Advanced Object-Oriented Programming

Lab Assignment 2: World Travel Planner

Stacks and Immutability in Java

Background

In this lab, you will build upon your understanding of graphs from Lab 1 and explore two crucial concepts:

- **Stack Data Structure** — Used for depth-first exploration and backtracking
- **Immutability** — A design principle where objects cannot be modified after creation, leading to safer, more predictable code

You explored **BFS (Breadth-First Search)** in Lab 1, which uses a **Queue** to explore graphs level by level. Now you will implement **DFS (Depth-First Search)** using a **Stack**, which explores as deeply as possible before backtracking.

Problem Statement

You are a *travel enthusiast* planning an epic around-the-world journey. You have a list of cities you want to visit, and you know which cities have direct flight connections. Your goal is to:

- Design an **immutable City class** representing each destination
- Represent the world as a **graph** where cities are vertices and flights are edges
- Implement **Depth-First Search (DFS)** using a Stack to find a path from your starting city to your dream destination
- Print the complete travel route

Design Requirements

1. Immutable City Class

Create a **City** class with the following requirements:

- **Fields**: name (String), country (String), and timezone (String)
- All fields must be **private final**
- No setter methods (immutability principle)
- Provide getter methods for all fields
- Override **equals()** and **hashCode()** based on the city name
- Override **toString()** to return a formatted string: *"City{name='Tokyo', country='Japan', timezone='JST'}"*

2. TravelGraph Class

Create a **TravelGraph** class to represent the world travel network:

- Use **adjacency list representation** with **Map<City, List<City>>**
- Store the map as a **private final** field
- Provide methods to add cities and connections
- **Do NOT use static methods**

Task 1: Implement DFS Path Finding

Add a method **findPath(City start, City destination)** in the TravelGraph class that:

- Uses a **Stack** to implement DFS
- Tracks visited cities to avoid cycles
- Returns a **List<City>** representing the path from start to destination
- Returns an empty list if no path exists

Expected Output Example

For the following world travel network:

```
New York -- London -- Paris
    |            |
  Tokyo  -- Singapore -- Dubai
```

Finding a path from New York to Dubai might produce:

Travel Route: New York → Tokyo → Singapore → Dubai

Task 2: Driver Class

Create a **TravelPlannerTest** class with a main method that:

- Creates at least 8 City objects (use real world cities)
- Builds a TravelGraph with realistic flight connections
- Tests the findPath method with multiple scenarios
- Prints the complete travel route in a readable format

Task 3: Input Validation & Exception Handling

Enhance your classes to handle edge cases:

- Throw **IllegalArgumentException** if City constructor receives null or empty parameters

- Throw **IllegalArgumentException** if findPath receives null cities
- Throw **IllegalArgumentException** if start or destination cities don't exist in the graph
- Handle the case where no path exists gracefully

Bonus Task: Compare BFS vs DFS

Implement both BFS (from Lab 1) and DFS path-finding in the same graph:

- Add a **findPathBFS(City start, City destination)** method
- Compare the paths returned by DFS and BFS
- Print both paths and explain which one might be shorter and why

Conceptual Questions

Answer these questions in comments at the top of your TravelPlannerTest class:

- **Stack vs Queue:** Why does DFS use a Stack while BFS uses a Queue? What fundamental difference in exploration strategy does this create?
- **Immutability Benefits:** What are three key advantages of making the City class immutable? How does this help with using City objects as Map keys?
- **Path Optimality:** Does DFS guarantee finding the shortest path? If not, which algorithm would? Why?
- **Real-world Application:** In a real flight booking system, what additional information would you add to make this more realistic? (Consider flight duration, cost, layover time)
- **Encapsulation:** Identify and explain three examples of encapsulation in your design.

Implementation Notes

- Use **java.util.Stack** for DFS implementation
- Use **java.util.HashSet** to track visited cities efficiently
- Use **java.util.HashMap** for the adjacency list
- Consider using **java.util.ArrayList** for storing paths
- For path reconstruction, you may need to maintain parent tracking

Submission Requirements

Submit the following files:

- **City.java** — Immutable city class
- **TravelGraph.java** — Graph representation with DFS
- **TravelPlannerTest.java** — Driver class with test cases and conceptual answers

Academic Integrity

This is an individual assignment. You may discuss concepts with classmates, but **all code must be your own work**. Any instances of copying code from online sources, other students, or AI tools (unless explicitly permitted for CLO7) will result in an F grade for the course as per the Unfair Means Policy.

Good luck and happy coding!

5

New York;USA;EDT

Tokyo;Japan;JPT

<3 more line>

6

New york (s1) ; Tokyo (s2)

Map<String, City> cities;

City c = new City(“A”, “”, “”);

cities.put(“A”, c);

travelGraph.get(cities.get(s1)).add(cities.get(s2));