1.Employee Sorting System

Project Overview

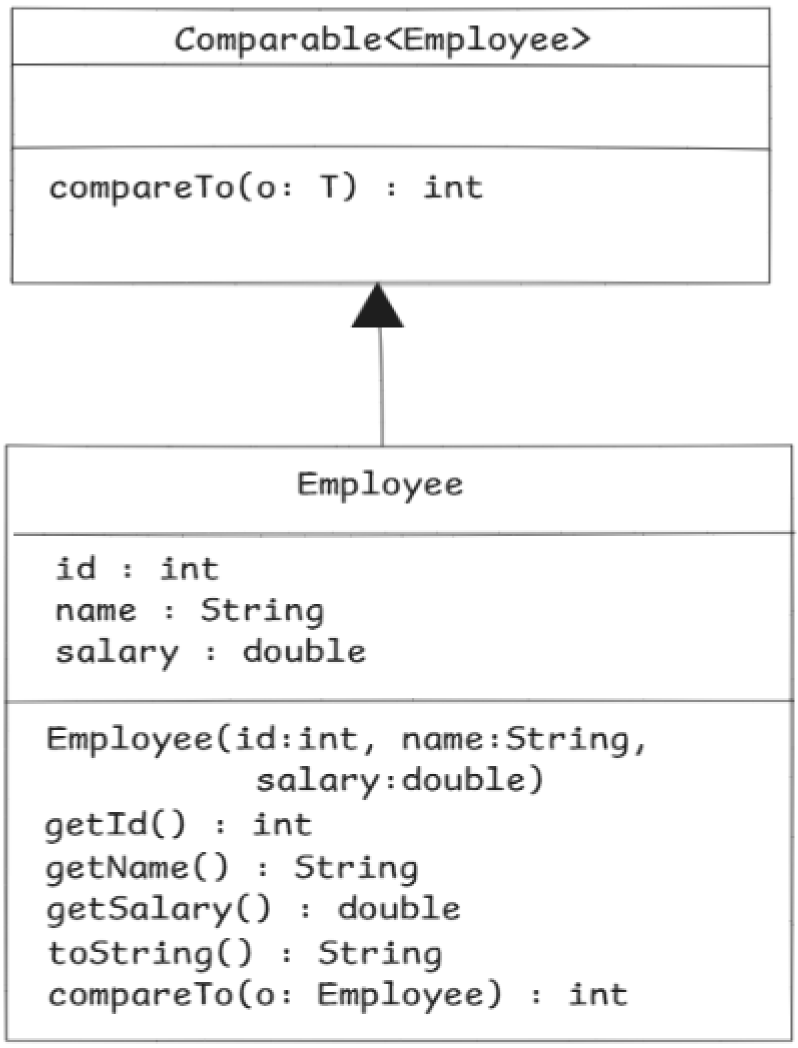
A company named **TechAxis Pvt. Ltd.** wants to implement an internal **Employee Management System** to maintain employee records and display them in sorted order.
To improve efficiency in reporting, the system must automatically sort employee records by their **Employee ID** in ascending order using Java's `Comparable` interface.

Objective

Develop a Java program that defines an `Employee` class implementing `Comparable<Employee>`.
The system should:

- Store employee data (ID, name, salary)
- Display records before and after sorting
- Sort the employee list by **ID in ascending order**

Functional Requirements



Step 1 — Define the `Employee` Class

Create a class `Employee` with:

- **Private fields:** id, name, and salary
- **Constructor:** Initialize all fields
- **Getter methods:** For all fields
- **toString():** For formatted output
- **Implements Comparable:** Override `compareTo()` to compare employees by ID in ascending order.

Step 2 — Hardcode Employee Data

Create multiple employee records with sample data:

- Employee 1 → ID: 201, Name: John, Salary: 50000.0
- Employee 2 → ID: 103, Name: Emma, Salary: 75000.0

- Employee 3 → ID: 150, Name: Liam, Salary: 62000.0
- Employee 4 → ID: 120, Name: Olivia, Salary: 58000.0

Step 3 — Display Employees Before Sorting

Show all employee details in their **original unsorted order**.

```
Employees before sorting:
ID: 201, Name: John, Salary: 50000.0
ID: 103, Name: Emma, Salary: 75000.0
ID: 150, Name: Liam, Salary: 62000.0
ID: 120, Name: Olivia, Salary: 58000.0
```

Step 4 — Sort and Display Employees After Sorting

Sort the list of employees based on their ID (ascending order) and display them.

```
Employees after sorting:
ID: 103, Name: Emma, Salary: 75000.0
ID: 120, Name: Olivia, Salary: 58000.0
ID: 150, Name: Liam, Salary: 62000.0
ID: 201, Name: John, Salary: 50000.0
```

Expected Output

```
Employees before sorting:
ID: 201, Name: John, Salary: 50000.0
ID: 103, Name: Emma, Salary: 75000.0
ID: 150, Name: Liam, Salary: 62000.0
ID: 120, Name: Olivia, Salary: 58000.0

Employees after sorting:
ID: 103, Name: Emma, Salary: 75000.0
ID: 120, Name: Olivia, Salary: 58000.0
ID: 150, Name: Liam, Salary: 62000.0
ID: 201, Name: John, Salary: 50000.0
```

Concepts Demonstrated

- **Comparable Interface:** Implementing `compareTo()` for custom sorting
- **Encapsulation:** Using private fields with public accessors
- **Data Organization:** Sorting structured employee data efficiently
- **Readable Output:** Clear console display using `toString()` override

2. Student Performance Sorting System

Project Overview

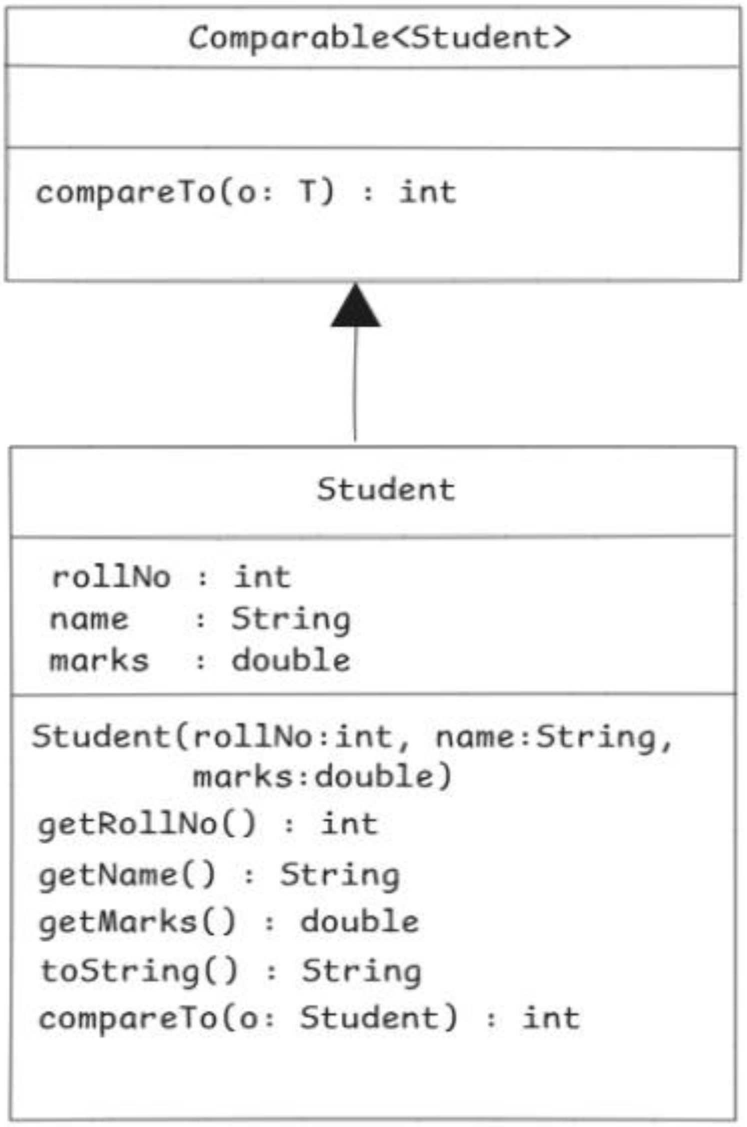
An educational institute named **EduTrack Academy** wants to build a **Student Performance Management System** that maintains student records and sorts them by their marks to generate performance reports.
To achieve this, developers must implement **custom sorting logic** using Java's `Comparable` interface.

Objective

Develop a Java program that defines a `Student` class implementing `Comparable<Student>`.
The system should:

- Store student data (Roll No, Name, Marks)
- Display student records before and after sorting
- Sort students by **Marks in ascending order**

Functional Requirements



Step 1 — Define the `Student` Class

Create a class `Student` with:

- **Private fields:** `rollNo`, `name`, and `marks`
- **Constructor:** Initializes all fields
- **Getter methods:** For all fields
- **toString():** To print student details clearly
- **Implements Comparable:** Override `compareTo()` to compare students based on marks in ascending order

---

Step 2 — Hardcode Student Data

Create an array of `Student` objects using sample data:

- Student 1 → RollNo: 101, Name: Alice, Marks: 85
- Student 2 → RollNo: 102, Name: Bob, Marks: 72
- Student 3 → RollNo: 103, Name: Charlie, Marks: 90
- Student 4 → RollNo: 104, Name: Diana, Marks: 78

Step 3 — Display Students Before Sorting

Show all student records in the **original unsorted order**:

```
Students before sorting:
RollNo: 101, Name: Alice, Marks: 85
RollNo: 102, Name: Bob, Marks: 72
RollNo: 103, Name: Charlie, Marks: 90
RollNo: 104, Name: Diana, Marks: 78
```

Step 4 — Sort and Display Students After Sorting

Sort the student list based on **marks in ascending order** and print the sorted records:

```
Students after sorting (by marks ascending):
RollNo: 102, Name: Bob, Marks: 72
RollNo: 104, Name: Diana, Marks: 78
RollNo: 101, Name: Alice, Marks: 85
RollNo: 103, Name: Charlie, Marks: 90
```

Expected Output

```
Students before sorting:
RollNo: 101, Name: Alice, Marks: 85
RollNo: 102, Name: Bob, Marks: 72
RollNo: 103, Name: Charlie, Marks: 90
RollNo: 104, Name: Diana, Marks: 78

Students after sorting (by marks ascending):
RollNo: 102, Name: Bob, Marks: 72
RollNo: 104, Name: Diana, Marks: 78
RollNo: 101, Name: Alice, Marks: 85
RollNo: 103, Name: Charlie, Marks: 90
```

Concepts Demonstrated

- **Comparable Interface:** Implementing `compareTo()` for custom sorting logic
- **Encapsulation:** Using private fields and controlled access
- **Data Management:** Handling structured student data
- **Sorting of Objects:** Leveraging natural ordering through `Comparable`
- **Readable Output:** Using `toString()` for better console display

3. Problem Statement

Create a Driver class and perform the following tasks:

Tasks:

1. Create a new **`ArrayList`** which should be homogenous and must only store String values.
2. Add 5 names to the ArrayList.
3. Print the list elements
4. Remove third employee from the ArrayList
5. Print the list elements.

Example Output

```
[Smith, Allen, John, King, Tyler]
[Smith, Allen, King, Tyler]
```

4. Bookstore Inventory Sorting System

Project Overview

A digital bookstore named **Readify Books Pvt. Ltd.** is upgrading its inventory management system.
The new system must efficiently organize books either by **title** (alphabetically, ignoring case) or by **price** (ascending order).

To achieve this, developers are required to implement **two separate Comparator classes** to demonstrate flexible and reusable sorting logic.

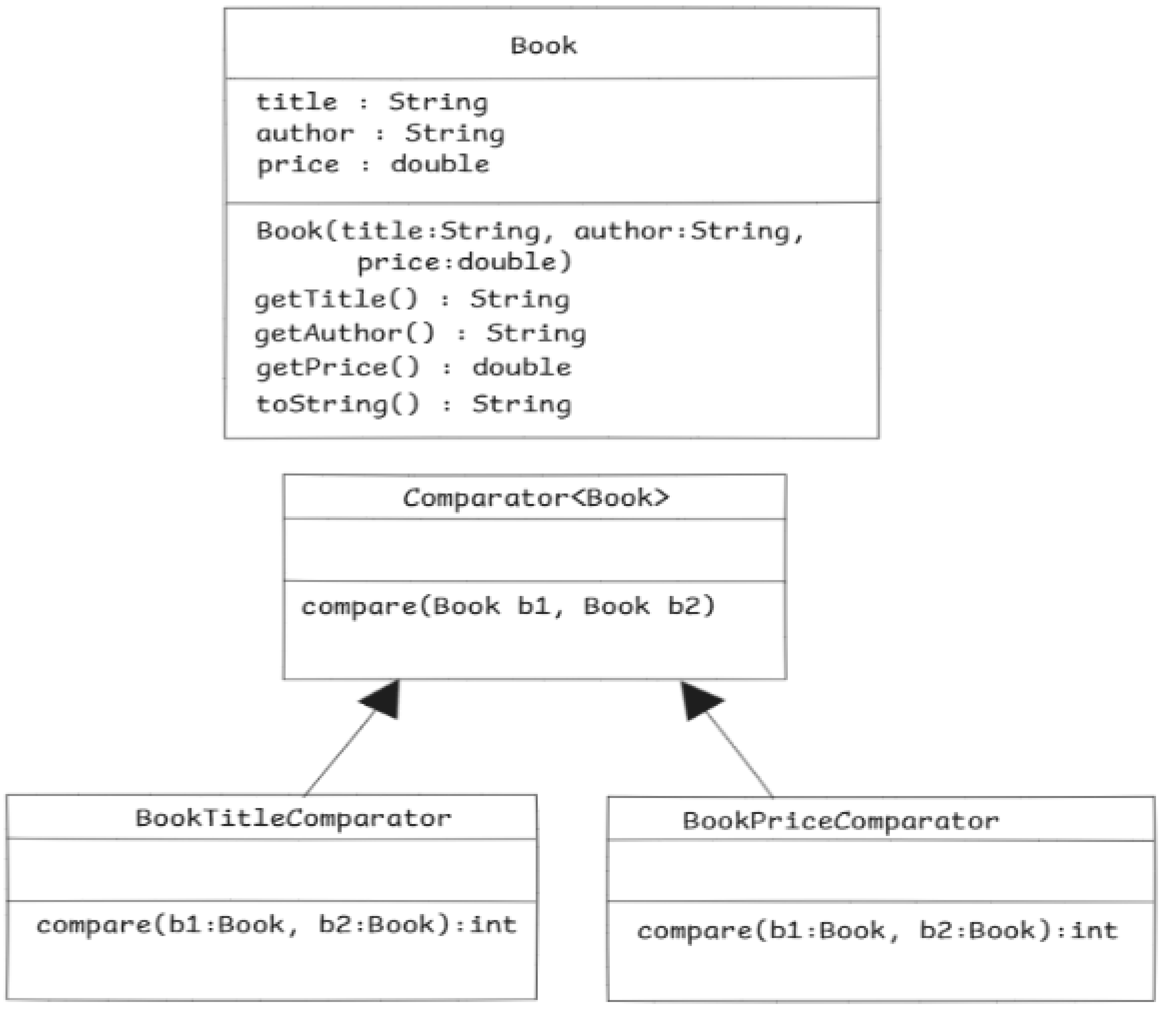
Objective

Develop a Java program that defines a `Book` class and two `Comparator` implementations:

- **BookTitleComparator:** Sorts books by title (case-insensitive)
- **BookPriceComparator:** Sorts books by price (ascending order)

The program must display the list of books before sorting, after sorting by title, and after sorting by price.

Functional Requirements



Step 1 — Define the `Book` Class

Create a class `Book` with:

- **Private fields:** `title`, `author`, and `price`
- **Constructor:** To initialize all fields
- **Getters:** For all fields
- **toString():** To display book details in the format
- `Title: <title>, Author: <author>, Price: <price>`

Step 2 — Create `BookTitleComparator`

- Implement the `Comparator<Book>` interface.
- Override `compare(Book b1, Book b2)` to compare titles **case-insensitively**.
  Logic: `b1.getTitle().compareToIgnoreCase(b2.getTitle())`

Step 3 — Create `BookPriceComparator`

- Implement the `Comparator<Book>` interface.
- Override `compare(Book b1, Book b2)` to compare books by price in ascending order.
  Logic: `Double.compare(b1.getPrice(), b2.getPrice())`

Step 4 — Hardcode Book Data

Create an array of `Book` objects using the following dataset:

- "The Alchemist", "Paulo Coelho", 299.0
- "harry potter", "J.K. Rowling", 399.0
- "1984", "George Orwell", 199.0
- "Clean Code", "Robert C. Martin", 499.0
- "The Pragmatic Programmer", "Andrew Hunt", 450.0

Step 5 — Display Books Before Sorting

Show the list of books in their original order:

```
Books before sorting:
Title: The Alchemist, Author: Paulo Coelho, Price: 299.0
Title: harry potter, Author: J.K. Rowling, Price: 399.0
Title: 1984, Author: George Orwell, Price: 199.0
Title: Clean Code, Author: Robert C. Martin, Price: 499.0
Title: The Pragmatic Programmer, Author: Andrew Hunt, Price: 450.0
```

Step 6 — Sort by Title (Case-Insensitive)

Display the results after sorting alphabetically by title:

```
Books after sorting by title (case-insensitive):
Title: 1984, Author: George Orwell, Price: 199.0
Title: Clean Code, Author: Robert C. Martin, Price: 499.0
Title: harry potter, Author: J.K. Rowling, Price: 399.0
Title: The Alchemist, Author: Paulo Coelho, Price: 299.0
Title: The Pragmatic Programmer, Author: Andrew Hunt, Price: 450.0
```

Step 7 — Sort by Price (Ascending Order)

Display the results after sorting by price:

```
Books after sorting by price (ascending):
Title: 1984, Author: George Orwell, Price: 199.0
Title: The Alchemist, Author: Paulo Coelho, Price: 299.0
```

```
Title: harry potter, Author: J.K. Rowling, Price: 399.0
Title: The Pragmatic Programmer, Author: Andrew Hunt, Price: 450.0
Title: Clean Code, Author: Robert C. Martin, Price: 499.0
```

Concepts Demonstrated

- **Comparator Interface:** Implementing multiple comparison strategies
- **Encapsulation:** Private fields with controlled access through getters
- **Sorting Flexibility:** Sorting the same dataset in multiple ways
- **Case-Insensitive String Comparison:** Using `compareToIgnoreCase()`
- **Data Organization:** Sorting structured object data effectively

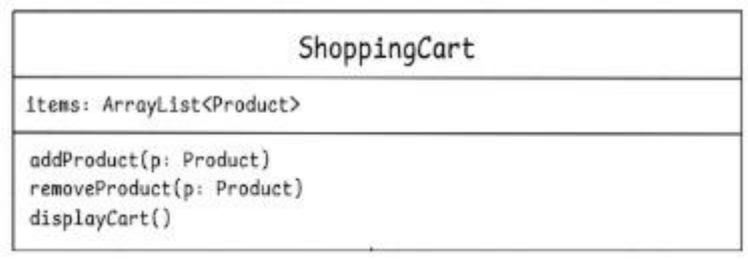
5. Create a program that uses the collection framework with the class `ShoppingCart` to store `Product` objects in the shopping cart object.

Create a class `Product` with the following attributes and methods:

```
- int productID
- String productName
```

Override toString() and equals(Object) method.

Design a constructor to initialize both the attributes



Create a class `ShoppingCart` which contains an `ArrayList` of `Product`.

- This class should have: `items: ArrayList<Product>`
- This class should have the following methods:
  - void addProduct(Product) : To add a Product object to the `items` list.
  - removeProduct(Product): To remove a Product object from the `items` list. Print suitable message if the item is removed from the list or not.
  - displayCart(): Print the Products of the list.

Driver Class Execution

Perform the following tasks

1. Create a `ShoppingCart` object.
2. Add 5 products to the `ShoppingCart` with productID and productName such as `1 Milk`, `2 Tea`, `3 Biscuit`, `4 Coffee`, `5 Chocolate`
3. Call `displayCart()` method.
4. Remove `1 Milk` product object.
5. Call `displayCart()` method.
6. Remove `3 Honey` product object.
7. Call `displayCart()` method.

Example Output

```
[1 Milk, 2 Tea, 3 Biscuit, 4 Coffee, 5 Chocolate]
Item removed successfully
[2 Tea, 3 Biscuit, 4 Coffee, 5 Chocolate]
Item not removed from the cart
[2 Tea, 3 Biscuit, 4 Coffee, 5 Chocolate]
```

6. Create a double primitive and convert it into a Double object using auto-boxing. Then, display the object's type and value using getClass().getName().

Instructions

- Declare a double primitive with a hardcoded value, e.g., double num = 45.67;.
- Use auto-boxing to convert the double primitive into a Double object: Double obj = num;.
- Print the class name using obj.getClass().getName() to verify the object type.
- Print the value of the object using System.out.println(obj);.

Example Output

```
Object Type: java.lang.Double
Object Value: 45.67
```

7. Create an int variable and convert it into an Integer object using three different approaches:

- Using the Integer constructor.
- Using the static Integer.valueOf() method.
- Using auto-boxing (automatic conversion).
- Print all three Integer objects to verify the conversions.

Instructions

- Declare an int variable with a hardcoded value. Example: int num = 42;.
- Convert the int to an Integer using the constructor: Integer obj1 = new Integer(num);.
- Convert the int to an Integer using valueOf(): Integer obj2 = Integer.valueOf(num);.
- Use auto-boxing to convert the int to an Integer: Integer obj3 = num;.
- Print the three Integer objects. (They will print the numeric value when using System.out.println.)

Example Output

```
Using constructor: 42
Using valueOf(): 42
Using auto-boxing: 42
```

8.Use Case: Smart City Traffic Management System

Scenario

You have been hired by a futuristic **Smart City Corporation** to develop a **Traffic Management System**. The system should dynamically manage and analyze vehicle flow across the city.

Every vehicle passing a checkpoint is recorded in the system. The city wants to ensure:

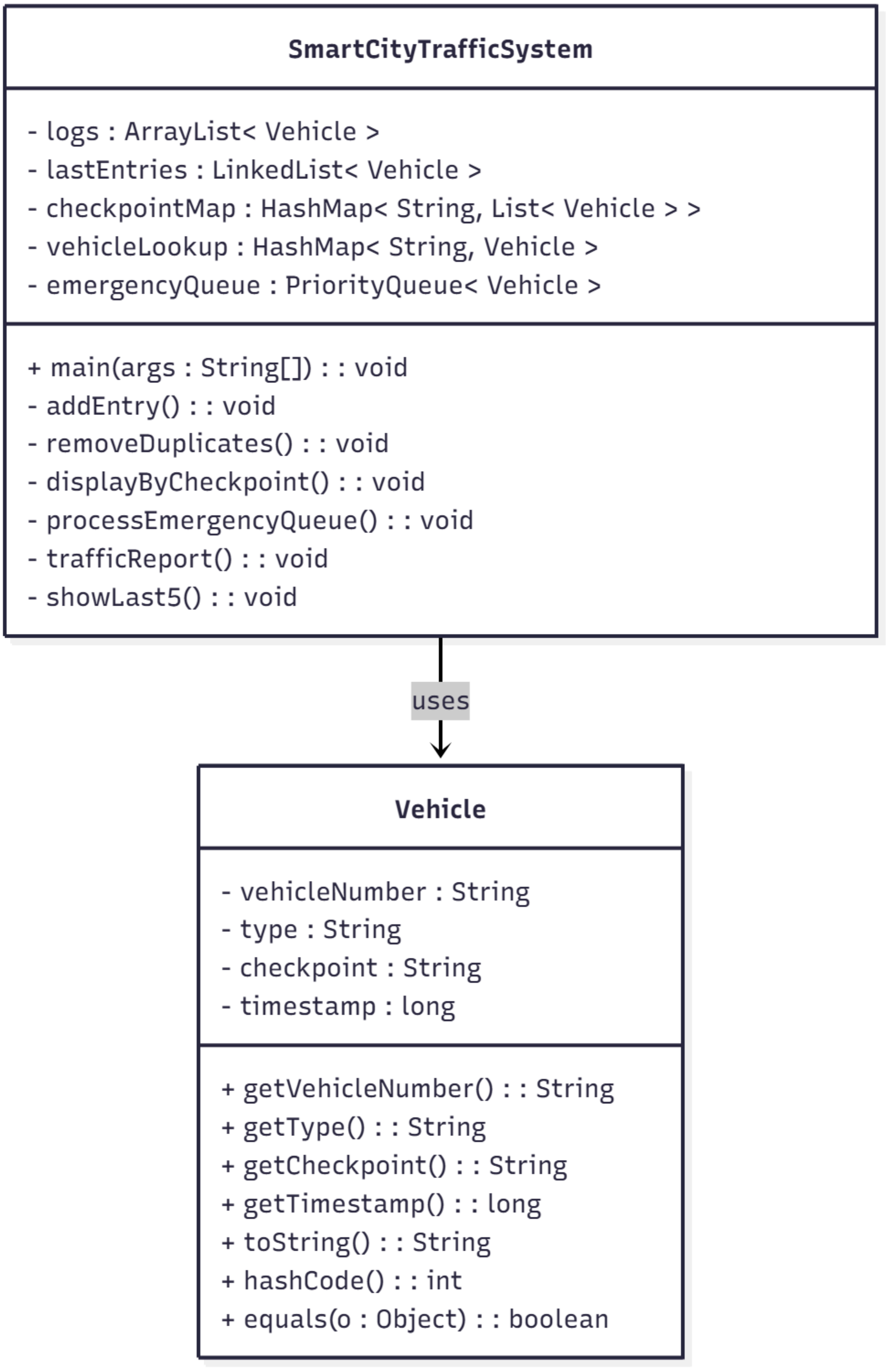
- Efficient storage and retrieval of vehicle data
- Elimination of duplicate entries
- Prioritization of emergency vehicles
- Real-time reporting of traffic congestion

Vehicle Attributes

Each vehicle has:

- `vehicleNumber` (unique, e.g., "MH12AB1234")
- `type` (Car, Bike, Bus, Truck, Ambulance, FireTruck)
- `checkpointName` (e.g., "North Gate")
- `timestamp` (system time when passing checkpoint)

UML Diagram


SmartCityTrafficSystem
- logs : ArrayList< Vehicle >
- lastEntries : LinkedList< Vehicle >
- checkpointMap : HashMap< String, List< Vehicle > >
- vehicleLookup : HashMap< String, Vehicle >
- emergencyQueue : PriorityQueue< Vehicle >
+ main(args : String[]) : : void
- addEntry() : : void
- removeDuplicates() : : void
- displayByCheckpoint() : : void
- processEmergencyQueue() : : void
- trafficReport() : : void
- showLast5() : : void
uses
Vehicle
- vehicleNumber : String
- type : String
- checkpoint : String
- timestamp : long
+ getVehicleNumber() : : String
+ getType() : : String
+ getCheckpoint() : : String
+ getTimestamp() : : long
+ toString() : : String
+ hashCode() : : int
+ equals(o : Object) : : boolean

Functional Requirements

1. Vehicle Entry Logging

- Store all vehicle entries in an **ArrayList**.
- Each new vehicle is logged with its details.

2. Remove Duplicate Entries

- Eliminate duplicates using **HashSet**.
- Duplicates are defined as entries with the same `vehicleNumber` and `timestamp`.

3. Track Vehicles by Checkpoint

- Group vehicles by checkpoint using a **HashMap<String, List>**.
- Allow querying of vehicles for a specific checkpoint.

4. Emergency Vehicle Queue

- Use a **PriorityQueue** to prioritize emergency vehicles (Ambulance > FireTruck > Others).
- Process vehicles from the queue based on their priority.

5. Traffic Analysis

- Calculate congestion per checkpoint using **Map<String, Integer>**.
- Identify busiest and least busy checkpoints.

6. Search Vehicles

- Provide search functionality by `vehicleNumber` (O(1) using HashMap).
- Provide search functionality by `vehicle type`.

7. Generate Reports

- Total vehicles today.
- Count by vehicle type.
- Show last 5 vehicles passing checkpoints (use LinkedList).
- Top 3 busiest checkpoints (can use TreeMap or sorting).

Expected Output

=== SMART CITY TRAFFIC MANAGEMENT ===

```
1. Add Vehicle Entry
2. Remove Duplicates
3. Display Vehicles by Checkpoint
4. Process Emergency Vehicle Queue
5. Traffic Report
6. Show Last 5 Vehicles
7. Exit
   Choose: 1
   Vehicle Number: MH12AB1234
   Type (Car/Bike/Ambulance/FireTruck): Ambulance
   Checkpoint: North Gate
   Entry added!


=== SMART CITY TRAFFIC MANAGEMENT ===
Choose: 1
Vehicle Number: MH12XY5678
Type (Car/Bike/Ambulance/FireTruck): Car
Checkpoint: North Gate
Entry added!

=== SMART CITY TRAFFIC MANAGEMENT ===
Choose: 1
Vehicle Number: MH12AB1234
Type (Car/Bike/Ambulance/FireTruck): Ambulance
Checkpoint: North Gate
Entry added!

=== SMART CITY TRAFFIC MANAGEMENT ===
Choose: 2
Duplicates removed!

=== SMART CITY TRAFFIC MANAGEMENT ===
Choose: 3
Enter checkpoint name: North Gate
Vehicles at North Gate:
Vehicle[MH12AB1234, Ambulance, North Gate, 1699999999999]
Vehicle[MH12XY5678, Car, North Gate, 1699999999999]

=== SMART CITY TRAFFIC MANAGEMENT ===
Choose: 4
Processing emergency vehicle:
Vehicle[MH12AB1234, Ambulance, North Gate, 1699999999999]

=== SMART CITY TRAFFIC MANAGEMENT ===
Choose: 5
--- Traffic Report ---
Checkpoint congestion:
North Gate: 2
Busiest: North Gate
```

```
Least Busy: North Gate

=== SMART CITY TRAFFIC MANAGEMENT ===
Choose: 6
Last 5 vehicles:
Vehicle[MH12XY5678, Car, North Gate, 1699999999999]
Vehicle[MH12AB1234, Ambulance, North Gate, 1699999999999]

=== SMART CITY TRAFFIC MANAGEMENT ===
Choose: 7
Exiting...
```

Objectives / Learning Goals

- Understand **Java Collection Framework**: `ArrayList`, `HashSet`, `HashMap`, `LinkedList`, `PriorityQueue`, `TreeMap`.
- Learn **grouping, sorting, and prioritization** using collections.
- Implement **real-world scenario logic** using OOP and collections.
- Handle **dynamic data efficiently** without using arrays only.
