420-301-VA: Programming Patterns

Lab 6: Applications of Data Structures

Computer Science & Technology
Vanier College

Introduction

Having explored the fundamental data structures within the Java Collections Framework—such as Lists, Stacks, Queues, Sets, and Maps—we now shift our focus from theory to practice. Understanding **what** a data structure is designed for is only half the battle; knowing **when** and **how** to apply it effectively is what distinguishes a proficient programmer. This lab is dedicated to building that practical skill. We will tackle common computer science problems and build application features by selecting and combining the right data structures for the job.

Objectives

This lab provides hands-on experience in applying data structures to solve complex problems. By the end of this lab, you should be comfortable with:

- Analyzing a problem's requirements to select the most appropriate data structure(s).
- Using a `Stack` to manage nested or sequential data, such as in expression parsing.
- Implementing core application logic, such as an undo/redo system, by composing multiple data structures.
- Gaining confidence in using the Java Collections Framework to build robust and efficient software components.

Instructions

- For each exercise, create the required classes and a main method to test your implementation.
- Copy-paste your formatted code for each exercise into a separate document, followed by screenshots of the console outputs to validate that your code worked as expected.
- You can work with your peers, but each person must write, execute, and document their own work.

Part 1: Problem-Solving using Stacks and Maps

0.1 Concept Review: Validating Balanced Brackets

A classic computer science problem is to determine if an expression containing brackets—(), [], {}—is "balanced". This means every opening bracket has a corresponding closing bracket in the correct order. For example, `{[()]}` is balanced, but `{[)]}` is not.

This problem is a perfect use case for a `Stack` due to its Last-In, First-Out (LIFO) nature, which mirrors the nesting of brackets. The algorithm is as follows:

- Iterate through the expression character by character.
- If you encounter an **opening bracket** ('(', '[', '{'), **push** it onto the stack.
- If you encounter a **closing bracket**, check the top of the stack.
  - If the stack is empty or the bracket at the top is not the corresponding opening bracket, the expression is unbalanced.
  - If it is the correct opening bracket, **pop** it from the stack.
- After iterating through the entire expression, if the stack is **empty**, the expression is balanced. Otherwise, it is unbalanced.

To easily check for matching pairs (e.g., that ')' matches with '('), a `Map` is an excellent tool. We can store the closing brackets as keys and their corresponding opening brackets as values for instant lookups.

```java
import java.util.ArrayDeque;
import java.util.Deque;
import java.util.Map;
import java.util.HashMap;

public class BracketValidator {
    public static void main(String[] args) {
        String expression1 = "a + b * {c - (d / e)}";
        String expression2 = "a + [b * (c - d})]";

        System.out.println("Expression 1 is balanced: " + isBalanced
           (expression1));
        System.out.println("Expression 2 is balanced: " + isBalanced
           (expression2));
    }

    public static boolean isBalanced(String expression) {
        // Use a Map to store the matching pairs
        Map<Character, Character> bracketPairs = new HashMap<>();
        bracketPairs.put(')', '(');
        bracketPairs.put(']', '[');
        bracketPairs.put('}', '{');

        // Use a Deque as a Stack
        Deque<Character> stack = new ArrayDeque<>();
        for (char c : expression.toCharArray()) {
            // If it's an opening bracket, push it onto the stack
            if (bracketPairs.containsValue(c)) {
                stack.push(c);
            }
            // If it's a closing bracket
            else if (bracketPairs.containsKey(c)) {
                // If stack is empty or the top doesn't match, it's
                   unbalanced
                if (stack.isEmpty() || stack.peek() != bracketPairs.
                   get(c)) {
                    return false;
                }
                // If it matches, pop the opening bracket
                stack.pop();
            }
        }

        // If the stack is empty at the end, all brackets were
           matched
        return stack.isEmpty();
    }
}
```

Listing 1: Validating balanced brackets using a Stack and a Map

Exercise 1: Method Call Tracer

In programming, the flow of method calls (especially nested or recursive calls) is managed by a mechanism called the 'call stack'. When a method is invoked, it is pushed onto the stack; when it returns, it is popped off. In this exercise, you will build a `CallTracer` to simulate this behavior and use a `Map` to gather statistics on how many times each method is called.

The Goal: Create a tool that can trace a simulated program's execution, showing when methods are entered and exited, and then provide a summary of how many times each method was invoked.

Your Task:

1. **Create the `CallTracer` class:**
   - It should have two private instance variables:
     - `private Deque<String> callStack;` (Use an `ArrayDeque`)
     - `private Map<String, Integer> callCounts;` (Use a `HashMap`)
   - Initialize both in the constructor.

2. **Implement the tracer methods:**

   - `public void enter(String methodName)`:
     - This method simulates a method call.
     - Get the current count for `methodName` from the `callCounts` map (defaulting to 0 if not present), increment it, and put it back in the map.
     - Print an indented message showing the call. The indentation should be based on the current stack size. For example, use `"\t".repeat(callStack.size())`.
     - Finally, push the `methodName` onto the `callStack`.
   - `public void exit(String methodName)`:
     - This method simulates a method return.
     - Check if the `callStack` is empty or if the top element does not match `methodName`. If either is true, print an error message.
     - If it matches, pop the method from the stack.
     - Print an indented message showing the exit, using the new stack size for indentation.
   - `public void printSummary()`:
     - Iterate through the `callCounts` map and print a summary of how many times each method was called.

3. **Simulate Execution in `main`**

   - In a `main` method, create an instance of your `CallTracer`.
   - Simulate a sequence of method calls. For example:

     ```java
     CallTracer tracer = new CallTracer();
     tracer.enter("main");
     tracer.enter("methodA");
     tracer.enter("methodB");
     tracer.exit("methodB");
     tracer.enter("methodC");
     tracer.exit("methodC");
     tracer.exit("methodA");
     tracer.enter("methodA"); // Call methodA again
     tracer.enter("methodB");
     tracer.exit("methodB");
     tracer.exit("methodA");
     tracer.exit("main");
     tracer.printSummary();
     ```

   - Your output should show the nested calls clearly and provide a correct summary at the end.

Expected Output

```
Entering: main
    Entering: methodA
        Entering: methodB
        Exiting: methodB
        Entering: methodC
        Exiting: methodC
    Exiting: methodA
    Entering: methodA
        Entering: methodB
        Exiting: methodB
    Exiting: methodA
Exiting: main

Call Summary :
main was called 1 time(s).
methodA was called 2 time(s).
methodC was called 1 time(s).
methodB was called 2 time(s).
```

Part 2: Application - Customer Support Ticket System

A common real-world application is a system for managing customer support requests. Such a system needs to handle incoming requests in a fair, first-come, first-served manner, while also allowing support agents to quickly look up the status of any specific request and track customer history. This is a perfect scenario to combine the strengths of a `Queue` and a `Map`.

Concept Review: The Ticket System Logic

- **User-Defined Object:** We will create a `Ticket` class to hold all the information for a single support request, such as a unique ID, the customer's name, the problem description, and its current status (e.g., 'OPEN', 'CLOSED').
- **Queue for Processing:** A `Queue<Ticket>` is the ideal structure for managing incoming tickets. New tickets are added to the end of the queue (`add`), and when a support agent is ready, they handle the ticket at the front of the queue (`remove`). This ensures a First-In, First-Out (FIFO) workflow.
- **Map for Lookup:** A `Map<Integer, Ticket>` will serve as a database of all tickets ever created. The ticket's unique ID is the key, and the `Ticket` object is the value. This allows for instant retrieval of any ticket's details.
- **Map for Counting:** A `Map<String, Integer>` can be used to track statistics, such as how many tickets each customer has submitted. The customer name is the key, and their total ticket count is the value.

Exercise 2: Implement a Support Ticket System

You will build a backend for a support system that can create tickets, process them in order, look them up by ID, and report statistics on customer activity.

Your Task:

1. **Step 1: Create the `Ticket` Class and `Status` Enum**
   - Create an `enum` named `Status` with values `OPEN` and `CLOSED`.
   - Create a class named `Ticket` with private attributes: `int ticketId`, `String customerName`, `String issueDescription`, and `Status status`.
   - Implement a constructor to initialize the fields. The status should be `OPEN` by default.
   - Implement getters for all fields and a setter for the `status`.
   - Override the `toString()` method to return a formatted string with all ticket details.
2. **Step 2: Create the `SupportSystem` Class**
   - Add four private instance variables:
     - `private Queue<Ticket> openTickets;` (Use a `LinkedList`)
     - `private Map<Integer, Ticket> allTickets;` (Use a `HashMap`)

  - private Map<String, Integer> ticketsPerCustomer; (Use a HashMap)
  - private int nextTicketId = 1;
- In the constructor, initialize all queues and maps.

3. **Step 3: Implement the Core Logic**

   - public void createTicket(String customer, String issue):
     - Creates a new Ticket object with the next available ID.
     - Adds the new ticket to the openTickets queue.
     - Puts the new ticket into the allTickets map, using its ID as the key.
     - **Update customer count:** Get the current count for the customer from the ticketsPerCustomer map (defaulting to 0 if not present). Increment the count and put it back in the map.
     - Increments nextTicketId.
   - public Ticket processNextTicket():
     - Checks if the openTickets queue is empty. If so, return null.
     - If not empty, poll() the ticket from the front of the queue.
     - Set the status of this ticket to CLOSED.
     - Return the ticket that was just processed.
   - public Ticket getTicketById(int ticketId):
     - Returns the ticket from the allTickets map corresponding to the given ID.
   - public void printCustomerSummary():
     - Prints a header for the summary.
     - Iterates through the ticketsPerCustomer map's entry set. For each entry, print the customer's name (key) and the number of tickets they've created (value).

4. **Step 4: Detailed Simulation in main**

   - Create a main method and instantiate your SupportSystem.
   - Perform the following sequence of operations **exactly** as listed, printing output at each stage to trace the system's behavior.

     (a) **Create initial tickets:**

     ```
     system.createTicket("Alice", "Printer is not working");
     system.createTicket("Bob", "Cannot connect to WiFi");
     system.createTicket("Alice", "Monitor is flickering");
     system.createTicket("Charlie", "Software license expired
        ");
     system.createTicket("Bob", "Mouse is broken");
     ```

     (b) **Process the first ticket:**

     ```
     System.out.println("--- Processing Next Ticket ---");
     Ticket processedTicket1 = system.processNextTicket();
     System.out.println("Processed: " + processedTicket1);
     System.out.println("----------------------------\n");
     ```

(c) **Look up a ticket by ID:**

```
System.out.println("--- Looking up Ticket ID 3 ---");
Ticket foundTicket = system.getTicketById(3);
System.out.println("Found: " + foundTicket);
System.out.println("----------------------------\n");
```

(d) **Process the second ticket:**

```
System.out.println("--- Processing Next Ticket ---");
Ticket processedTicket2 = system.processNextTicket();
System.out.println("Processed: " + processedTicket2);
System.out.println("----------------------------\n");
```

(e) **Look up a closed ticket:**

```
System.out.println("--- Looking up Closed Ticket ID 1
   ---");
Ticket closedTicket = system.getTicketById(1);
System.out.println("Found: " + closedTicket);
System.out.println("-----------------------------------\
   n");
```

(f) **Print the final customer summary:**

```
system.printCustomerSummary();
```

Expected Console Output

Running the simulation from Step 4 should produce the following output. Minor differences in formatting are acceptable, but the data and order of events should match.

```
--- Processing Next Ticket ---
Processed: Ticket ID: 1, Customer: Alice, Issue: Printer is not working, Status: CLOSED
----------------------------

--- Looking up Ticket ID 3 ---
Found: Ticket ID: 3, Customer: Alice, Issue: Monitor is flickering, Status: OPEN
----------------------------

--- Processing Next Ticket ---
Processed: Ticket ID: 2, Customer: Bob, Issue: Cannot connect to WiFi, Status: CLOSED
----------------------------

--- Looking up Closed Ticket ID 1 ---
Found: Ticket ID: 1, Customer: Alice, Issue: Printer is not working, Status: CLOSED
-----------------------------------

--- Customer Ticket Summary ---
Alice: 2 tickets
Bob: 2 tickets
Charlie: 1 tickets
-----------------------------
```

Part 3 (Optional): Application - Text Editor with Undo/Redo and Versioning

Many applications require undo/redo functionality. This can be elegantly implemented using two stacks. In this exercise, you will build a simple text editor that not only supports undo/redo but also allows users to save and load named versions of their text using a `Map`.

0.2 Concept Review: The Undo/Redo/Versioning Logic

- **Undo/Redo:** Two stacks, an `undoStack` and a `redoStack`, are used to store previous states of the text. An action pushes the old state to the `undoStack`. Undoing an action moves a state from the `undoStack` to the `redoStack`. Redoing moves it back.
- **Versioning:** A `Map` is used to store named snapshots of the text. The key is the version name (a `String`) and the value is the text content at that moment (also a `String`). This allows for non-linear jumps to previously saved states.

Exercise 2: Implement a Simple Text Editor

You will build a `TextEditor` class that manages a string of text and supports adding text, undoing, redoing, and versioning.

Your Task:

1. **Step 1: Create the `TextEditor` Class**
   - Create a class named `TextEditor`.
   - Add four private instance variables:
     - `private StringBuilder text;`
     - `private Deque<String> undoStack;`
     - `private Deque<String> redoStack;`
     - `private Map<String, String> versions;`
   - In the constructor, initialize these variables. Use `ArrayDeque` for the stacks, a `HashMap` for the map, and initialize the `StringBuilder`.
2. **Step 2: Implement the Core Methods**
   - `public void type(String newText)`:
     - Push the current state (`text.toString()`) onto the `undoStack`.
     - Append `newText` to the `StringBuilder`.
     - Clear the `redoStack`.
   - `public void undo()` and `public void redo()`: Implement these as described in the previous version of the lab, using the two stacks.
   - `public void saveVersion(String name)`:
     - Puts the current text (`text.toString()`) into the `versions` map with `name` as the key.
   - `public void loadVersion(String name)`:

    - Find the saved text in the `versions` map.
    - If it exists, push the **current** text state onto the `undoStack` before making a change.
    - Replace the current text with the loaded version.
    - Clear the `redoStack`, as loading a version starts a new history branch.
  - `public String getText()`: A getter that returns `text.toString()`.

3. **Step 3: Simulate the Editor in `main`**

   - Create a `main` method and instantiate your `TextEditor`.
   - Perform a sequence of operations that test all features, including saving and loading a version, and print the state after each step. For example:
     (a) `editor.type("Hello ");`
     (b) `editor.type("World");` (Text: "Hello World")
     (c) `editor.saveVersion("v1");`
     (d) `editor.type(" from Java!");` (Text: "Hello World from Java!")
     (e) `editor.undo();` (Text: "Hello World")
     (f) `editor.loadVersion("v1");` (Text should still be "Hello World")
     (g) `editor.undo();` (Undo the load action, Text: "Hello World from Java!")

4. **Step 4 : Visualizing with a Simple UI**

   - To see your editor in action, you can connect it to a simple graphical user interface (GUI). The code below creates a window with a text area and buttons for all your editor's features.
   - **Your task:** Copy the code into a new Java file. Then, find all the comments marked `// TODO:` and replace them with the correct calls to the `editor` object to make the buttons work.

```java
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;

public class TextEditorGUI {
    // Our backend logic from Exercise 2
    private final TextEditor editor = new TextEditor();

    private JFrame frame;
    private JTextArea textArea;
    private JTextField inputField;
    private JTextField versionNameField;

    public TextEditorGUI() {
        // Frame and Main Panel Setup
        frame = new JFrame("Simple Text Editor");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)
            ;
        frame.setSize(600, 400);
frame.setLayout(new BorderLayout());

// Text Area (Top)
textArea = new JTextArea();
textArea.setEditable(false);
textArea.setFont(new Font("Monospaced", Font.PLAIN,
   14));
JScrollPane scrollPane = new JScrollPane(textArea);
frame.add(scrollPane, BorderLayout.CENTER);

// Controls Panel (Bottom)
JPanel controlsPanel = new JPanel(new GridLayout(3,
   1, 5, 5));

// Typing controls
JPanel typePanel = new JPanel(new BorderLayout(5,0))
   ;
inputField = new JTextField();
JButton typeButton = new JButton("Append Text");
typePanel.add(new JLabel(" Text to Append: "),
   BorderLayout.WEST);
typePanel.add(inputField, BorderLayout.CENTER);
typePanel.add(typeButton, BorderLayout.EAST);

// Undo/Redo controls
JPanel undoRedoPanel = new JPanel(new FlowLayout());
JButton undoButton = new JButton("Undo");
JButton redoButton = new JButton("Redo");
undoRedoPanel.add(undoButton);
undoRedoPanel.add(redoButton);

// Versioning controls
JPanel versionPanel = new JPanel(new BorderLayout
   (5,0));
versionNameField = new JTextField();
JButton saveButton = new JButton("Save Version");
JButton loadButton = new JButton("Load Version");
versionPanel.add(new JLabel(" Version Name: "),
   BorderLayout.WEST);
versionPanel.add(versionNameField, BorderLayout.
   CENTER);
JPanel versionButtons = new JPanel(new GridLayout
   (1,2));
versionButtons.add(saveButton);
versionButtons.add(loadButton);
versionPanel.add(versionButtons, BorderLayout.EAST);

controlsPanel.add(typePanel);
controlsPanel.add(undoRedoPanel);
controlsPanel.add(versionPanel);
frame.add(controlsPanel, BorderLayout.SOUTH);
```

```
        // Action Listeners
        typeButton.addActionListener(e -> {
            // TODO: Get text from inputField and call the
               editor's type method.
            updateTextArea();
            inputField.setText("");
        });

        undoButton.addActionListener(e -> {
            // TODO: Call the editor's undo method.
            updateTextArea();
        });

        redoButton.addActionListener(e -> {
            // TODO: Call the editor's redo method.
            updateTextArea();
        });

        saveButton.addActionListener(e -> {
            // TODO: Get version name and call the editor's
               saveVersion method.
        });

        loadButton.addActionListener(e -> {
            // TODO: Get version name and call the editor's
               loadVersion method.
            updateTextArea();
        });

        // Finalize
        updateTextArea(); // Initial state
        frame.setLocationRelativeTo(null); // Center the
           window
        frame.setVisible(true);
    }

    private void updateTextArea() {
        textArea.setText(editor.getText());
    }

    public static void main(String[] args) {
        // Ensure the GUI is created on the Event Dispatch
           Thread
        SwingUtilities.invokeLater(TextEditorGUI::new);
    }
}
```

Listing 2: Simple GUI for the Text Editor

References

Java Collections Framework and Algorithms

- Java Documentation for Deque Interface: The recommended interface for stack and queue implementations.
- Java Documentation for the Map Interface: Essential for creating lookups like bracket pairs or call counts.
- Baeldung: Guide to Java Deque: A practical guide on using `ArrayDeque` for stack (LIFO) and queue (FIFO) operations.