420-301-VA: Programming Patterns

Lab 3: The Java Collections Framework: List, Stack, Queue

Computer Science & Technology
Vanier College

Introduction

In this lab, we will dive into the Java Collections Framework, a cornerstone of the Java platform. This framework provides a set of high-performance, reusable data structures that are essential for any Java developer. We will focus on some of the most common linear data structures: Lists, Stacks, Queues, and Priority Queues. By understanding their distinct characteristics and use cases, you will learn how to choose the right tool for various programming challenges, leading to more efficient and readable code.

Objectives

This lab provides a hands-on introduction to the core collection interfaces in Java. By the end of this lab, you should be comfortable with:

- Understanding the key differences and use cases for Lists, Stacks, and Queues.
- Using common implementations like `ArrayList`, `LinkedList`, `Stack`, and `PriorityQueue`
- Performing standard operations: adding, removing, accessing, and iterating over elements.
- Implementing a `PriorityQueue` to handle elements based on their natural ordering.
- Applying these data structures to solve practical data processing tasks.

These skills are fundamental for managing data effectively in Java applications and are critical for tackling more complex programming problems and algorithms.

Instructions

- Please try out the examples first to deepen your understanding.
- 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.

1 Part 1: The `List` Interface

1.1 Concept Review: `ArrayList` and `LinkedList`

A `List` is an ordered collection that allows duplicate elements and gives the programmer precise control over where in the list each element is inserted. You can access elements by their integer index. The two most common implementations are `ArrayList` and `LinkedList`.

- `ArrayList`: Backed by a dynamic array. It provides fast random access (getting an element at a specific index) but can be slow for insertions or deletions in the middle of the list, as it may require shifting elements.
- `LinkedList`: Backed by a doubly-linked list. It is fast for adding and removing elements from the beginning or end, but slow for random access, as it requires traversing the list.

```java
import java.util.LinkedList;
import java.util.List;

public class ListExample {
    public static void main(String[] args) {
        // Create a List of Strings using LinkedList
        List<String> fruits = new LinkedList<>();

        // Add elements
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Cherry");
        System.out.println("Initial list: " + fruits);

        // Add an element at a specific index
        fruits.add(1, "Blueberry");
        System.out.println("After adding at index 1: " + fruits);

        // Remove an element
        fruits.remove("Banana");
        System.out.println("After removing 'Banana': " + fruits);
    }
}
```

Listing 1: Using a LinkedList

Exercise 1: Managing a To-Do List

In this exercise, you will create a simple to-do list manager using a `LinkedList`.

1. Create a Java class named `Task` with two private attributes: `taskName` (`String`) and `priority` (`int`).
2. Implement a constructor, getters, and override the `toString()` method to return a formatted string like `"Task: [taskName], Priority: [priority]"`.
3. In a separate class with a `main` method, create a `LinkedList` to store `Task` objects.
4. **Add Tasks:** Create and add at least three different `Task` objects to your list using the `add` method. Print the list.
5. **Add a High-Priority Task:** Add a new, very important task to the *beginning* of the list. Print the list again.
6. **Update a Task:** Use the `set` method to change the task at index 2 to something new. Print the list.
7. **Remove a Task:** Remove the task at index 1. Print the list again to show the change.
8. **Iterate and Display:** Use a for-each loop to iterate through the list and print the details of each remaining task on a new line.
9. **Thinking Exercise:** Change your list's implementation from `new LinkedList<>();` to `new ArrayList<>();`. Does your code still work? Why or why not? What kind of operations would be faster with a `LinkedList` versus an `ArrayList`?

2 Part 2: The `Stack` (LIFO)

2.1 Concept Review: Last-In, First-Out

A `Stack` represents a last-in, first-out (LIFO) collection of objects. The last item added ("pushed") onto the stack is the first item to be removed ("popped"). Java provides a `java.util.Stack` class that includes classic stack operations. Elements are accessed only from the top of the stack.

```java
import java.util.Stack;

public class StackExample {
    public static void main(String[] args) {
        Stack<String> history = new Stack<>();

        // Push elements onto the stack
        history.push("google.com");
        history.push("vaniercollege.qc.ca");
        history.push("github.com");
        System.out.println("Browser History Stack: " + history);

        // Look at the top element without removing it
        String currentPage = history.peek();
        System.out.println("Current Page is: " + currentPage);

        // Pop an element from the stack
        String previousPage = history.pop();
        System.out.println("Went back to: " + previousPage);
        System.out.println("Remaining History: " + history);

        // Check if the stack is empty
        System.out.println("Is stack empty? " + history.empty());
    }
}
```

Listing 2: Using the Stack Class

Exercise 2: Reversing a String

A classic use case for a stack is to reverse a sequence. You will use the `Stack` class to reverse a words.

1. In your `main` method, create a `Stack<Character>`.
2. Choose a string, for example, `"hello"`.
3. **Push Characters:** Iterate through the string one character at a time and `push` each character onto the stack.
4. **Build Reversed String:** After pushing all characters, create a `StringBuilder`. While the stack is not empty, `pop` each character from the stack and append it to the `StringBuilder`.
5. Print the original string and the reversed string to verify your logic.

3 Part 3: The `Queue` Interface (FIFO)

3.1 Concept Review: First-In, First-Out

A `Queue` is a collection designed for holding elements prior to processing, following a first-in, first-out (FIFO) principle. The first element added to the queue will be the first one to be removed. The `Queue` is an interface, so you need to instantiate a concrete implementation. A `LinkedList` is a very common choice for this, as it efficiently supports the add-at-end and remove-from-beginning operations required by a queue.

```java
import java.util.LinkedList;
import java.util.Queue;

public class QueueExample {
    public static void main(String[] args) {
        Queue<String> customerLine = new LinkedList<>();

        // Add elements to the queue
        customerLine.add("Alice");
        customerLine.add("Bob");
        customerLine.add("Charlie");
        System.out.println("Customers in line: " + customerLine);

        // Serve a customer (remove from the queue)
        String servedCustomer = customerLine.remove();
        System.out.println("Served: " + servedCustomer);
        System.out.println("Remaining customers: " + customerLine);
    }
}
```

Listing 3: Using LinkedList as a Queue

Exercise 3: Simulating a Printer Queue

You will simulate a basic printer queue that processes print jobs in the order they are received.

1. Create a class named `PrintJob` with a single attribute: `documentName` (`String`). Include a constructor and a `toString()` method.
2. In a `main` method, create a `Queue<PrintJob>` using a `LinkedList` implementation.
3. **Add Jobs:** Create and add three `PrintJob` objects to the queue (e.g., "Report.docx", "Presentation.pptx", "Spreadsheet.xlsx"). Print the queue status.
4. **Process Jobs:** Write a loop that continues as long as the queue is not empty. In each iteration, `remove` a job from the queue and print a message like `Printing: [documentName]`. After the loop finishes, print a message saying that all jobs are complete.

4 Part 4: The `PriorityQueue`

4.1 Concept Review: Priority-Based Ordering

A `PriorityQueue` is a special type of queue that orders elements based on their "priority". By default, it uses the natural ordering of the elements, meaning the objects must implement the `Comparable` interface. When you call `poll()` or `remove()`, the element with the highest priority (e.g., the smallest value for numbers, or as defined by `compareTo`) is retrieved, regardless of when it was added.

```java
import java.util.PriorityQueue;
import java.util.Queue;

public class PriorityQueueExample {
    public static void main(String[] args) {
        // A min-heap by default; smallest element has highest priority
        Queue<Integer> pq = new PriorityQueue<>();

        pq.add(30);
        pq.add(10);
        pq.add(20);

        System.out.println("Polling: " + pq.poll()); // Outputs 10
        System.out.println("Polling: " + pq.poll()); // Outputs 20
    }
}
```

Listing 4: Using a PriorityQueue with Integers

Exercise 4: Emergency Room Patient Queue

You will model a patient queue for an emergency room where patients are seen based on the severity of their condition, not just their arrival time.

1. Create a class named `Patient` that implements the `Comparable<Patient>` interface.
2. Add two attributes: `name` (`String`) and `urgencyLevel` (`int`), where 1 is the most urgent and 5 is the least urgent.
3. Implement the `compareTo` method. The comparison should be based on `urgencyLevel`. A patient with a lower urgency level should have higher priority.
4. Override the `toString()` method to display patient details.
5. In your `main` method, create a `PriorityQueue<Patient>`.
6. Add several `Patient` objects to the queue with different names and urgency levels, adding them in a non-sorted order.
7. Write a loop that processes patients until the queue is empty. Print the details of each patient as they are "seen" to demonstrate that they are being processed in order of urgency.

References

The List Interface

- Oracle Java Docs: List Interface: The official API documentation.
- Baeldung: ArrayList vs. LinkedList: A detailed comparison of the two main List implementations.
- GeeksforGeeks: List Interface in Java: A tutorial with examples.

The Stack Class

- Oracle Java Docs: Stack Class: The official API documentation for the legacy Stack class.
- Baeldung: A Guide to the Java Stack: A practical guide on how to use the Stack class.
- GeeksforGeeks: Stack Class in Java: A tutorial with code examples.

The Queue Interface

- Oracle Java Docs: Queue Interface: The official API documentation.
- Baeldung: The Java Queue Interface: An excellent overview of queues in Java.
- GeeksforGeeks: Queue Interface in Java: A comprehensive tutorial.

The PriorityQueue Class

- Oracle Java Docs: PriorityQueue Class: The official API documentation.
- Baeldung: Introduction to Java PriorityQueue: A detailed guide on how priority queues work.
- GeeksforGeeks: PriorityQueue Class in Java: A tutorial with various examples.