# Transportation Fleet Management System

## Key Data Structures and Features

This system leverages core Java collections and comparison logic to manage and analyze the fleet data:

- **ArrayList**: The primary data structure for the fleet is private List<Vehicle> fleet = new ArrayList<>();. An ArrayList is used to dynamically store all Vehicle objects, allowing for easy addition, removal, and iteration.
- **Collections.sort()**: The system provides multiple sorting options by leveraging Collections.sort():
  - **Natural Ordering**: sortFleetByEfficiency() calls Collections.sort(fleet), which relies on the Vehicle class implementing Comparable<Vehicle> and overriding the compareTo method to sort by fuel efficiency by default.
  - **Custom Comparator**: sortFleetBySpeed() and sortFleetByModel() use custom Comparator objects (defined in getComparatorBySpeed() and getComparatorByModel()) to provide alternative sorting logic based on different vehicle attributes.
- **Set (via TreeSet)**: The getDistinctModelNames() method uses a Set<String> modelNames = new TreeSet<>();. This efficiently creates a unique list of all vehicle models in the fleet. A TreeSet is chosen specifically to ensure the list is automatically sorted in alphabetical order.
- **Exception Handling**: The program uses custom exceptions (InsufficientFuelException, InvalidOperationException, OverloadException) to manage runtime errors like attempting to move without fuel or add a vehicle with a duplicate ID.

## Directory Structure

- **exceptions/**
  - InsufficientFuelException.java
  - InvalidOperationException.java
  - OverloadException.java
- **interfaces/**
  - CargoCarrier.java
  - FuelConsumable.java

    - Maintainable.java
    - PassengerCarrier.java
- **vehicles/**
    - Vehicle.java (Abstract base class)
    - AirVehicle.java (Abstract)
    - LandVehicle.java (Abstract)
    - WaterVehicle.java (Abstract)
    - Airplane.java
    - Bus.java
    - Car.java
    - CargoShip.java
    - Truck.java
- **manager/**
    - FleetManager.java (Core logic class)
- **mainapp/**
    - Main.java (Entry point and CLI)
- fleet.csv (Sample CSV for persistence)

---

## Running the Program

**Compile:** Navigate to the src directory and compile all Java files, outputting them to a bin or out folder:
Bash
```
javac -d ../out */*.java */*/*.java
```

1.

**Run:** From the out directory (or using the correct classpath from the root):
Bash
```
java mainapp.Main
```

2.
3. **Startup:** The program starts by:
    1. Creating one vehicle of each type (Car, Truck, Bus, Airplane, CargoShip) and adding them to the fleet manager.
    2. Launching the interactive command-line menu.

---

## Using the CLI

Enter numeric options (1-16) to navigate the menu. Follow the prompts for inputs (e.g., vehicle type, ID, maxSpeed).

## Menu Options

1. **Add Vehicle**: Prompt for vehicle type and all required properties.
2. **Remove Vehicle**: Remove a vehicle by its unique ID.
3. **Start Journey**: Simulate a journey for a **single vehicle** (prompts for ID and distance).
4. **Refuel All**: Add a specified amount of fuel to all FuelConsumable vehicles.
5. **Perform Maintenance**: Perform maintenance on all vehicles that require it.
6. **Generate Report**: Display fleet summary statistics and a full list of all vehicles.
7. **Save Fleet**: Save the current fleet state to a CSV file.
8. **Load Fleet**: Clear the current fleet and load a new one from a CSV file.
9. **Search by Type**: List all vehicles of a specific class (e.g., Car) or interface (e.g., CargoCarrier).
10. **List Vehicles Needing Maintenance**: Show only vehicles that flag needsMaintenance() as true.
11. **Sort Fleet by Speed**: Sorts the fleet from slowest to fastest.
12. **Sort Fleet by Model**: Sorts the fleet alphabetically by model name (A-Z).
13. **Sort Fleet by Efficiency**: Sorts the fleet from least to most fuel-efficient.
14. **Show Fastest & Slowest Vehicle**: Displays the vehicles with the max and min speed.
15. **List Distinct Vehicle Models**: Shows a unique, alphabetical list of all model names.
16. **Exit**: Quit the program.

## Notes

- Input validation is in place to catch non-numeric inputs where numbers are expected.
- Errors (e.g., duplicate IDs, insufficient fuel, invalid operations) are caught and displayed as user-friendly messages.