placeholders to add values. So by using 'preparedStatment' object we can manipulate those table with the values we desire.

```
// Show Table data queries
1 usage  ↳ banuka2001
public String showCarDetails() {
    String showDetails = "SELECT * FROM CarDetails WHERE isAvailable = 1";
    return showDetails ;
}

no usages  ↳ banuka2001
public String showCustomerDetails() {
    String showDetails = "SELECT * FROM CustomerDetails";
    return showDetails;
}

no usages  ↳ banuka2001
public String showBookingDetails() {
    String showDetails = "SELECT * FROM BookingDetails";
    return showDetails;
}

no usages  ↳ banuka2001
public String showPaymentDetails() {
    String showDetails = "SELECT * FROM PaymentDetails";
    return showDetails;
}
```

These are the methods which initiate show table sql queries. Which used to retrieve data from the tables.

```
//Check if CarDetail table already exist or not
1 usage  ↳ banuka2001
public String checkTableCarDetail() {
    String checkTable = "SELECT 1 FROM information_schema.tables WHERE table_schema = ? AND table_name = ? LIMIT 1";
    return checkTable;
}
```

This is the method with sql query to check if the CarDetail table already exist or not.

```
1 usage  ↳ banuka2001
public String updateCars() {
    String updateCars = "UPDATE CarDetails SET isAvailable = 0 WHERE Car_id = ? AND isAvailable = 1";
    return updateCars;
}
```

This is the method with sql queries used to update car availability if a customer rented out a car.

```

1 usage  ± banuka2001
public String generateReport() {
    String report = "SELECT " +
        "    CarDetails.Brand_Name AS Car_Brand, " +
        "    CarDetails.Model_Name AS Car_Model, " +
        "    CustomerDetails.Name AS Customer_Name, " +
        "    PaymentDetails.Total_Payment AS Payment_Amount, " +
        "    BookingDetails.Due_date AS Due_Date, " +
        "    CustomerDetails.Phone_number AS number " +
        "FROM " +
        "BookingDetails " +
        "JOIN " +
        "CarDetails ON BookingDetails.Car_id = CarDetails.Car_id " +
        "JOIN " +
        "CustomerDetails ON BookingDetails.Customer_id = CustomerDetails.Customer_id " +
        "JOIN " +
        "PaymentDetails ON BookingDetails.Booking_id = PaymentDetails.Booking_id;";
    return report;
}

```

This is a method with sql query which used to generate a report for the all transactions happened.

```

// getters

1 usage  ± banuka2001
public String getCustomerIdQuery() {
    return "SELECT Customer_id FROM CustomerDetails WHERE NIC = ?";
}

1 usage  ± banuka2001
> public String getCarRent() { return "SELECT Cost FROM CarDetails WHERE Car_id = ?"; }

1 usage  ± banuka2001
> public String getBookingId() { return "SELECT Booking_id FROM BookingDetails WHERE Car_id = ?"; }
}

```

Those are the getters to retrieve specific data from the tables for further use.

- This is a class of SQL statements for creation, manipulation, and other operations over the database in your car rental system. It encapsulates all SQL queries related to creating a database, manipulating its tables, inserting data, retrieving, updating, and reporting of data. This improves the organization, reusability, and simplicity of maintaining the code by separating the SQL logic from the rest of the application.

Owner Class

Purpose of the Owner class is to allow the owner to either add new cars into the database or check the transaction records. It lets user interaction with the system through the password authentication. (Password set as "Hello" for demonstration.)

The instance of the CarDetails class that executes all operations involving car information in the Database end.

An instance of the Report class: for generating and handling reports regarding transactions and other data.

A Scanner object that reads user input from the console.

Methods :

public void ownerAccess() throws SQLException

It provides a menu-driven interface to the owner for adding new cars into the database or for checking transaction records.

Process:

The user is prompted to enter the access password of the owner.

The entered password is checked for matching with the predefined password.
Here it is set as "Hello".

In case of correct password,

The user is asked whether they want to "Add" new cars or "Check" transaction records.

If "Add" is selected, it invokes the addExtraCars() method of class CarDetails for adding new cars.

If "Check" is selected, it invokes the generateReport() method of class Report to generate a transaction report.

If none of the above options is selected, it prints the message,
"Access Denied".

The class is designed to be used by the owner of the car rental for the administrative role. With it, an owner is able to manage entries regarding cars and can view transaction records. The ownerAccess method handles user authentication and provides a simple command-line interface for the owner to select whether to add cars or check reports.

Error Handling is done.

The ownerAccess() method throws an SQLException, indicating that database-related errors might occur when performing, for example, add car or generate report operations. Proper error handling for these operations should be managed where the ownerAccess() method is called.

CODE :

```

package RentACar;

import java.sql.SQLException;
import java.util.Scanner;

2 usages  ▲ banuka2001
public class Owner {

    1 usage
    CarDetails carDetails = new CarDetails();

    1 usage
    Report report = new Report();

    2 usages
    Scanner scanner = new Scanner(System.in);

    1 usage
    private final String password = "Hello";

```

```

1 usage  ▲ banuka2001
public void ownerAccess() throws SQLException {

    System.out.print("To Add new cars to the system enter the Owner Access Password: ");

    String input = scanner.nextLine();

    if (input.equals(password)) {
        System.out.println("Do you want to add Extra class to the database (Type: Add)\n ----- Or Do you want to check transaction records (Type: Check)");
        String execute = scanner.nextLine();
        if (execute.equalsIgnoreCase( anotherString: "add")) {
            carDetails.addExtraCars();
        } else if (execute.equalsIgnoreCase( anotherString: "check")) {
            report.generateReport();
        } else {
            System.out.println("Access Denied !!!");
        }
    }
}
}

```

CarDetails Class

The `CarDetails` class plays a key role in handling car data in the database. This class contains functions to save current car info, put in more cars if owner wants to, and Retrieving car details from the database.

Connection connection : A connection to the MySQL database, established via the `ConnectSQL.getConnection()` method.

SQLstatements sql : An instance of a class containing SQL queries as methods, used throughout this class.

Report report: An instance of the `Report` class.

```
package RentACar;

import java.sql.*;
import java.util.Scanner;

6 usages  ▲ banuka2001 *
public class CarDetails {
    5 usages
    private Scanner scanner = new Scanner(System.in);
    4 usages
    Connection connection = ConnectSQL.getConnection();
    4 usages
    SQLstatements sql = new SQLstatements();

    1 usage
    Report report = new Report();
```

Methods:

1. saveExistingCars()

To create the `CarDetails` table if it doesn't exist and populate it with a predefined list of car details

```
1 usage  ▲ banuka2001 *
public void saveExistingCars() {

    try {
        Statement statement = connection.createStatement();

        // Creating the table if it doesn't exist
        statement.executeUpdate(sql.createCarDetails());
        System.out.println("Table 'Car Owner' checked/created.");

        // Insert data into the table
        PreparedStatement preparedStatement = connection.prepareStatement(sql.setCarDetails());
```



```
// data set to update car details
setCarDetails(preparedStatement, brandName: "Toyota", modelName: "Camry", modelYear: 2022, cost: 30000.00);
setCarDetails(preparedStatement, brandName: "Honda", modelName: "Civic", modelYear: 2021, cost: 25000.00);
setCarDetails(preparedStatement, brandName: "Ford", modelName: "Focus", modelYear: 2020, cost: 28000.00);
setCarDetails(preparedStatement, brandName: "Chevrolet", modelName: "Malibu", modelYear: 2019, cost: 27000.00);
setCarDetails(preparedStatement, brandName: "BMW", modelName: "3 Series", modelYear: 2018, cost: 45000.00);
setCarDetails(preparedStatement, brandName: "Mercedes", modelName: "C-Class", modelYear: 2022, cost: 48000.00);
setCarDetails(preparedStatement, brandName: "Audi", modelName: "A4", modelYear: 2021, cost: 47000.00);
setCarDetails(preparedStatement, brandName: "Volkswagen", modelName: "Passat", modelYear: 2020, cost: 35000.00);
setCarDetails(preparedStatement, brandName: "Nissan", modelName: "Altima", modelYear: 2019, cost: 22000.00);
setCarDetails(preparedStatement, brandName: "Hyundai", modelName: "Elantra", modelYear: 2018, cost: 21000.00);
setCarDetails(preparedStatement, brandName: "Kia", modelName: "Optima", modelYear: 2022, cost: 23000.00);
setCarDetails(preparedStatement, brandName: "Mazda", modelName: "3", modelYear: 2021, cost: 24000.00);
setCarDetails(preparedStatement, brandName: "Subaru", modelName: "Impreza", modelYear: 2020, cost: 26000.00);
setCarDetails(preparedStatement, brandName: "Lexus", modelName: "ES", modelYear: 2019, cost: 40000.00);
setCarDetails(preparedStatement, brandName: "Acura", modelName: "TLX", modelYear: 2018, cost: 38000.00);
setCarDetails(preparedStatement, brandName: "Infiniti", modelName: "Q50", modelYear: 2022, cost: 42000.00);
setCarDetails(preparedStatement, brandName: "Jaguar", modelName: "XE", modelYear: 2021, cost: 46000.00);
setCarDetails(preparedStatement, brandName: "Porsche", modelName: "Macan", modelYear: 2020, cost: 49000.00);
setCarDetails(preparedStatement, brandName: "Volvo", modelName: "S60", modelYear: 2018, cost: 47000.00);
setCarDetails(preparedStatement, brandName: "Mitsubishi", modelName: "Lancer", modelYear: 2022, cost: 20000.00);
setCarDetails(preparedStatement, brandName: "Chrysler", modelName: "300", modelYear: 2021, cost: 31000.00);
setCarDetails(preparedStatement, brandName: "Jeep", modelName: "Cherokee", modelYear: 2020, cost: 34000.00);
setCarDetails(preparedStatement, brandName: "Dodge", modelName: "Charger", modelYear: 2019, cost: 29000.00);
setCarDetails(preparedStatement, brandName: "Ram", modelName: "1500", modelYear: 2018, cost: 33000.00);
setCarDetails(preparedStatement, brandName: "Peugeot", modelName: "308", modelYear: 2022, cost: 28000.00);
setCarDetails(preparedStatement, brandName: "Renault", modelName: "Megane", modelYear: 2021, cost: 27000.00);

} catch (SQLException e) {
    e.printStackTrace();
}
}
```

2. addExtraCars()

Allows the user to input additional car details that can be saved into the `CarDetails` table.

In this method exception handling is not done inside the method. It must be handle when this method is using.

```
public void addExtraCars() throws SQLException {

    PreparedStatement preparedStatement = connection.prepareStatement(sql.setCarDetails());

    while (true) {
        scanner.nextLine();

        System.out.print("Enter Brand Name (or type 'exit' to leave this section): ");
        String brandName = scanner.nextLine();

        if (brandName.equalsIgnoreCase("exit")) {
            report.generateReport();
            break;
        }

        System.out.print("Enter Model Name: ");
        String modelName = scanner.nextLine();
    }
}
```

```

        System.out.print("Enter Model Year: ");
        int modelYear = scanner.nextInt();

        System.out.print("Enter Rent Cost per month: ");
        double cost = scanner.nextDouble();

        preparedStatement.setString( parameterIndex: 1, brandName);
        preparedStatement.setString( parameterIndex: 2, modelName);
        preparedStatement.setInt( parameterIndex: 3, modelYear);
        preparedStatement.setDouble( parameterIndex: 4, cost);

        preparedStatement.executeUpdate();
    }
}

```

3. getCarDetails()

To retrieve and display all car details from the `CarDetails` table.

```

1 usage  ± banuka2001
public void getCarDetails() {
    try {
        Statement statement = connection.createStatement();
        ResultSet resultSet = statement.executeQuery(sql.showCarDetails());

        System.out.println("Available Cars are: ");

// Refer this
        System.out.printf("%-20s | %-20s | %-10s | %-10s\n",
            "Brand Name", "Model Name", "Model Year", "Rent Per Month");

        while (resultSet.next()) {
            //Brand_Name, Model_Name, Model_Year, Cost
            int Car_id = resultSet.getInt( columnLabel: "car_id");
            String brandName = resultSet.getString( columnLabel: "Brand_Name");
            String modelName = resultSet.getString( columnLabel: "Model_Name");
            int modelYear = resultSet.getInt( columnLabel: "Model_Year");
            double cost = resultSet.getDouble( columnLabel: "Cost");
            boolean Availability = resultSet.getBoolean( columnLabel: "isAvailable");

            System.out.printf("%-10d | %-20s | %-20s | %-10d | $%-10.2f | %-10b\n",
                Car_id, brandName, modelName, modelYear, cost, Availability);
        }
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
}
}

```

4. setCarDetails()

Takes parameters for brand name, model name, model year, and cost, then sets these values into a prepared SQL statement before executing the update.

```
26 usages  A banuka2001
private void setCarDetails (PreparedStatement preparedStatement, String brandName, String modelName, int modelYear, double cost) throws SQLException {
    preparedStatement.setString( parameterIndex: 1, brandName);
    preparedStatement.setString( parameterIndex: 2, modelName);
    preparedStatement.setInt( parameterIndex: 3, modelYear);
    preparedStatement.setDouble( parameterIndex: 4, cost);
    preparedStatement.executeUpdate();
}
```

This method is set to private. So this method can only be used in CarDetails class. This is abstraction of OOP in action.

The CarDetails class gives a way to handle car details in a more organized way in the car rental service.

It provides for the successful insertion and update of car record records in the database, such as the automatic addition of a predefined list of cars to the user and the manual addition of new cars by the user.

Additionally, it gets the available cars, their details and availability status, and displays it from the database.

Exception handling is performed to make sure the application can manage unexpected errors easily, maintaining a frictionless user experience.

This class cleverly handles JDBC API for the tasks related to the database like statement creation, parameter preparation, and exception handling, which consequently leads to the handling of data that is both secured and correct.

Customer Class

The `Customer` class is responsible for activities related to the customer and data management with respect to car rentals. This class facilitates operations such as the creation and updating of customer records in a database, collection and validation of customer details, and management of car rentals. Key functionality among them includes displaying available cars, validation of user inputs.

for example, checking that the phone number is an integer and ensuring that no two customers have the same NIC in the database.

It uses custom designed exception handling for user exits and invalid inputs. Furthermore, it handles SQL exceptions also in case of duplicate entries.

Following are the Object-Oriented Programming principles in this class: encapsulation, abstraction, modularity, reusability, well-structured code, ease of maintenance, and forms an integral part related to application stability and functionality.

Methods :

1. updateCustomers()

```
public void updateCustomers() throws StopProgrammeException {  
    try {  
  
        Statement statement = connection.createStatement();  
        statement.executeUpdate(sql_createCustomerDetails());  
        System.out.println("Customer Table Created Successfully!");  
  
        PreparedStatement preparedStatement = connection.prepareStatement(sql_setCustomerDetails());  
  
        System.out.println("Do you want to check what cars are available (yes/no): ");  
        String inputSelection = scanner.next();  
  
        // view available cars  
        if (inputSelection.equalsIgnoreCase( anotherString: "yes")) {  
            carDetails.getCarDetails();  
        } else {  
  
            throw new StopProgrammeException("User choosed to exit!");  
        }  
    }  
}
```

```
// add customer details  
System.out.print("Do you want to rent out a car ? (yes/no)");  
inputSelection = scanner.next();  
  
if (inputSelection.equalsIgnoreCase( anotherString: "yes")) {  
    collectCustomerDetails(scanner, preparedStatement);  
}
```

```

        collectCustomerDetails(scanner, preparedStatement);
    } else {
        System.out.println("BYE! COME AGAIN");
        throw new StopProgrammeException("User choosed to exit!");
    }
}

} catch (SQLException e) {
    throw new RuntimeException(e);
}
}

```

2. collectCustomerDetails()

```

private void collectCustomerDetails(Scanner scanner, PreparedStatement preparedStatement) throws SQLException, StopProgrammeException {
    // clear input buffer
    scanner.nextLine();
    System.out.println("Enter your following details if you want to rent a car");
    System.out.print("Your first name: ");
    String name = scanner.nextLine();
    System.out.println("Your current address: ");
    String address = scanner.nextLine();
    System.out.println("Your phone no: ");
    String phone = null;

```

```

    //Ensure user enters an integer as phone number
    while (phone == null) {
        try {
            phone = scanner.nextLine();
            Integer.parseInt(phone); // Check if phone is a valid integer
        } catch (NumberFormatException e) {
            throw new StopProgrammeException("Invalid phone number.");
        }
    }

    System.out.println("Your NIC: ");
    NIC = scanner.nextLine();

    if (customerDetailsValidation(name, address, phone, NIC)) {
        setCustomerDetailsTable(preparedStatement, name, address, phone, NIC);
    } else {
        System.out.println("Invalid Inputs! Check again !!!");
    }
}

```

This method is setted as Private, Which limits method usage only inside this Customer class which shows abstraction. Exception handling for SQLException is not handled and it must handle when the class is used.

3. customerDetailsValidation()

```
private boolean customerDetailsValidation(String name, String address, String phone, String NIC) {  
    return !name.isEmpty() && !address.isEmpty() && !phone.isEmpty() && !NIC.isEmpty();  
}
```

This is a private method which only can be used within the class. This method is used in previously mentioned method. This method will make sure the name, address, phone number, NIC number fields are not empty and its entered correctly by user. If user haven't entered any of above fields this notifies the user that the input is invalid and doesn't proceed with storing the data.

4. setCustomerDetailsTable()

```
private void setCustomerDetailsTable(PreparedStatement preparedStatement, String name, String address, String phone, String NIC) {  
    while (true) {  
        try {  
            preparedStatement.setString( parameterIndex: 1, name);  
            preparedStatement.setString( parameterIndex: 2, address);  
            preparedStatement.setString( parameterIndex: 3, phone);  
            preparedStatement.setString( parameterIndex: 4, NIC);  
            preparedStatement.executeUpdate();  
            // System.out.println("Customer details inserted successfully!"); // used for debugging purposes  
            break;  
        }  
    }  
}
```

```
        } catch (SQLException e) {  
            // Check for unique constraint violation by MySQL error code  
            if (e.getErrorCode() == 1062) { // Error code 1062 is for a duplicate entry  
                System.out.println("Error: The NIC '" + NIC + "' already exists. Please enter a unique NIC.");  
                System.exit( status: 1);  
            } else {  
                e.printStackTrace();  
                break; // Exit loop  
            }  
        }  
    }  
}
```

This method is also a private method which used inside previously mentioned collectCustomerDetails() method. This method's job is insert the details entered by user into CustomerDetails Table in database. it use PreparedStatement class's objects for that job.

Exception handling is done and if SQLException is caught it will check that exception message using an if condition. It will check the message and if code 1062 which depicts duplicate entry was found, its generate an user friendly message to enter a unique value to the entry.

4. getLastCustomerId()

```
1 usage  ▸ banuka2001
public int getLastCustomerId () throws SQLException {

    PreparedStatement preparedStatement = connection.prepareStatement(sql.getCustomerIdQuery());
    preparedStatement.setString( parameterIndex: 1, NIC);

    ResultSet resultSet = preparedStatement.executeQuery();
    if (resultSet.next()) {
        customer_id = resultSet.getInt( columnLabel: "Customer_id");
    } else {
        System.out.println("No customer ID found" + NIC);
    }
    // Close resources
    resultSet.close();
    preparedStatement.close();

    return customer_id;
}
```

This method is used in Booking class. This method will look for latest customerId in the CustomerDetails table in the database and it will return its value.

Exception handling is not done inside the method. It is added to method signature. When the method is used exception must be handled in there.

Booking Class

The Booking class is responsible for managing car rental bookings in the RentACar system. The customer can rent cars by interacting with the database and save details of the booking. This class does a lot; it computes due dates and costs of cars, defines the relationship between the customer and car in the rental system, and even fetches car costs.

Below are the primary components of the class:

- SQLstatements sql: An instance of the SQLstatements class, which provides SQL queries needed for various database operations.
- Connection connection: A connection object obtained from ConnectSQL.getConnection() for executing SQL commands.
- Customer customer: An instance of the Customer class, used to manage customer-related operations.
- LocalDate currentDate: Stores the current date, used to determine booking dates.
- Scanner scanner: A Scanner object to capture user input.
- Payments payments: A Payments object, passed through a setter method for managing payment details.
- int days: Stores the number of days a car is rented.
- int carID: Stores the ID of the car being rented.

Methods

1. addBooking()

This is the method responsible for implementing the adding of a new booking. It updates the customer's details by interacting with him, queries the user for the choice of car, and calculates the due date for the rental. The method updates the database on the availability of the car and inserts the details of the booking into the appropriate table.

```
public void addBooking () {  
  
    try {  
        // executing update customer method from Customer class  
        customer.updateCustomers();  
  
        Statement statement = connection.createStatement();  
        statement.executeUpdate(sql.createBookingDetails());  
  
        statement.close();  
  
        int ID = customer.getLastCustomerId(); // getter to access last customerId  
        System.out.println(ID);  
    }  
}
```



```

        System.out.print("Select a car no for booking: ");
        carID = scanner.nextInt();

        updateCarTable();

        // get days from user
        System.out.print("How many days do you like to rent the car you selected: ");
        days = scanner.nextInt();

        PreparedStatement preparedStatement = connection.prepareStatement(sql.setBookingDetails());
        // update booking table
        setBookingDetailsTable(preparedStatement,currentDate, String.valueOf(getDueDate()), carID, ID);

        preparedStatement.close();
    } catch (SQLException | StopProgrammeException e) {
        throw new RuntimeException(e);
    }
}

```

2. updateCarTable()

Updates the car's availability status in the database based on the selected car ID. Throws a custom StopProgrammeException which is custom made exception, if the car ID is invalid.

```

private void updateCarTable() throws StopProgrammeException {
    try {
        // update cars list when a customer booked a car
        PreparedStatement prst = connection.prepareStatement(sql.updateCars());
        prst.setInt( parameterIndex: 1, carID);
        // Execute update query
        int rowsAffected = prst.executeUpdate();
        // Check if the update was successful
        if (rowsAffected > 0) {
            System.out.println("Car available");
        } else {
            System.out.println();
            throw new StopProgrammeException("Check the Car Id again and input correct input!");
        }
        prst.close();
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
}

```

3. getCost()

Retrieves the monthly rental cost of the selected car from the database.

```

public double getCost() throws SQLException {

    double cost = 0;
    //get cost for selected car
    PreparedStatement prsta = connection.prepareStatement(sql.getCarRent());
    prsta.setInt( parameterIndex: 1, carID);
    prsta.executeQuery();
    ResultSet resultSet = prsta.executeQuery();
    while (resultSet.next()) {
        cost = resultSet.getDouble( columnLabel: "Cost");
    }

    prsta.close();
    resultSet.close();
    return cost;
}

```

4. **setBookingDetailsTable**(PreparedStatement preparedStatement, LocalDate currentDate, String dueDate, int carId, int customerId)

A private method used to set the booking details in the database by executing an update on the booking table.

```

private void setBookingDetailsTable (PreparedStatement preparedStatement,LocalDate currentDate,String dueDate, int carId, int customerId) throws SQLException {
    preparedStatement.setString( parameterIndex: 1, String.valueOf(currentDate));
    preparedStatement.setString( parameterIndex: 2,dueDate);

    preparedStatement.setInt( parameterIndex: 3,carId);
    preparedStatement.setInt( parameterIndex: 4,customerId);

    preparedStatement.executeUpdate();
}

```

5. **getDueDate**()

Calculates and returns the due date for the car rental based on the current date and the number of days rented.

```

private LocalDate getDueDate () {
    LocalDate dueDate = currentDate.plusDays(days);
    return dueDate;
}

```

6. getDays()

Returns the number of days for which the car is rented.

```
public int getDays() {  
    return days;  
}
```

7. getLastBookingID()

Retrieves the last booking ID from the database to help update the payment table.

```
public int getLastBookingID () throws SQLException {  
  
    int bookingId = 0;  
    PreparedStatement preparedStatement = connection.prepareStatement(sql.getBookingId());  
    preparedStatement.setInt( parameterIndex: 1, carID);  
    preparedStatement.executeQuery();  
  
    ResultSet resultSet = preparedStatement.executeQuery();  
    while (resultSet.next()) {  
        bookingId = resultSet.getInt( columnLabel: "Booking_id");  
    }  
    return bookingId;  
}
```

The Booking class makes use of inbuilt and custom exceptions for handling errors. Those exceptions include:

1. StopProgrammeException: A custom-made exception which will be raised when the wrong ID of a car is put in by the user. In that way, the program stops gracefully due to any bad inputs.
2. SQLExceptions: These are caught and re-thrown as runtime exceptions to make sure any problem related to the database is properly propagated.

This class encapsulates all the functionality related to bookings. In other words, it encapsulates the data and the methods related to bookings within one unit. The Booking class itself is only responsible for booking operations and leaves to the classes Customer and Payment the respective responsibilities, therefore increasing modularity and making it easier to maintain.

Another point to consider is this class achieves decoupling by using setter methods to inject the instance of the Payments class into the class. It will be easier to modify and test.

Payments Class

The class Payments is responsible for performing all the payment calculations and updates in the system of RentACar. This class cooperates with the Booking class to obtain a booking's details, compute the total cost of rental based on the duration of the car hire, and update information in the database regarding the payment. This class also abides by OOP principles and modularity and separation of concerns principles.

Below are the primary components of the class:

- **int days:** Where it holds the number of days that the car is booked for; it obtains this value from class Booking.
- **double totalCost:** holds total calculated cost for the rental.
- **Booking booking:** This would be a reference to the Booking class. It would be used to obtain any information associated with a particular booking.
- **Connection connection:** A connection object obtained from `ConnectSQL.getConnection()` to run SQL commands.
- **SQLStatements sql:** An instance of class `SQLStatements`. The class contains the SQL statements required to perform all the database operations.

Methods :

1. `setBooking(Booking booking)`

A setter method to pass a Booking object to the Payments class, decoupling the two classes and allowing for flexible interaction.

```
public void setBooking (Booking booking) {  
    this.booking = booking;  
}
```

2. `calculateTotalRent()`

A private method that calculates the total rental cost. It first ensures that the `addBooking()` method from the Booking class is executed, then retrieves the car's cost and the number of rental days. The method applies a discount if the rental period is two months or longer.

```

private void calculateTotalRent () throws SQLException {
    // make sure the addbooking method run before calculation part.
    booking.addBooking();
    double cost = booking.getCost();
    days = booking.getDays();

    double costPerDay = cost / 30;

    int months = calculateMonths();
    int remainingDays = calculateDays();

    if (months >= 2) {
        double discount = 12.5;
        double rentCost = (cost * months) + (costPerDay * remainingDays) ;
        totalCost = Math.round(rentCost * (100 - discount)/100);
        System.out.println("Your payment amount is: " + totalCost);
    } else {
        double rentCost = (cost * months) + (costPerDay * remainingDays);
        System.out.println("Your payment amount is: " + rentCost);
    }

}

```

3. calculateMonths(): A private helper method that calculates the number of full months based on the total rental days.

```

private int calculateMonths() {
    int temporary = days;
    int months = 0;
    while (temporary >= 30) {
        temporary -= 30;
        months += 1;
    }

    return months;
}

```

4. calculateDays(): A private helper method that calculates the remaining days after full months are accounted for.

```
private int calculateDays() {  
    int remainingDays = days - (calculateMonths()*30);  
    return remainingDays;  
}
```

5. updatePaymentsDetails(): The main method that updates the payment details in the database. It first calculates the total rent and then inserts the payment details into the database, using the booking ID retrieved from the Booking class.

```
public void updatePaymentsDetails() throws SQLException {  
    calculateTotalRent();  
  
    Statement statement = connection.createStatement();  
    statement.executeUpdate(sql.createPaymentDetails());  
  
    int bookingId = booking.getLastBookingID();  
  
    PreparedStatement preparedStatement = connection.prepareStatement(sql.setPaymentDetails());  
    setPaymentDetailsTable(preparedStatement, bookingId, totalCost);  
}
```

6. setPaymentDetailsTable(.): private method that sets the payment details in the database using a prepared statement.

```
private void setPaymentDetailsTable(PreparedStatement preparedStatement, int bookingId, double totalCost) throws SQLException {  
    preparedStatement.setInt( parameterIndex: 1, bookingId);  
    preparedStatement.setDouble( parameterIndex: 2, totalCost);  
    preparedStatement.executeUpdate();  
    preparedStatement.close();  
}
```

7. getTotalCost(): A public method that returns the total calculated cost of the rental.

```
public double getTotalCost () {  
    return totalCost;  
}
```

In class, it catches and throws the SQL Exceptions as runtime exceptions wherever appropriate, ensuring that any problems related to database interaction are dealt with properly. It is decoupled from the class Bookings since it gets the Booking object via setter methods; thus, it is easier to maintain and test.

Calculation Logic

Below is the payment calculation logic in the class Payments:

- The monthly rental cost is to be divided by 30 and cost per day is calculated.
- Discount is setted as 12.5% of the total cost if the renting time is 2 months or more.
- The total cost is then calculated by its number of full months and remaining days, with consideration for a discount if available.

Report Class

This class will be used to generate transaction reports for the RentACar system. This report will include details such as the brand, model, customer name, amount paid, and due date of the car. It logs into the database, retrieves the information, and formats it into a readable overview of the rental records. This class uses a specific formatting string in the generateReport() method to ensure that the output is neatly aligned and easy to read.

Below are the primary components of the class:

- Connection connection: A connection object obtained from ConnectSQL.getConnection() for executing SQL commands.
- SQLstatements sql: An instance of the SQLstatements class, which provides SQL queries needed for report generation.
- Scanner scanner: A scanner object for taking user input to determine if the report should be generated.
- String name: Stores the name of the customer.
- double payment: Stores the payment amount.
- String number: Stores the contact number or identifier related to the transaction.
- String dueDate: Stores the due date of the rental.
- String brand: Stores the car's brand name.
- String model: Stores the car's model name.

Methods

generateReport(): The main method that triggers the report on all activities according to the request by the user. The method fetches all records of rentals by querying the database and formats it in tabular form to produce the table on the console.

```
public void generateReport() throws SQLException {

    System.out.println("Do you want to check Transaction Report: (yes/no)");
    String userInput = scanner.next();

    if (userInput.equalsIgnoreCase("yes")) {
        Statement statement = connection.createStatement();
        ResultSet resultSet = statement.executeQuery(sql.generateReport());

        System.out.println("----- Car Rental Records -----");
        System.out.printf("%-20s | %-20s | %-20s | %-15s | %-10s | %-10s\n",
            "Car Brand", "Car Model", "Customer Name", "Number", "Payment Amount", "Due Date");
        System.out.println("-----");

        while (resultSet.next()) {
            brand = resultSet.getString(columnLabel: "Car_Brand");

            model = resultSet.getString(columnLabel: "Car_Model");

            name = resultSet.getString(columnLabel: "Customer_Name");

            payment = resultSet.getDouble(columnLabel: "Payment_Amount");
            number = resultSet.getString(columnLabel: "number");
            dueDate = resultSet.getString(columnLabel: "Due_date");

            // Print data rows (Make sure variables are properly assigned from result set or source)
            System.out.printf("%-20s | %-20s | %-20s | %-15s | %-10.2f | %-10s\n",
                brand, model, name, number, payment, dueDate);
        }
    }
}
```


StopProgrammeException Class

The StopProgrammeException class is a specifically designed exception for a certain type of error that may happen during its execution. Since this custom exception extends the class Exception, it can thus be thrown and caught like any other exception, providing a way of showing an error unique to the logic of the application for better control on error handling.

This depicts Inheritance in OOP. This class is inherited from the standard Exception class. Due to this inheritance, StopProgrammeException will be treated as a checked exception. As such, it has to be explicitly handled by the code that might throw it in a try-catch block or declared in the throws clause of the method.

```
public class StopProgrammeException extends Exception {  
    4 usages  — banuka2001  
    public StopProgrammeException(String message) {  
        super(message);  
    }  
}
```

Conclusion

The system of RentACar is a fully-fledged application capable of handling the most complex operations in terms of car rentals, customer interactions, and the processing of payments. This system will, therefore, be smartly organized for its prime functioning on account of Object-Oriented Programming, where multiple classes are at work managing aspects pertaining to the renting process. This architecture would aid in giving programs modularity, hence facilitating maintenance, extension, and debugging.

Run the program using .jar file

I have added a .jar file. It can be executed using following steps.

Open Command Prompt.

Navigate to the Directory Containing the .jar File:

using: **cd path/.../...**

Run following command.

java -jar RentACar.jar

NOTE: Full code is added as another pdf file.

END