

Day 5: Manisha Assignment

Task 1: Implementing a Linked List

1) Write a class **CustomLinkedList** that implements a singly linked list with methods for **InsertAtBeginning**, **InsertAtEnd**, **InsertAtPosition**, **DeleteNode**, **UpdateNode**, and **DisplayAllNodes**. Test the class by performing a series of insertions, updates, and deletions.

1. Node Class: Represents each element in the linked list, containing data and a reference to the next node.

2. CustomLinkedList Class: Implements a singly linked list with methods to manipulate the list:

- **InsertAtBeginning:** Adds a new node at the start of the list.
- **InsertAtEnd:** Adds a new node at the end of the list.
- **InsertAtPosition:** Inserts a new node at a specified position in the list.
- **DeleteNode:** Removes the node with a specified value from the list.
- **UpdateNode:** Changes the value of an existing node.
- **DisplayAllNodes:** Prints all nodes in the list to the console.

3. Main Class: Contains the main method to test the linked list by performing insertions, updates, and deletions, and displays the list after each operation to verify correctness.

Explanation:

1. **Node Class:** A simple node class that holds an integer data and a reference to the next node.
2. **CustomLinkedList Class:** Implements the linked list with methods for inserting at the beginning, end, and specific positions, deleting a node, updating a node, and displaying all nodes.
3. **Main Class:** Contains the main method to test the **CustomLinkedList** class with a series of insertions, updates, and deletions. The output is printed to the console to verify the operations.

```
File Edit Source Refactor Navigate Project Run Window Help
File Edit Source Refactor Navigate Project Run Window Help
1 package com;
2 import java.util.*;
3 class Node {
4     int data;
5     Node next;
6 }
7
8 public class Node {
9     int data;
10    Node next;
11 }
12
13 // CustomLinkedList class
14 class CustomLinkedList {
15     private Node head;
16
17     // Insert a node at the beginning
18     void insertAtBeginning(int data) {
19         Node node = new Node(data);
20         node.next = head;
21         head = node;
22     }
23
24     // Insert a node at the end
25     void insertAtEnd(int data) {
26         Node node = new Node(data);
27         if (head == null) {
28             head = node;
29         } else {
30             Node current = head;
31             while (current.next != null) {
32                 current = current.next;
33             }
34             current.next = node;
35         }
36     }
37
38     // Insert a node at a specific position
39     void insertAtPosition(int data, int position) {
40         if (position < 0) {
41             throw new IndexOutOfBoundsException("Position cannot be negative");
42         }
43         Node node = new Node(data);
44         if (position == 0) {
45             node.next = head;
46             head = node;
47         } else {
48             Node current = head;
49             for (int i = 0; i < position - 1; i++) {
50                 current = current.next;
51             }
52             if (current.next == null) {
53                 throw new IndexOutOfBoundsException("Position out of range");
54             }
55             node.next = current.next;
56             current.next = node;
57         }
58     }
59
60     // Delete a node with a specific value
61     void deleteNode(int data) {
62         if (head == null) return;
63
64         Node current = head;
65         if (current.data == data) {
66             head = current.next;
67             return;
68         }
69         while (current.next != null && current.next.data != data) {
70             current = current.next;
71         }
72         if (current.next == null) {
73             return;
74         }
75         if (current.next.data == data) {
76             current.next = current.next.next;
77         }
78     }
79
80     // Update a node's value
81     void updateNode(int oldData, int newData) {
82         Node current = head;
83         while (current != null) {
84             if (current.data == oldData) {
85                 current.data = newData;
86                 return;
87             }
88             current = current.next;
89         }
90     }
91
92     // Display all nodes
93     public void displayAllNodes() {
94         Node current = head;
95
96         while (current != null) {
97             System.out.println(current.data);
98             current = current.next;
99         }
100    }
101 }
```

```
File Edit Source Refactor Navigate Project Run Window Help
File Edit Source Refactor Navigate Project Run Window Help
1 package com;
2 import java.util.*;
3 class CustomLinkedList {
4     private Node head;
5
6     void insertAtBeginning(int data) {
7         Node node = new Node(data);
8         node.next = head;
9         head = node;
10    }
11
12    void insertAtEnd(int data) {
13        Node node = new Node(data);
14        if (head == null) {
15            head = node;
16        } else {
17            Node current = head;
18            while (current.next != null) {
19                current = current.next;
20            }
21            current.next = node;
22        }
23    }
24
25    void insertAtPosition(int data, int position) {
26        if (position < 0) {
27            throw new IndexOutOfBoundsException("Position cannot be negative");
28        }
29        Node node = new Node(data);
30        if (position == 0) {
31            node.next = head;
32            head = node;
33        } else {
34            Node current = head;
35            for (int i = 0; i < position - 1; i++) {
36                current = current.next;
37            }
38            if (current.next == null) {
39                throw new IndexOutOfBoundsException("Position out of range");
40            }
41            node.next = current.next;
42            current.next = node;
43        }
44    }
45
46    void deleteNode(int data) {
47        Node current = head;
48
49        if (current.data == data) {
50            head = current.next;
51            return;
52        }
53        while (current.next != null && current.next.data != data) {
54            current = current.next;
55        }
56        if (current.next == null) {
57            return;
58        }
59        if (current.next.data == data) {
60            current.next = current.next.next;
61        }
62    }
63
64    void updateNode(int oldData, int newData) {
65        Node current = head;
66
67        while (current != null) {
68            if (current.data == oldData) {
69                current.data = newData;
70                return;
71            }
72            current = current.next;
73        }
74    }
75
76    void displayAllNodes() {
77        Node current = head;
78
79        while (current != null) {
80            System.out.println(current.data);
81            current = current.next;
82        }
83    }
84 }
```

```
File Edit Source Refactor Navigate Project Run Window Help
File Edit Source Refactor Navigate Project Run Window Help
1 package com;
2 import java.util.*;
3
4 public class Main {
5     public static void main(String[] args) {
6         CustomLinkedList list = new CustomLinkedList();
7
8         list.insertAtBeginning(10);
9         list.insertAtBeginning(20);
10        list.insertAtEnd(30);
11        list.insertAtPosition(40, 15);
12
13        list.displayAllNodes();
14
15        list.deleteNode(20);
16
17        list.displayAllNodes();
18
19        list.updateNode(40, 50);
20
21        list.displayAllNodes();
22
23        list.deleteNode(50);
24
25        list.displayAllNodes();
26
27        list.insertAtPosition(50, 25);
28
29        list.displayAllNodes();
30
31        list.deleteNode(50);
32
33        list.displayAllNodes();
34
35        list.insertAtPosition(50, 25);
36
37        list.displayAllNodes();
38
39        list.deleteNode(50);
40
41        list.displayAllNodes();
42
43        list.insertAtPosition(50, 25);
44
45        list.displayAllNodes();
46
47        list.deleteNode(50);
48
49        list.displayAllNodes();
50
51        list.insertAtPosition(50, 25);
52
53        list.displayAllNodes();
54
55        list.deleteNode(50);
56
57        list.displayAllNodes();
58
59        list.insertAtPosition(50, 25);
60
61        list.displayAllNodes();
62
63        list.deleteNode(50);
64
65        list.displayAllNodes();
66
67        list.insertAtPosition(50, 25);
68
69        list.displayAllNodes();
70
71        list.deleteNode(50);
72
73        list.displayAllNodes();
74
75        list.insertAtPosition(50, 25);
76
77        list.displayAllNodes();
78
79        list.deleteNode(50);
80
81        list.displayAllNodes();
82
83        list.insertAtPosition(50, 25);
84
85        list.displayAllNodes();
86
87        list.deleteNode(50);
88
89        list.displayAllNodes();
90
91        list.insertAtPosition(50, 25);
92
93        list.displayAllNodes();
94
95        list.deleteNode(50);
96
97        list.displayAllNodes();
98
99        list.insertAtPosition(50, 25);
100    }
101 }
```

```
File Edit Source Refactor Navigate Project Run Window Help
File Edit Source Refactor Navigate Project Run Window Help
1 package com;
2 import java.util.*;
3
4 public class Main {
5     public static void main(String[] args) {
6         CustomLinkedList list = new CustomLinkedList();
7
8         list.insertAtBeginning(10);
9         list.insertAtBeginning(20);
10        list.insertAtEnd(30);
11        list.insertAtPosition(40, 15);
12
13        list.displayAllNodes();
14
15        list.deleteNode(20);
16
17        list.displayAllNodes();
18
19        list.updateNode(40, 50);
20
21        list.displayAllNodes();
22
23        list.deleteNode(50);
24
25        list.displayAllNodes();
26
27        list.insertAtPosition(50, 25);
28
29        list.displayAllNodes();
30
31        list.deleteNode(50);
32
33        list.displayAllNodes();
34
35        list.insertAtPosition(50, 25);
36
37        list.displayAllNodes();
38
39        list.deleteNode(50);
40
41        list.displayAllNodes();
42
43        list.insertAtPosition(50, 25);
44
45        list.displayAllNodes();
46
47        list.deleteNode(50);
48
49        list.displayAllNodes();
50
51        list.insertAtPosition(50, 25);
52
53        list.displayAllNodes();
54
55        list.deleteNode(50);
56
57        list.displayAllNodes();
58
59        list.insertAtPosition(50, 25);
60
61        list.displayAllNodes();
62
63        list.deleteNode(50);
64
65        list.displayAllNodes();
66
67        list.insertAtPosition(50, 25);
68
69        list.displayAllNodes();
70
71        list.deleteNode(50);
72
73        list.displayAllNodes();
74
75        list.insertAtPosition(50, 25);
76
77        list.displayAllNodes();
78
79        list.deleteNode(50);
80
81        list.displayAllNodes();
82
83        list.insertAtPosition(50, 25);
84
85        list.displayAllNodes();
86
87        list.deleteNode(50);
88
89        list.displayAllNodes();
90
91        list.insertAtPosition(50, 25);
92
93        list.displayAllNodes();
94
95        list.deleteNode(50);
96
97        list.displayAllNodes();
98
99        list.insertAtPosition(50, 25);
100    }
101 }
```

Task 2: Stack and Queue Operations

1) Create a `CustomStack` class with operations `Push`, `Pop`, `Peek`, and `IsEmpty`. Demonstrate its LIFO behavior by pushing integers onto the stack, then popping and displaying them until the stack is empty.

CustomStack Class Implementation

1. CustomStack Class:

- Push: Adds an element to the top of the stack.
 - Pop: Removes and returns the element from the top of the stack.
 - Peek: Returns the element at the top of the stack without removing it.
 - IsEmpty: Checks if the stack is empty.

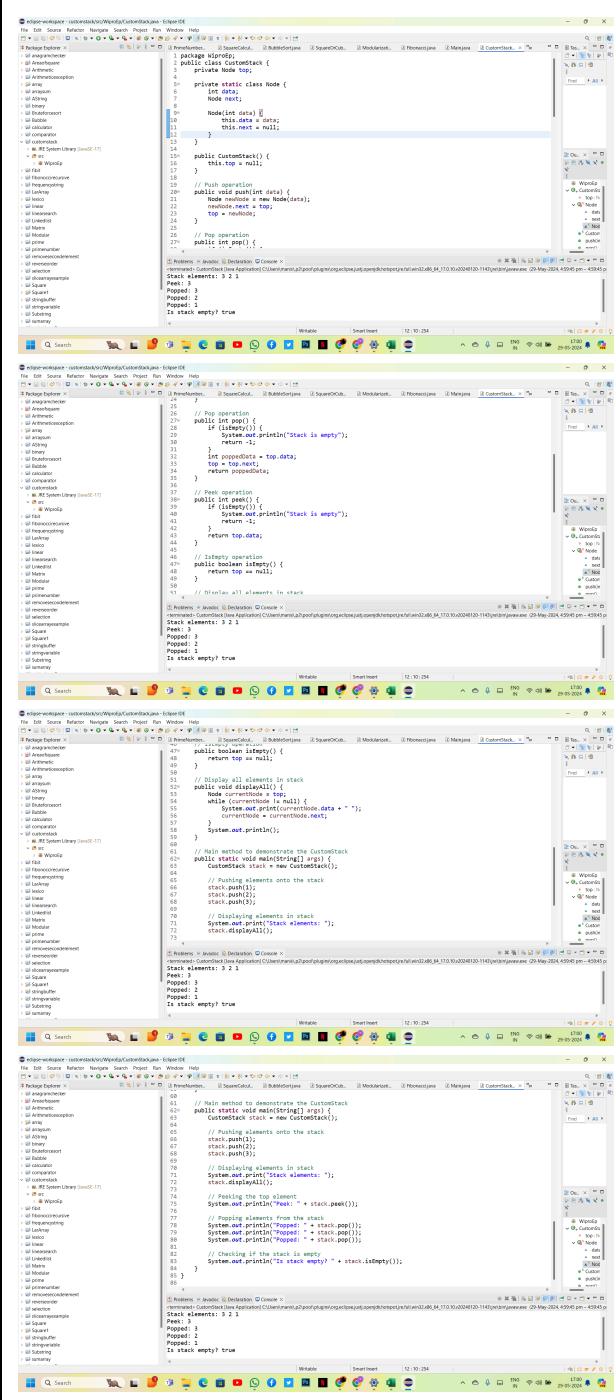
Explanation

1. Node Class: Represents an element in the stack. It contains data and a reference to the next node.

2. CustomStack Class: Contains methods for stack operations.

- `push(int data)`: Adds a new node with the given data to the top of the stack.
 - `pop()`: Removes and returns the top node's data. If the stack is empty, it prints a message and returns -1.
 - `peek()`: Returns the top node's data without removing it. If the stack is empty, it prints a message and returns -1.
 - `isEmpty()`: Checks if the stack is empty.
 - `displayAll()`: Prints all elements in the stack

3. Main Method: Demonstrates the stack's LIFO behavior by pushing integers onto the stack, then popping and displaying them until the stack is empty.



2) Develop a CustomQueue class with methods for Enqueue, Dequeue, Peek, and IsEmpty. Show how your queue can handle different data types by enqueueing strings and integers, then dequeuing and displaying them to confirm FIFO order.

Explanation of the Functionality

1. CustomQueue Class:

- The 'CustomQueue' class uses a 'Queue' from the Java Collections Framework (specifically a 'LinkedList' implementation) to store elements.

- It has a generic type '<T>', making it capable of handling any data type.

2. Enqueue Method:

- Adds an element to the end of the queue using `queue.add(item)`.

3. Dequeue Method:

- Removes and returns the first element of the queue using `queue.poll()`.

- Checks if the queue is empty before attempting to dequeue to avoid exceptions.

4. Peek Method:

- Returns the first element of the queue without removing it using `queue.peek()`.

- Checks if the queue is empty before attempting to peek to avoid exceptions.

5. IsEmpty Method:

- Returns 'true' if the queue is empty and 'false' otherwise using `queue.isEmpty()`.

6. Main Method:

- Demonstrates the usage of 'CustomQueue' with two instances: one for strings and one for integers.

- Enqueues and dequeues elements, showing how the queue maintains the First-In-First-Out (FIFO) order.

```

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

public class CustomQueue<T> {
    private Queue<T> queue = new LinkedList<T>();
}

// Enqueue method to add elements to the queue
public void enqueue(T item) {
    queue.add(item);
}

// Dequeue method to remove and return the first element from the queue
public T dequeue() {
    if (isEmpty()) {
        System.out.println("Queue is empty. Cannot dequeue.");
        return null;
    }
    return queue.poll();
}

// Peek method to return the first element without removing it
public T peek() {
    if (isEmpty()) {
        System.out.println("Queue is empty. Nothing to peek.");
        return null;
    }
    return queue.peek();
}

// IsEmpty method to check if the queue is empty
public boolean isEmpty() {
    return queue.isEmpty();
}

```

```

public class CustomQueue {
    public static void main(String[] args) {
        CustomQueue<String> stringQueue = new CustomQueue<String>();
        CustomQueue<Integer> intQueue = new CustomQueue<Integer>();

        // Enqueue strings
        stringQueue.enqueue("A");
        stringQueue.enqueue("B");
        stringQueue.enqueue("C");

        // Enqueue integers
        intQueue.enqueue(1);
        intQueue.enqueue(2);
        intQueue.enqueue(3);

        // Dequeue and display strings
        while (!stringQueue.isEmpty()) {
            System.out.println(stringQueue.dequeue());
        }

        // Dequeue and display integers
        while (!intQueue.isEmpty()) {
            System.out.println(intQueue.dequeue());
        }
    }
}

```

```

public class CustomQueue {
    public static void main(String[] args) {
        CustomQueue<String> stringQueue = new CustomQueue<String>();
        CustomQueue<Integer> intQueue = new CustomQueue<Integer>();

        // Enqueue strings
        stringQueue.enqueue("A");
        stringQueue.enqueue("B");
        stringQueue.enqueue("C");

        // Enqueue integers
        intQueue.enqueue(1);
        intQueue.enqueue(2);
        intQueue.enqueue(3);

        // Dequeue and display strings
        while (!stringQueue.isEmpty()) {
            System.out.println(stringQueue.dequeue());
        }

        // Dequeue and display integers
        while (!intQueue.isEmpty()) {
            System.out.println(intQueue.dequeue());
        }
    }
}

```

```

public class CustomQueue {
    public static void main(String[] args) {
        CustomQueue<String> stringQueue = new CustomQueue<String>();
        CustomQueue<Integer> intQueue = new CustomQueue<Integer>();

        // Enqueue strings
        stringQueue.enqueue("A");
        stringQueue.enqueue("B");
        stringQueue.enqueue("C");

        // Enqueue integers
        intQueue.enqueue(1);
        intQueue.enqueue(2);
        intQueue.enqueue(3);

        // Dequeue and display strings
        while (!stringQueue.isEmpty()) {
            System.out.println(stringQueue.dequeue());
        }

        // Dequeue and display integers
        while (!intQueue.isEmpty()) {
            System.out.println(intQueue.dequeue());
        }
    }
}

```

Task 3: Priority Queue Scenario

a) Implement a priority queue to manage emergency room admissions in a hospital. Patients with higher urgency should be served before those with lower urgency.

Explanation of the Functionality

1. Patient Class:

- Represents a patient with a name and an urgency level.
- Contains a constructor, getter methods, and a `toString` method for displaying patient information.

2. UrgencyComparator Class:

- Implements the `Comparator` interface to compare `Patient` objects based on their urgency.
- The `compare` method returns a comparison result such that patients with higher urgency are given higher priority.

3. EmergencyRoom Class:

- Contains the `main` method where the program execution begins.
- Creates a `PriorityQueue` of `Patient` objects with the custom `UrgencyComparator` to manage the order of patients.
- Adds patients to the queue with varying urgency levels.
- Processes patients in order of their urgency by polling from the priority queue and displaying the patient being treated.

The screenshot displays three Java code editors side-by-side, each showing a different class definition:

- Patient Class:**

```
package Mips;
import java.util.PriorityQueue;
public class Patient {
    private String name;
    private int urgency;
    public Patient(String name, int urgency) {
        this.name = name;
        this.urgency = urgency;
    }
    public String getName() {
        return name;
    }
    public int getUrgency() {
        return urgency;
    }
}
```
- UrgencyComparator Class:**

```
package Mips;
import java.util.Comparator;
public class UrgencyComparator implements Comparator<Patient> {
    @Override
    public int compare(Patient p1, Patient p2) {
        return p2.getUrgency() - p1.getUrgency();
    }
}
```
- EmergencyRoom Class:**

```
package Mips;
import java.util.PriorityQueue;
public class EmergencyRoom {
    public static void main(String[] args) {
        PriorityQueue<Patient> queue = new PriorityQueue<Patient>((new UrgencyComparator());
        queue.add(new Patient("Michael Brown", 10));
        queue.add(new Patient("Jane Smith", 8));
        queue.add(new Patient("John Doe", 5));
        queue.add(new Patient("Emily Davis", 3));
        // Processing patients in order of urgency
        while (!queue.isEmpty()) {
            Patient patient = queue.poll();
            System.out.println("Treating " + patient);
        }
    }
}
```

Each editor window shows the code with syntax highlighting and some output from the Java runtime environment at the bottom, indicating the processing of patients in order of urgency.