Day 5: Manisha Assignment

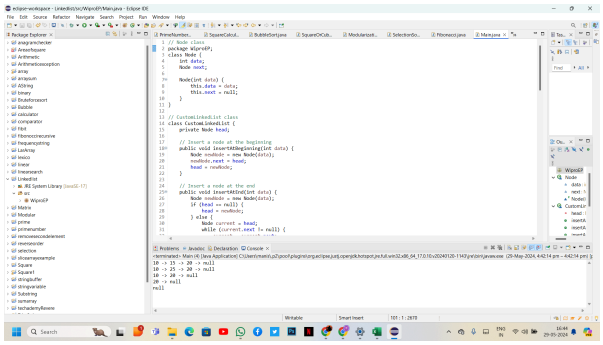
Task 1: Implementing a Linked List

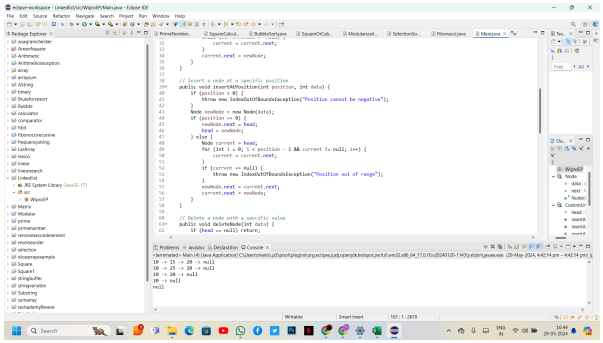
1) Write a class CustomLinkedList that implements a singly linked list with methods for InsertAtBeginning, InsertAtEnd, InsertAtPosition, DeleteNode, UpdateNode, and DisplayAllNodes. Test the class by performing a series of insertions, updates, and deletions.

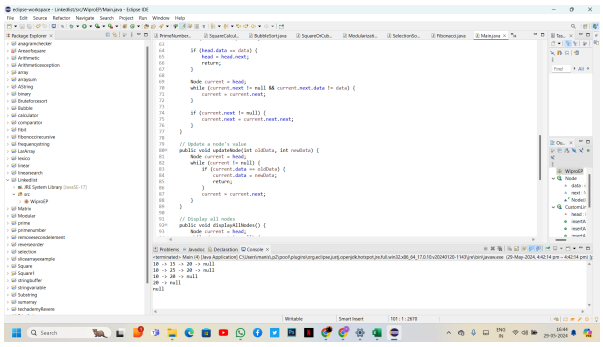
1. Node Class: Represents each element in the linked list, containing data and a reference to the next node.

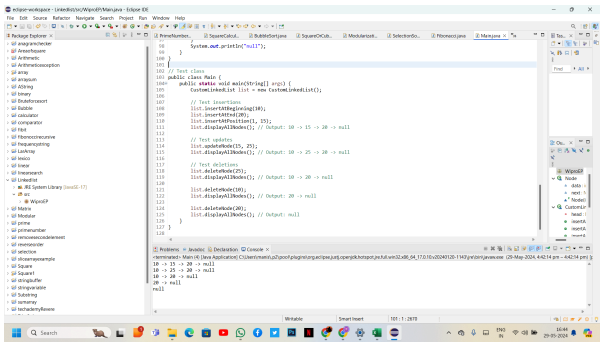
2. CustomLinkedList Class: Implements a singly linked list with methods to manipulate the list:
- InsertAtBeginning: Adds a new node at the start of the list.
- InsertAtEnd: Adds a new node at the end of the list.
- InsertAtPosition: Inserts a new node at a specified position in the list.
-DeleteNode: 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.**

Task 2: Stack and Queue Operations

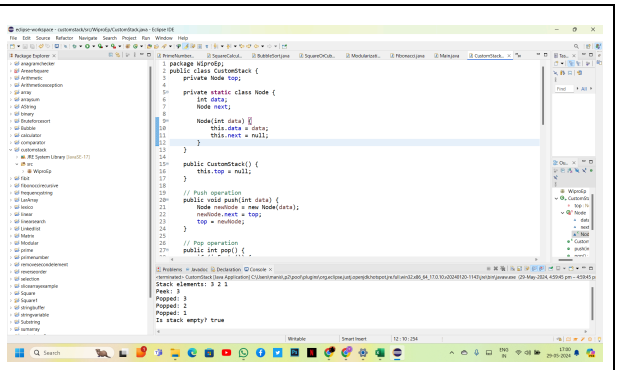
1) Create a CustomStack class with operations Push, Pop, Peek, and IsEmpty. Demonstrate its LIFO behavior by pushing integers onto the stack, then popping and displaying them until the stack is empty.

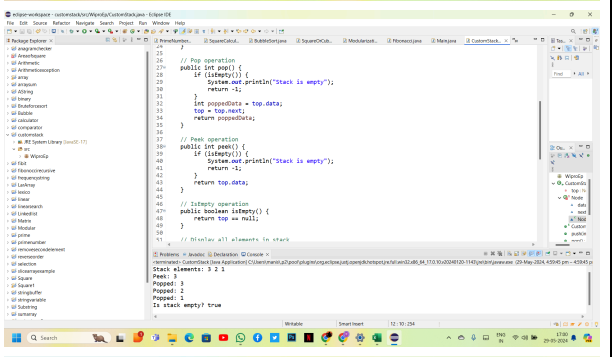
CustomStack Class Implementation

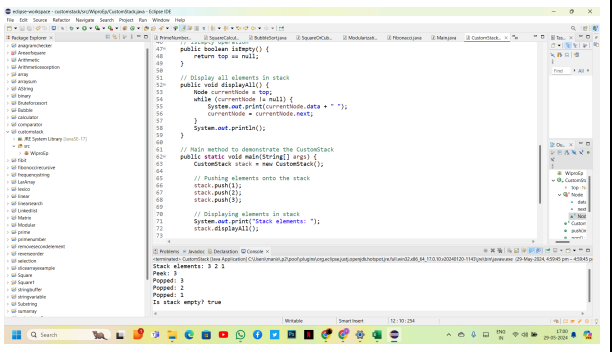
1. CustomStack Class:
   - Push: Adds an element to the top of the stack.
   - Pop: Removes and returns the element from the top of the stack.
   - Peek: Returns the element at the top of the stack without removing it.
   - IsEmpty: Checks if the stack is empty.

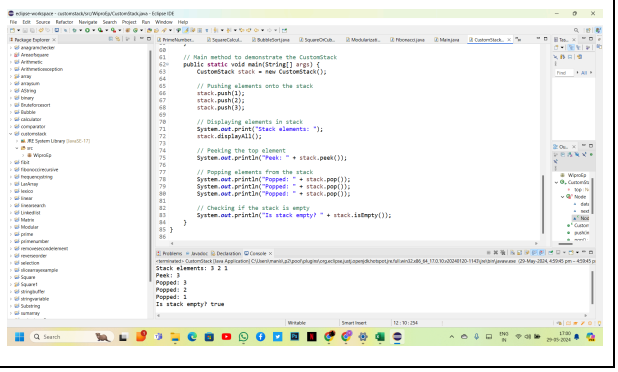
Explanation

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 enqueuing 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.

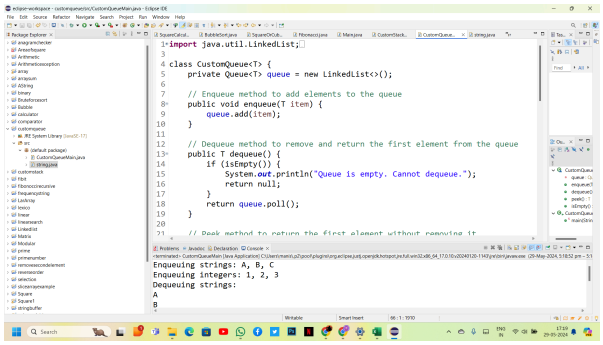
2. Enqueue Method:
   - Adds an element to the end of the queue using `queue.add(item)`.

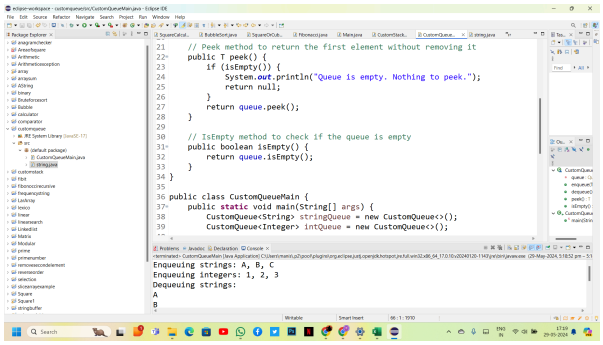
3. Dequeue Method:
   - Removes and returns the first element of the queue using `queue.poll()`.
   - Checks if the queue is empty before attempting to dequeue to avoid exceptions.

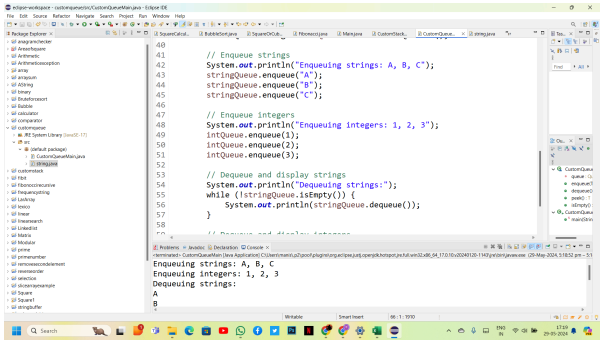
4. Peek Method:
   - Returns the first element of the queue without removing it using `queue.peek()`.
   - Checks if the queue is empty before attempting to peek to avoid exceptions.

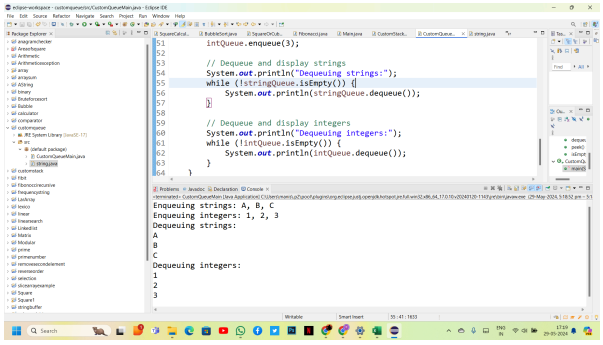
5. IsEmpty Method:
   - Returns `true` if the queue is empty and `false` otherwise using `queue.isEmpty()`.

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.









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.