Transportation Fleet Management System Object-Oriented Programming Assignment

Sambuddho

Due: **September 15, 2025 (2359hrs)**

1 Overview

In this assignment, you will design and implement a Java program simulating a transportation fleet management system for a logistics company. The system manages a diverse fleet of vehicles, handling operations like route planning, cargo and passenger management, maintenance scheduling, persistent storage, and a command-line interface (CLI) for user interaction. This assignment emphasizes core OOPs concepts, requiring a multi-level class hierarchy with inheritance, polymorphism through dynamic method dispatch, abstract classes for shared structure, and interfaces for modular behaviors like fuel management, cargo handling, and maintenance tracking.

2 Problem Statement

A logistics company manages a fleet of vehicles across land, air, and water, including cars, trucks, buses, airplanes, and cargo ships. Each vehicle has properties like ID, model, max speed, and type-specific attributes. The system must support:

- Creating and managing vehicles with varied behaviors (e.g., moving, fueling, loading cargo/passengers).
- A fleet manager to add/remove vehicles, simulate journeys, calculate efficiencies, and generate reports.
- Persistent storage: Save/load fleet data to/from files.
- A CLI for users to add vehicles, perform operations, and view reports.
- Route planning: Simulate journeys with distance, time, and fuel consumption calculations.
- Maintenance: Schedule and track vehicle repairs.
- Error handling: Custom exceptions for invalid operations (e.g., overloading cargo).

The system must demonstrate OOP principles by enabling polymorphic operations (e.g., "move all vehicles" executes type-specific movements) and using interfaces for modular features.

3 Requirements

3.1 Abstract Class: Vehicle

- **Role**: Root of the hierarchy.
- **Properties**: `private String id`, `private String model`, `private double maxSpeed`, `private double currentMileage` (tracks total distance traveled).
- **Constructor**: Initialize all properties; validate ID (non-empty).
- **Abstract Methods**:
  - `abstract void move(double distance)`: Updates mileage, prints type-specific movement; throws `InvalidOperationException` if distance < 0.
  - `abstract double calculateFuelEfficiency()`: Returns km per liter (or 0 for non-fuel vehicles).
  - `abstract double estimateJourneyTime(double distance)`: Returns time in hours (distance / maxSpeed, adjusted by type).
- **Concrete Methods**:
  - `void displayInfo()`: Prints all properties in a formatted string.
  - `double getCurrentMileage()`: Getter for mileage.
  - `String getId()`: Getter for ID.

3.2 Abstract Class: LandVehicle extends Vehicle

- **Property**: `private int numWheels`.
- **Constructor**: Call super, initialize `numWheels`.
- **Override**: `estimateJourneyTime(double distance)`: Add 10% time for traffic (base time $\times$ 1.1).
- Keep `move` and `calculateFuelEfficiency` abstract.

3.3 Abstract Class: AirVehicle extends Vehicle

- **Property**: `private double maxAltitude`.
- **Constructor**: Call super, initialize `maxAltitude`.
- **Override**: `estimateJourneyTime(double distance)`: Reduce 5% time for direct paths (base time $\times$ 0.95).

3.4 Abstract Class: WaterVehicle extends Vehicle

- **Property**: `private boolean hasSail` (affects fuel efficiency).
- **Constructor**: Call super, initialize `hasSail`.
- **Override**: `estimateJourneyTime(double distance)`: Add 15% time for currents (base time $\times$ 1.15).

3.5 Interfaces

- **FuelConsumable**:
  - `void refuel(double amount)`: Adds fuel; throws `InvalidOperationException` if amount ≤ 0.
  - `double getFuelLevel()`: Returns current fuel level.
  - `double consumeFuel(double distance)`: Reduces fuel based on efficiency; returns consumed amount; throws `InsufficientFuelException` if not enough fuel.
- **CargoCarrier**:
  - `void loadCargo(double weight)`: Loads if $\leq$ capacity; throws `OverloadException` if exceeded.
  - `void unloadCargo(double weight)`: Unloads; throws `InvalidOperationException` if weight $>$ current cargo.
  - `double getCargoCapacity()`: Returns max capacity.
  - `double getCurrentCargo()`: Returns current cargo.
- **PassengerCarrier**:
  - `void boardPassengers(int count)`: Boards if $\leq$ capacity; throws `OverloadException`.
  - `void disembarkPassengers(int count)`: Disembarks; throws `InvalidOperationExceptio` if count $>$ current passengers.
  - `int getPassengerCapacity()`: Returns max capacity.
  - `int getCurrentPassengers()`: Returns current passengers.
- **Maintainable**:
  - `void scheduleMaintenance()`: Sets maintenance flag.
  - `boolean needsMaintenance()`: True if mileage > 10000 km.
  - `void performMaintenance()`: Resets flag, prints message.

3.6 Concrete Classes

- **Car extends LandVehicle implements FuelConsumable, PassengerCarrier, Maintainable**:
  - **Properties**: `private double fuelLevel` (init 0), `private int passengerCapacity` (5), `private int currentPassengers`, `private boolean maintenanceNeeded`.
  - **Override** `move(double distance)`: "Driving on road...", consume fuel, update mileage; check fuel sufficiency.
  - `calculateFuelEfficiency()`: 15.0 km/l.
  - Implement all interface methods.
- **Truck extends LandVehicle implements FuelConsumable, CargoCarrier, Maintainable**:

  - **Properties**: `fuelLevel`, `private double cargoCapacity` (5000 kg), `private double currentCargo`, `maintenanceNeeded`.
  - **Override** `move`: "Hauling cargo...", adjust fuel consumption if loaded ($> 50\%$ capacity reduces efficiency by 10%).
  - `calculateFuelEfficiency()`: 8.0 km/l (adjusted for cargo).

- **Bus extends LandVehicle implements FuelConsumable, PassengerCarrier, CargoCarrier, Maintainable**:
  - **Properties**: `fuelLevel`, `passengerCapacity` (50), `currentPassengers`, `cargoCapacity` (500 kg), `currentCargo`, `maintenanceNeeded`.
  - **Override** `move`: "Transporting passengers and cargo...".
  - `calculateFuelEfficiency()`: 10.0 km/l.

- **Airplane extends AirVehicle implements FuelConsumable, PassengerCarrier, CargoCarrier, Maintainable**:
  - **Properties**: `fuelLevel`, `passengerCapacity` (200), `currentPassengers`, `cargoCapacity` (10000 kg), `currentCargo`, `maintenanceNeeded`.
  - **Override** `move`: "Flying at [maxAltitude]...".
  - `calculateFuelEfficiency()`: 5.0 km/l.

- **CargoShip extends WaterVehicle implements CargoCarrier, Maintainable** (FuelConsumable if `hasSail=false`):
  - **Properties**: `cargoCapacity` (50000 kg), `currentCargo`, `maintenanceNeeded`, `fuelLevel` (if fueled).
  - **Override** `move`: "Sailing with cargo...".
  - `calculateFuelEfficiency()`: 4.0 km/l if fueled, else 0.

3.7 Custom Exceptions

- `OverloadException extends Exception`: For cargo/passenger overload.
- `InvalidOperationException extends Exception`: For negative distances, invalid unloads, etc.
- `InsufficientFuelException extends Exception`: For insufficient fuel in `move()`.

3.8 FleetManager Class

- **Properties**: `private List<Vehicle> fleet` (use `ArrayList<Vehicle>`).
- **Methods** (leverage polymorphism):
  - `void addVehicle(Vehicle v)`: Check ID uniqueness; throw `InvalidOperationException` if duplicate.
  - `void removeVehicle(String id)`: Remove by ID; throw `InvalidOperationException` if not found.

- `void startAllJourneys(double distance)`: Call `move(distance)` on each; handle exceptions.
- `double getTotalFuelConsumption(double distance)`: Sum `consumeFuel(distance)` for `FuelConsumable` vehicles.
- `void maintainAll()`: Call `performMaintenance()` if `needsMaintenance()`.
- `List<Vehicle> searchByType(Class<?> type)`: Return vehicles `instanceof` type (e.g., `Car.class`, `FuelConsumable.class`).
- `void sortFleetByEfficiency()`: Implement `Comparable<Vehicle>` in `Vehicle` (compare by `calculateFuelEfficiency()`), use `Collections.sort(fleet)`.
- `String generateReport()`: Summary of fleet stats (total vehicles, count by type, average efficiency, total mileage, maintenance status).
- `List<Vehicle> getVehiclesNeedingMaintenance()`: Filter vehicles where `needsMaintenance()` is true.

3.9 Persistence

- `void saveToFile(String filename)`: Save fleet to CSV (e.g., "Car,V001,Toyota,120.0,4,50.0,5,0" for a `Car`).
- `void loadFromFile(String filename)`: Load from CSV, recreate vehicles (use factory method for type parsing).
- Handle `IOExceptions` with user-friendly messages.

3.10 Command-Line Interface (CLI)

- In `Main` class, implement a menu-driven CLI using `Scanner`:
- **Options**: Add vehicle (prompt for type, properties), remove vehicle, start journey (prompt for distance), refuel all (prompt for amount), maintain all, generate report, save/load fleet, search by type, list maintenance needs, exit.
- Use a loop to display menu and process inputs.
- Validate inputs (e.g., numeric inputs for distances, valid vehicle types).
- **Example Menu**:

```
1. Add Vehicle
2. Remove Vehicle
3. Start Journey
4. Refuel All
5. Perform Maintenance
6. Generate Report
7. Save Fleet
8. Load Fleet
9. Search by Type
10. List Vehicles Needing Maintenance
11. Exit
```

3.11 Main Class

- **Demo**: Create sample vehicles (one of each type), add to fleet, simulate a journey (e.g., 100 km), generate report, save to file.
- Launch CLI for user interaction.
- Include sample CSV file for testing load/save.

4 Implementation Guidelines

- **Encapsulation**: Use private fields, public getters/setters where needed.
- **Comparable**: Implement in `Vehicle` for sorting by fuel efficiency.
- **Exception Handling**: Use try-catch, propagate exceptions, provide user-friendly messages in CLI.
- **File I/O**: Use CSV format; ensure robust parsing (e.g., handle malformed input).
- **Factory Method**: Recommended for creating vehicles from CLI/file input (e.g., `Vehicle createVehicle(String type, String[] data)`).
- **No External Libraries**: Use standard Java (`java.util`, `java.io`).
- **Validation**: Check for negative values, null inputs, duplicate IDs.
- **Code Structure**: Organize classes in packages (e.g., `vehicles`, `exceptions`, `fleet`).

5 Submission

- **ZIP File**: Include all `.java` files, sample CSV file.
- **UML Diagram**: PDF/PNG (hand-drawn or tool-generated) showing class hierarchy, interfaces, and relationships.
- **README.txt**:
  - Explain how your code demonstrates inheritance, polymorphism, abstract classes, and interfaces.
  - Provide clear instructions to compile (e.g., `javac *.java`), run (e.g., `java Main`), and test persistence with the sample CSV.
  - Describe how to use the CLI and demo features (e.g., add vehicles, simulate journey, save/load).
  - Include a brief walkthrough of running the demo and expected output.

6 Grading Criteria

- **Inheritance (15%)**: Correct multi-level hierarchy with proper overriding.
- **Polymorphism (15%)**: Extensive use in fleet operations (move, efficiency, etc.).
- **Abstract Classes (15%)**: Proper use in `Vehicle`, `LandVehicle`, etc.
- **Interfaces (15%)**: Correct implementation of multiple interfaces.
- **Functionality (20%)**: CLI, persistence, journey simulation, maintenance, etc., work as specified.
- **Exception Handling (10%)**: Robust error handling with custom exceptions.
- **Documentation/README (10%)**: Clear, complete, with accurate compile/run instructions showing appropriate test cases.

Submit by September 15, 2025 (2359hrs.). No extensions! Start early and use office hours for clarification. Good luck!