AP-M2025 Assignment 2

Fleet Management with Collections and File Handling

Due Date: November 1, 2025 (23:59 hrs)

1. Overview

This second assignment extends your *Transportation Fleet Management System* developed in Assignment 1. Previously, you explored **object-oriented programming (OOP)** concepts such as class hierarchies, abstraction, inheritance, interfaces, and polymorphism. In this assignment, you will evolve your system into a **data-driven application** that uses the **Java Collections Framework** and **File I/O**. The goal is to make your program behave like a realistic software system—able to dynamically add or remove vehicles, efficiently maintain and query information, and persist its data between program runs. This exercise helps you understand how modern OOP systems scale in practice through dynamic data structures and persistent storage.

2. Overall Learning Objectives

By the end of this assignment, you should be able to:

- Choose suitable collection types (`List`, `Set`, `TreeSet`) to represent and organize data.
- Demonstrate dynamic data management using addition, removal, and iteration.
- Apply sorting and ordering using comparators or tree-based collections.
- Perform **File I/O** using Java's standard `java.io` package.
- Integrate these mechanisms into your existing OOP class hierarchy.
- Produce modular, well-commented, and maintainable Java code.

3. Detailed Problem Statement

3.1. Context

The logistics company from Assignment 1 continues to expand its operations. Its fleet now includes various vehicles, *e.g.*, cars, trucks, buses, airplanes, and cargo ships, *etc.*, each with specific attributes such as ID, model, speed, and efficiency. In your previous version, vehicles may have been stored in arrays or basic lists. While functional, such approaches may have limitations: arrays are fixed in size, managing duplicates could have been cumbersome, and data may have been lost when the program terminated.

3.2. Your Task

You are now required to refactor and extend your system to be both **collection-based** and **persistent**.

1. **Replace array-based storage:** Use one or more appropriate `Java Collections` (e.g., `ArrayList`, `LinkedList`, `HashSet`, `TreeSet`) to hold your vehicles dynamically.
2. **Ensure uniqueness and ordering:** Use a `Set` to maintain distinct model names and a `TreeSet` (or sorting) for automatic alphabetical order.
3. **Provide data analysis:** Implement methods or utilities to sort vehicles by attributes such as maximum speed, model name, or efficiency. Allow users to view the fastest and slowest vehicles using collection utilities like `Collections.max()` or `Collections.min()`.

4. **Implement persistence:** Enable saving and loading of fleet data using text or CSV files. Each line should contain sufficient details (type, ID, model, speed, efficiency). Handle missing or malformed files gracefully.
5. **Create a simple CLI or driver program:** Provide an interface that allows users to add, display, sort, and persist data. The design and method signatures are left to your discretion.

4. Functional Expectations

You have complete flexibility in designing class structures and method signatures, but the following functionalities should be clearly visible:

- **Dynamic storage:** Vehicles are stored using one or more collections that allow resizing, iteration, and removal.
- **Uniqueness and sets:** Demonstrate duplicate handling using a `HashSet`.
- **Ordering:** Maintain or display sorted data using either `TreeSet` or comparators with `Collections.sort()`.
- **File I/O:** Save and load the fleet data in CSV format using standard Java I/O classes.
- **Reporting:** Generate simple summaries—number of vehicles, distinct models, fastest and slowest vehicle, etc.
- **CLI interaction:** Implement an illustrative console menu to demonstrate all major operations (add, remove, display, sort, save, load, exit).

5. Implementation Guidelines

- Maintain encapsulation—collections should be `private` members.
- You may freely define method names, helper classes, and data formats.
- Use `try-catch-finally` for file operations to ensure files close automatically.
- Always validate inputs and handle exceptions gracefully.
- Begin testing with hard-coded data, then extend to user input.
- Comment your code clearly—especially explaining your choice of collection types.

6. Submission Requirements

1. All Java source files (`.java`), organized and compilable.
2. A sample fleet data file (e.g., `fleetdata.csv`).
3. A concise `README.txt` including:
   - Description on how to compile and run the program.
   - Description on how collections are used and their justification.
   - Description of file I/O implementation.
   - Sample run demonstrating the program's features.

7. Grading Scheme (100 marks)

Breakdown of Marks:

Component	Marks	Detailed Evaluation Criteria
Use of Collections	25	• Correct and meaningful use of at least two different collection classes (e.g., `ArrayList`, `HashSet`, `TreeSet`). • Data manipulation demonstrated through add/remove/iteration. • Full marks: clear understanding and correct behaviour. • Partial marks: collection used but without justification or misused (e.g., duplicates not handled).
Sorting and Ordering	15	• Demonstrates ordered display using either `TreeSet` or `Collections.sort()`. • Comparator or Comparable correctly implemented. • Full marks: correct ordering and clear output. • Partial marks: partial sorting, unclear criterion, or runtime sorting errors.
File I/O and Persistence	20	• Fleet data successfully saved and reloaded from file. • File structure is consistent and readable (CSV or similar). • Proper exception handling for missing/invalid files. • Full marks: robust, tested save/load functionalities. • Partial marks: saves correctly but load incomplete, or vice versa.
Functional Correctness	20	• All major operations (add, remove, display, sort, save, load) work as intended. • Reasonable console interface or automated demo. • No crashes for normal input. • Partial marks: some menu options or features non-functional.
Design and Integration	15	• System integrates seamlessly with Assignment 1 hierarchy. • Vehicle classes reused rather than duplicated. • Good encapsulation; no global/static collection misuse. • Full marks: coherent and extensible design.
Code Quality and Documentation	5	• Proper indentation, naming, and logical structure. • Helpful comments and brief README explanations. • Full marks: clean, professional code. • Partial marks: minimal comments or inconsistent formatting.

Table 1: Marks Breakdown

8. Deliverables Checklist

- Source code that compiles and runs successfully.
- Program uses collection classes (*e.g.,* no ordinary arrays).
- Sorting and persistence tested and verified.
- README includes explanations and sample output.
- Code that is well-formatted and commented.