Campus Car Parking Project.

Name: KV Sreekar
Roll No: A00336013
Submission Date: 29/11/2025

College: Technology university Shannon Athlone
Course: Software Design with Cloud Native Computing

1. Introduction

The Car Park Project is a Java 21 console-based application designed to model and manage a simple parking system for a college campus. The system allows different types of vehicles to be parked, listed, filtered, and exited by calculating the total parking fee based on vehicle type and parking duration.

The project demonstrates Object-Oriented Programming concepts and modern Java 21 language features including sealed interfaces, records, lambdas, switch expressions with pattern matching, defensive copying, custom immutable types, method references, and more.

The aim is to provide a clean, structured example that clearly demonstrates strong software design practices suitable for cloud-native systems.

2. Implemented User Stories

• US1 – Park a vehicle and generate a parking ticket.

• US2 – List all currently parked vehicles.

• US3 – Filter vehicles by type (e.g., ElectricCar only).

• US4 – Exit a vehicle and calculate its total bill.

• US5 – Handle car park full errors and invalid ticket scenarios.

These user stories collectively simulate real-world parking lot behavior while showcasing fundamental and advanced Java concepts.

3. System Design Overview

The system consists of several classes that reflect real-world entities:

• Vehicle hierarchy using a sealed interface: Car, ElectricCar, Motorbike.

• ParkingLot manages a Vehicle[] array for bays, active ParkingTicket objects, and billing logic.

• ParkingTicket is a Java record storing essential ticket metadata.

• Money is an immutable value class representing charges.

• SimpleTariffCalculator calculates fees using switch expressions with pattern matching.

• CarParkApp drives the program through a demonstration scenario.

The design is modular, readable, and demonstrates modern Java programming standards.

4. Core OOP Concepts Demonstrated

Encapsulation:

Private fields with public getters in Car, ParkingLot, Money ensure data protection.

Inheritance & Polymorphism:

ElectricCar extends Car, while Motorbike implements Vehicle. ParkingLot stores all vehicles polymorphically.

Method Overloading:

ParkingLot contains two parkVehicle methods—one auto-timed, one with a custom timestamp.

this() vs this.:

Car demonstrates constructor chaining using this(), and field referencing using this.fieldName.

super() vs super.:

ElectricCar calls super() to initialize Car properties and uses super.toString() to extend the parent method.

Varargs:

logEvents(String... events) supports multiple message inputs.

5. Modern Java 21 Features

Sealed Interface:

Vehicle restricts subclassing to Car and Motorbike, improving code safety and clarity.

Records:

ParkingTicket is implemented as a Java record with built-in immutability.

Custom Immutable Value Type:

Money is a manually created immutable class with final fields and no setters.

Switch Expressions + Pattern Matching:

SimpleTariffCalculator uses modern switch expressions to differentiate Car, ElectricCar, and Motorbike.

Lambdas and Predicate:

parkedEVs = findVehicles(v -> v instanceof ElectricCar);

A functional, concise way to filter collections.

Method References:

vehicles.forEach(System.out::println) simplifies output.

LVTI (var):

var now = vehicle.defaultEntryTime() demonstrates cleaner local variable declarations.

Defensive Copying:

getOccupiedVehicles() never exposes internal arrays directly, protecting internal state.

Call-by-Value Proof:

tryToClearArray confirms Java always uses call-by-value semantics.

6. Demonstration Flow (Program Execution)

1. A Car, ElectricCar, and Motorbike are created.

2. All vehicles are parked and linked with ParkingTicket objects.

3. The ParkingLot lists all currently parked vehicles.

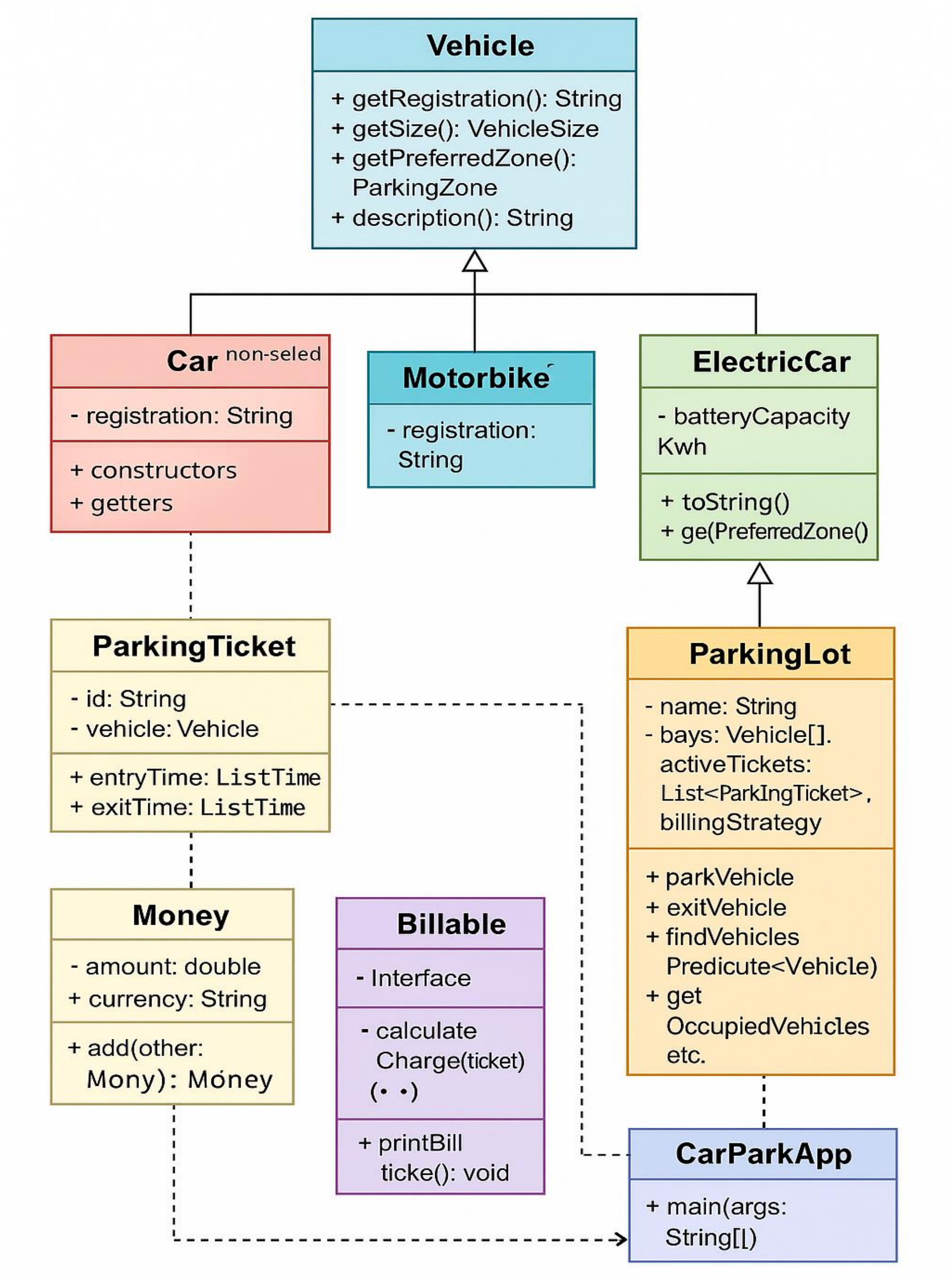
4. A lambda filters only ElectricCar instances.

5. The ElectricCar exits, generating a detailed billing output.

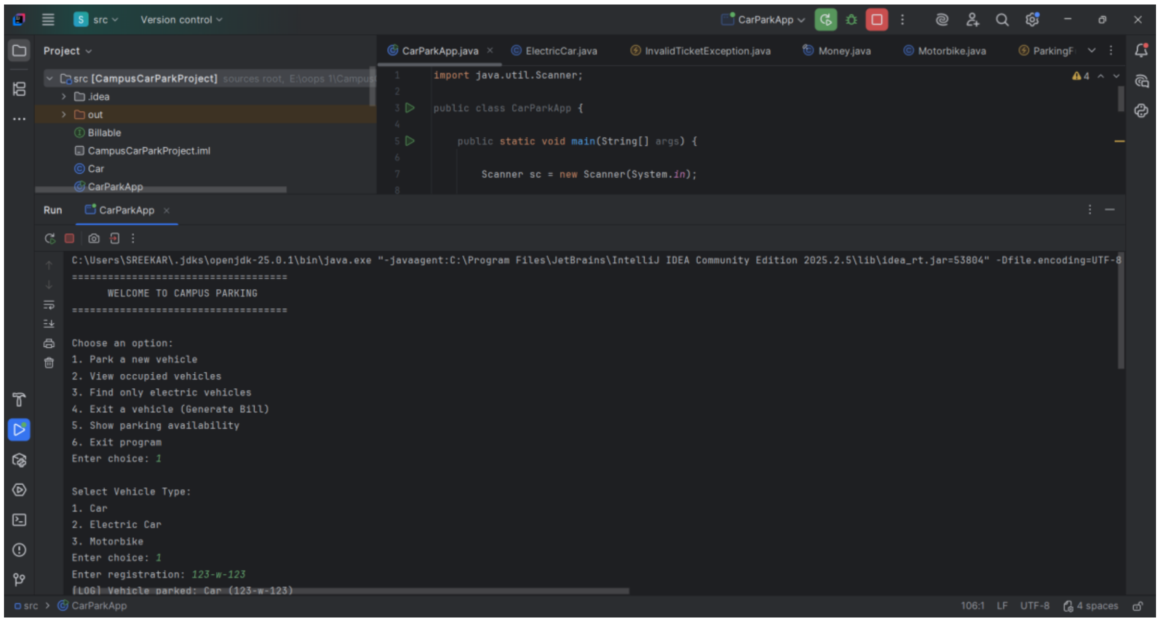
6. The final bay layout is displayed.

This flow accurately reflects real-world parking management behavior

7. UML Structure (Text Description)

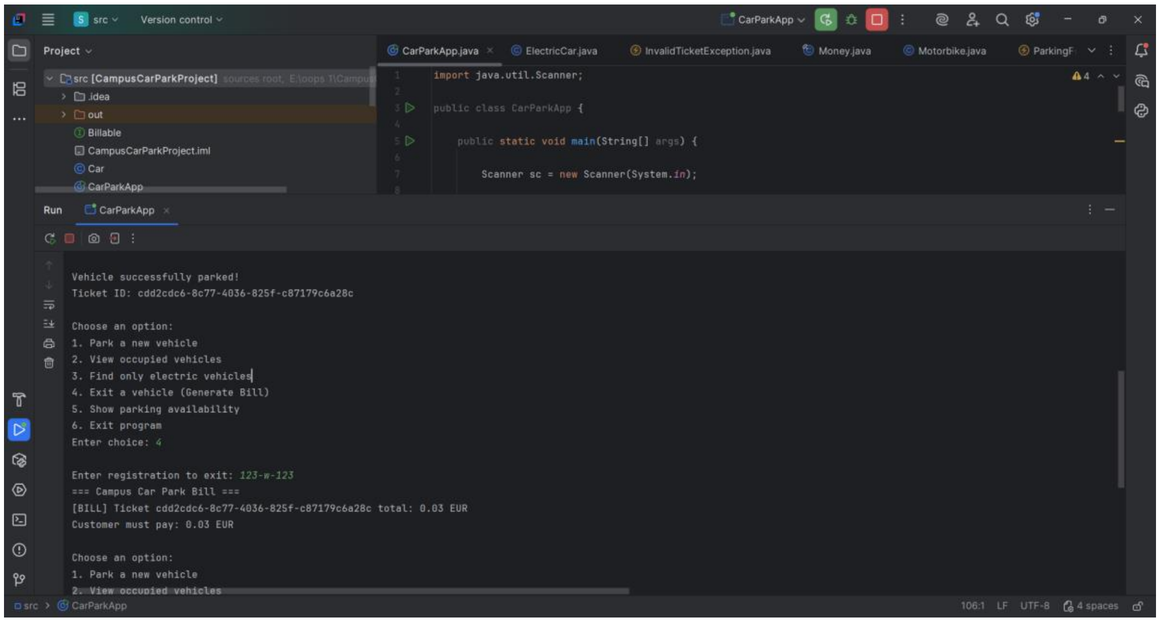


Output:



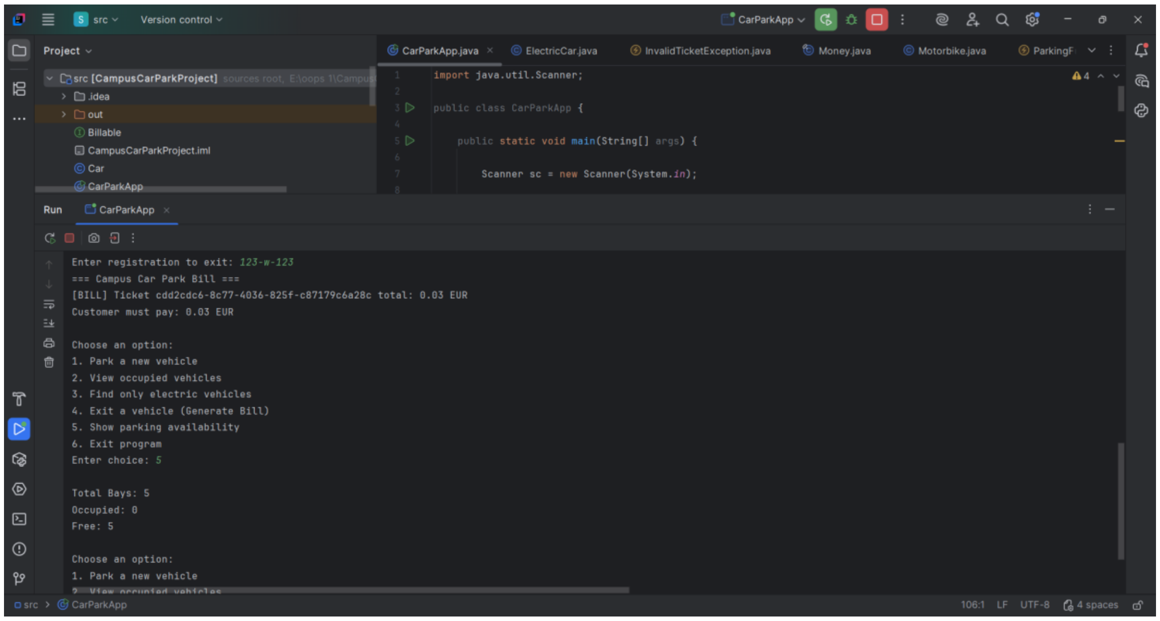
```java
import java.util.Scanner;

public class CarParkApp {

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);
```

```
C:\Users\SREEKAR\.jdks\openjdk-25.0.1\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2025.2.5\lib\idea_rt.jar=53804" -Dfile.encoding=UTF-8
====================================
      WELCOME TO CAMPUS PARKING
====================================

Choose an option:
1. Park a new vehicle
2. View occupied vehicles
3. Find only electric vehicles
4. Exit a vehicle (Generate Bill)
5. Show parking availability
6. Exit program
Enter choice: 1

Select Vehicle Type:
1. Car
2. Electric Car
3. Motorbike
Enter choice: 1
Enter registration: 123-w-123
[LOG] Vehicle parked: Car (123-w-123)
```



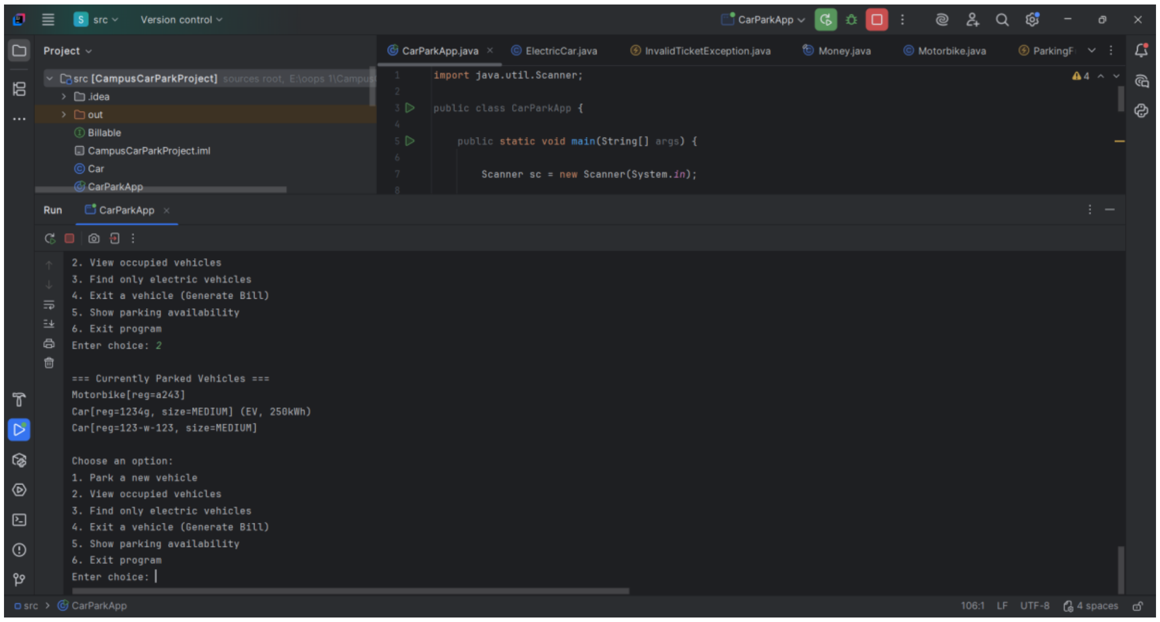
```java
import java.util.Scanner;

public class CarParkApp {

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);
```

```

Vehicle successfully parked!
Ticket ID: cdd2cdc6-8c77-4036-825f-c87179c6a28c

Choose an option:
1. Park a new vehicle
2. View occupied vehicles
3. Find only electric vehicles
4. Exit a vehicle (Generate Bill)
5. Show parking availability
6. Exit program
Enter choice: 4

Enter registration to exit: 123-w-123
=== Campus Car Park Bill ===
[BILL] Ticket cdd2cdc6-8c77-4036-825f-c87179c6a28c total: 0.03 EUR
Customer must pay: 0.03 EUR

Choose an option:
1. Park a new vehicle
2. View occupied vehicles
```



```java
import java.util.Scanner;

public class CarParkApp {

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);
```

```
Enter registration to exit: 123-w-123
=== Campus Car Park Bill ===
[BILL] Ticket cdd2cdc6-8c77-4036-825f-c87179c6a28c total: 0.03 EUR
Customer must pay: 0.03 EUR

Choose an option:
1. Park a new vehicle
2. View occupied vehicles
3. Find only electric vehicles
4. Exit a vehicle (Generate Bill)
5. Show parking availability
6. Exit program
Enter choice: 5

Total Bays: 5
Occupied: 0
Free: 5

Choose an option:
1. Park a new vehicle
2. View occupied vehicles
```



```java
import java.util.Scanner;

public class CarParkApp {

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);
```

```
2. View occupied vehicles
3. Find only electric vehicles
4. Exit a vehicle (Generate Bill)
5. Show parking availability
6. Exit program
Enter choice: 2

=== Currently Parked Vehicles ===
Motorbike[reg=a243]
Car[reg=1234g, size=MEDIUM] (EV, 250kWh)
Car[reg=123-w-123, size=MEDIUM]

Choose an option:
1. Park a new vehicle
2. View occupied vehicles
3. Find only electric vehicles
4. Exit a vehicle (Generate Bill)
5. Show parking availability
6. Exit program
Enter choice: 
```

8. Limitations & Future Enhancements

Current Limitations:

• Console-only; no GUI or web interface.

• No persistent storage (data resets after each run).

• Vehicles are created inside code, not user input.

• Basic tariff system.

Future Improvements:

• Add a command-line or menu-driven interface.

• Add database or file storage for tickets.

• Add reporting features (daily income, busiest hours).

• Add more vehicle classes and more advanced tariff rules.

9. Conclusion

The Car Park Project demonstrates the effective use of both foundational and advanced Java 21 features while maintaining clean design, logical flow, and real-world applicability. It highlights sealed interfaces, records, inheritance, lambdas, switch expressions, defensive copying, and other modern practices. This medium-sized project balances clarity and complexity, making it suitable for coursework in software design and cloud-native computing.