Part-01

```java
import java.util.EmptyStackException;

public class MyStack<E> {
    private Node<E> first;
    private int size;

    public MyStack() {
        first = null;
        size = 0;
    }

    public E push(E item) {
        Node<E> newNode = new Node<>(item, first);
        first = newNode;
        size++;
        return item;
    }

    private E detach() {
        if (isEmpty()) {
            throw new EmptyStackException();
        }
        E data = first.data;
        first = first.next;
        size--;
        return data;
```

```
        first = newNode;
        size++;
        return item;
    }

    private E detach() {
        if (isEmpty()) {
            throw new EmptyStackException();
        }
        E data = first.data;
        first = first.next;
        size--;
        return data;
    }

    public E pop() {
        return detach();
    }

    public E peek() {
        if (isEmpty()) {
            throw new EmptyStackException();
        }
        return first.data;
    }

    public int size() {
        return size;
```

```
        StringBuilder sb = new StringBuilder();
        sb.append("[");
        Node<E> current = first;
        Stack<E> tempStack = new Stack<>();
        while (current != null) {
            tempStack.push(current.data);
            current = current.next;
        }

        while (!tempStack.isEmpty()) {
            sb.append(tempStack.pop());
            if (!tempStack.isEmpty()) {
                sb.append(", ");
            }
        }

        sb.append("]");
        return sb.toString();
    }

    private static class Node<E> {
        private E data;
        private Node<E> next;

        public Node(E data, Node<E> next) {
            this.data = data;
            this.next = next;
        }
```

This code block provides the implementation of a generic stack data structure in Java. The MyStack class uses a linked list-based approach to implement the stack. It includes methods for pushing, popping, peeking, checking the size, and checking if the stack is empty.

The inner class Node represents each node in the linked list and contains the data and a reference to the next node. The push method adds a new node to the top of the stack, the pop method removes and returns the top element, the peek method returns the top element without removing it, and the size and isEmpty methods provide information about the stack.

The toString method is overridden to generate a string representation of the stack in the form of "[e1, e2, e3, ...]", where e1, e2, e3, etc., are the elements in the stack from top to bottom.

Overall, this code block implements a simple generic stack data structure using a linked list, providing fundamental stack operations and string representation.

Part-02

```java
package utils;

import java.util.NoSuchElementException;

public class MyQueue<E> {

    private static class Node<E> {
        E data;
        Node<E> next;

        public Node(E data) {
            this.data = data;
            this.next = null;
        }

        public Node(E data, Node<E> next) {
            this.data = data;
            this.next = next;
        }
    }

    private Node<E> first;
    private Node<E> last;
    private int size;

    public MyQueue() {
        this.first = null;
        this.last = null;
```

```
29          this.size = 0;
30      }
31
32      public boolean add(E item) {
33          append(item);
34          size++;
35          return true;
36      }
37
38      private void append(E item) {
39          Node<E> newNode = new Node<>(item);
40          if (last == null) {
41              first = newNode;
42          } else {
43              last.next = newNode;
44          }
45          last = newNode;
46      }
47
48      private E detach() {
49          if (first == null) {
50              throw new NoSuchElementException();
51          }
52          E data = first.data;
53          first = first.next;
54          if (first == null) {
55              last = null;
```

```
56          }
57          return data;
58      }
59
60      public E remove() {
61          E data = detach();
62          size--;
63          return data;
64      }
65
66      public E peek() {
67          if (first == null) {
68              return null;
69          }
70          return first.data;
71      }
72
73      public int size() {
74          return size;
75      }
76
77      public boolean isEmpty() {
78          return size == 0;
79      }
80
81      @Override
82      public String toString() {
```

```
    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        sb.append("[");
        Node<E> current = first;
        while (current != null) {
            sb.append(current.data);
            if (current.next != null) {
                sb.append(", ");
            }
            current = current.next;
        }
        sb.append("]");
        return sb.toString();
    }
}
```

This code block provides the implementation of a generic queue data structure in Java. The MyQueue class uses a linked list-based approach to implement the queue. It includes methods for adding elements to the queue, removing elements, peeking at the front element, checking the size, and checking if the queue is empty.

The inner class Node represents each node in the linked list and contains the data and a reference to the next node. The add method adds a new node to the end of the queue, the remove method removes and returns the front element, the peek method returns the front element without removing it, and the size and isEmpty methods provide information about the queue.

The toString method is overridden to generate a string representation of the queue in the form of "[e1, e2, e3, ...]", where e1, e2, e3, etc., are the elements in the queue from front to back.

Overall, this code block implements a simple generic queue data structure using a linked list, providing fundamental queue operations and string representation.

Part-03

_01_MyStackTest class demonstrates the usage of the MyStack class. It creates a new instance of MyStack and pushes several elements onto the stack. After pushing the elements, it prints the stack using the toString method.

_02_MyQueueTest class demonstrates the usage of the MyQueue class and provides additional

methods. It showcases three functionalities: converting a stack to a queue, converting a queue to a stack, and removing the minimum value from a stack.

The changeStackToQueue method takes a MyStack object as input, creates a new MyQueue object, and moves all elements from the stack to the queue using the pop and add methods.

The changeQueueToStack method takes a MyQueue object as input, creates a new MyStack object, and moves all elements from the queue to the stack using the remove and push methods.

The removeMin method takes a MyStack object as input, finds the minimum value in the stack, removes it from the stack, and returns the minimum value. It uses a temporary stack (tempStack) to store the popped values from the original stack while finding the minimum value. After finding the minimum value, it pushes back all the values from tempStack to the original stack except for the minimum value.

In the main method of _02_MyQueueTest, these methods are called and their results are printed for demonstration purposes.

Overall, these code blocks showcase the usage of the MyStack and MyQueue classes and demonstrate the functionalities of converting between stacks and queues and removing the minimum value from a stack.

Result: