Transportation Fleet Management System (Assignment 2)

1. Overview

This second assignment extends the Transportation Fleet Management System developed in Assignment 1.

The original system focused on object-oriented principles such as inheritance, interfaces, abstraction, and polymorphism.

In this assignment, the system is refactored into a data-driven application using the Java Collections Framework and CSV-based file persistence.

This upgraded version now supports:

- Dynamic addition and removal of vehicles
- Storage using ArrayList, HashSet, and TreeSet
- Guaranteed uniqueness of vehicle IDs
- Automatic sorting and ordering using Comparable and Comparator
- CSV reading and writing for persistence
- Data analysis utilities such as fastest/slowest vehicle and model-based summaries

The purpose is to simulate how real systems evolve from static OOP programs into scalable, persistent applications.

2. How Collections Are Used in the System

ArrayList (Primary storage for vehicles)

The FleetManager stores vehicles in an ArrayList<Vehicle> because:

- It allows dynamic resizing
- It supports fast iteration for reporting
- It preserves insertion order, which is convenient for CLI display

HashSet (Uniqueness enforcement)

A HashSet<String> is used to maintain **unique vehicle IDs or unique model names**.

This ensures no duplicates are added to the fleet.

HashSet is justified because:

- Set semantics prevent duplicate entries
- O(1) average insertion and lookup time

TreeSet (Alphabetical or attribute-based ordering)

A TreeSet<String> or TreeSet<Vehicle> (depending on the design) is used to maintain sorted order automatically.
Sorting can be based on:

- Model name (case-insensitive alphabetical order)
- Vehicle attributes via Comparators

TreeSet is appropriate because the assignment requires maintaining sorted views without repeatedly calling Collections.sort().

Comparable and Comparator

Two types of ordering are demonstrated:

- Natural ordering via Comparable<Vehicle> (e.g., by max speed or fuel efficiency)
- Custom ordering via Comparator<Vehicle> (e.g., alphabetical model sort, speed sort, efficiency sort)

This satisfies the assignment's requirement for sorting and ordering mechanisms.

3. File I/O and Persistence Explanation

The system uses CSV files to store fleet data across program executions.

Saving to CSV

- The FleetManager iterates through all vehicles and converts each into a comma-separated representation.
- The first field indicates the vehicle type (Car, Truck, Bus, Airplane, CargoShip).
- Remaining fields contain the attributes required to reconstruct the object.
- Standard Java I/O classes (FileWriter, BufferedWriter, etc.) are used.

Loading from CSV

- The system reads the file line by line.
- Each line is passed to VehicleFactory, which parses the fields and constructs the correct subclass.
- Malformed or missing files are handled using try–catch blocks.
- Vehicles are added back into the ArrayList, and uniqueness is re-checked using sets.

This ensures the system is fully persistent and continues to function correctly between runs.

4. Program Features Implemented

The CLI demonstrates all core functionalities required by Assignment 2:

Vehicle Management

- Add vehicle
- Remove vehicle
- Display full fleet
- Search vehicles by type (land, air, water categories or specific class)

Journey Simulation

- Start journeys for all vehicles
- Compute fuel consumption where applicable
- Update mileage dynamically

Maintenance

- Perform maintenance for applicable vehicles (e.g., cars, trucks, buses)

Sorting and Ordering

- Sort by maximum speed

- Sort by fuel efficiency
- Sort alphabetically by model name (case-insensitive)
- Automatic ordering via TreeSet or Comparators

Analysis Utilities

- Identify the fastest vehicle
- Identify the slowest vehicle
- Display all distinct model names
- Compute simple summaries such as vehicle counts

CSV Persistence

- Save fleet to a CSV file
- Load fleet from a CSV file
- Validate CSV structure against factory expectations

5. Compilation Instructions

1. Open a terminal inside the project directory.
2. Compile all files using:

javac vehicles/*.java exceptions/*.java VehicleFactory.java FleetManager.java Main.java

3. Ensure there are no compilation errors.

6. Running the Program

Execute the program using:

java Main

The CLI menu will appear, offering options such as:

1. Add Vehicle
2. Remove Vehicle
3. Start Journey
4. Refuel All
5. Perform Maintenance
6. View Fleet
7. Save Fleet to File
8. Load Fleet from File
9. Search by Type
10. Fastest / Slowest Vehicle
11. Exit

Users follow the prompts to interact with the fleet.

7. CSV File Format Description

Each line in the CSV corresponds to one vehicle.
The first field identifies the vehicle type, followed by fields required to reconstruct that type.

Example:

```
Car,C1,Honda,180.0,20000.0,4,50.0,15.0
Truck,T1,Volvo,90.0,10000.0,6,200.0,8.0,5000,1000
Bus,B1,Mercedes,80.0,3000.0,6,100.0,10.0,40,20
Airplane,A1,Boeing,800.0,1100.0,110.0,50,5000,3000,1000.0,2000.0
CargoShip,S1,Maersk,40.0,2000.0,true,50000,20000,5000
```

The order of fields must match the expectations of VehicleFactory.

8. Sample Program Output

```
===== Fleet Manager CLI =====
1. Add Vehicle
2. Remove Vehicle
3. Start Journey
4. Refuel All
5. Perform Maintenance
6. View Fleet
7. Save Fleet to File
8. Load Fleet from File
9. Search by Type
10. Fastest / Slowest Vehicle
11. Exit
Choose an option: 1
Vehicle type: Car
Car added successfully.

Choose an option: 6
--- Current Fleet ---
ID: C1, Model: Honda, Max Speed: 180.0, Mileage: 20000.0
Fuel Efficiency: 15.0 km/l

Choose an option: 3
Enter distance to move all vehicles: 100
Driving on road for 100.0 km.
Total fuel consumed: 6.67

Choose an option: 10
Fastest vehicle: C1 (Honda)
Slowest vehicle: T1 (Volvo)

Choose an option: 7
Fleet saved successfully to fleet_data.csv

Choose an option: 11
Exiting...
```