Topics

1. Create Queue Interface

```
public interface Queue<E> {
    boolean isEmpty();
    int size();
    E first();
    void enqueue(E element);
    E dequeue();
}
```

2. Create Queue Using Array

```
public class ArrayQueue<E> implements Queue<E> {
    private static final int DEFAULT_CAPACITY = 10;
    private E[] queueArray;
    private int front;
    private int rear;
    private int size;

    public ArrayQueue() {
        this(DEFAULT_CAPACITY);
    }

    public ArrayQueue(int capacity) {
        if (capacity <= 0) {
            throw new IllegalArgumentException("Capacity must be
positive");
        }
        queueArray = (E[]) new Object[capacity];
        front = 0;
        rear = -1;
        size = 0;
    }

    public boolean isEmpty() {
```

```
        return size == 0;
    }

    public int size() {
        return size;
    }

    public E first() {
        if (isEmpty()) {
            throw new IllegalStateException("Queue is empty");
        }
        return queueArray[front];
    }

    public void enqueue(E element) {
        if (size == queueArray.length) {
            resize(2 * queueArray.length);
        }
        rear = (rear + 1) % queueArray.length;
        queueArray[rear] = element;
        size++;
    }

    public E dequeue() {
        if (isEmpty()) {
            throw new IllegalStateException("Queue is empty");
        }
        E element = queueArray[front];
        queueArray[front] = null;
        front = (front + 1) % queueArray.length;
        size--;
        if (size > 0 && size == queueArray.length / 4) {
            resize(queueArray.length / 2);
        }
        return element;
```

```
    }

    private void resize(int capacity) {
        E[] newArray = (E[]) new Object[capacity];
        for (int i = 0; i < size; i++) {
            newArray[i] = queueArray[(front + i) % queueArray.length];
        }
        queueArray = newArray;
        front = 0;
        rear = size - 1;
    }
}
```

3. Create Queue Using Linked Lists

```
public class LinkedQueue<E> implements Queue<E> {
    private Node<E> front;
    private Node<E> rear;
    private int size;

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

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

    public LinkedQueue() {
        front = null;
        rear = null;
        size = 0;
    }

    public boolean isEmpty() {
```

```
        return size == 0;
    }

    public int size() {
        return size;
    }

    public E first() {
        if (isEmpty()) {
            throw new IllegalStateException("Queue is empty");
        }
        return front.element;
    }

    public void enqueue(E element) {
        Node<E> newNode = new Node<>(element, null);
        if (isEmpty()) {
            front = newNode;
        } else {
            rear.next = newNode;
        }
        rear = newNode;
        size++;
    }

    public E dequeue() {
        if (isEmpty()) {
            throw new IllegalStateException("Queue is empty");
        }
        E element = front.element;
        front = front.next;
        size--;
        if (isEmpty()) {
            rear = null;
        }
```

```
            return element;
        }
    }
```

4. Implement Basic Methods of Queue
    - isEmpty()
    - size()
    - first()
    - enqueue(E e)
    - dequeue()

```
public class Main {
    public static void main(String[] args) {
        Queue<Integer> arrayQueue = new ArrayQueue<>();
        arrayQueue.enqueue(1);
        arrayQueue.enqueue(2);
        arrayQueue.enqueue(3);

        System.out.println(arrayQueue.size());
        System.out.println(arrayQueue.first());

        arrayQueue.dequeue();
        System.out.println(arrayQueue.size());
        System.out.println(arrayQueue.first());

        Queue<String> linkedQueue = new LinkedQueue<>();
        linkedQueue.enqueue("Hello");
        linkedQueue.enqueue("World");

        System.out.println(linkedQueue.size());
        System.out.println(linkedQueue.first());

        linkedQueue.dequeue();
        System.out.println(linkedQueue.size());
        System.out.println(linkedQueue.first());
    }
}
```

Homework

1. Augment the ArrayQueue implementation with a new rotate( ) method having semantics identical to the combination, enqueue(dequeue( )). But, your implementation should be more efficient than making two separate calls (for example, because there is no need to modify the size).

```
public void rotate() {
    if (size > 1) {
        E element = dequeue();
        enqueue(element);
    }
}
```

2. Implement the clone( ) method for the ArrayQueue class.

```
@Override
public ArrayQueue<E> clone() {
    try {
        ArrayQueue<E> clone = (ArrayQueue<E>) super.clone();
        clone.queueArray = Arrays.copyOf(queueArray,
queueArray.length);
        return clone;
    } catch (CloneNotSupportedException e) {
        throw new InternalError(e);
    }
}
```

3. Implement a method with signature concatenate(LinkedQueue Q2) for the LinkedQueue class that takes all elements of Q2 and appends them to the end of the original queue. The operation should run in O(1) time and should result in Q2 being an empty queue.

```
public void concatenate(LinkedQueue<E> Q2) {
    if (Q2.isEmpty()) {
        return;
    }
```

```
        if (isEmpty()) {
            front = Q2.front;
        } else {
            rear.next = Q2.front;
        }
        rear = Q2.rear;
        size += Q2.size;
        Q2.front = null;
        Q2.rear = null;
        Q2.size = 0;
    }
```

4. Use a queue to solve the Josephus Problem.

```
    public static int josephus(int n, int k) {
        Queue<Integer> queue = new ArrayDeque<>();
        for (int i = 1; i <= n; i++) {
            queue.add(i);
        }

        while (queue.size() > 1) {
            for (int i = 0; i < k - 1; i++) {
                queue.add(queue.remove());
            }
            queue.remove();
        }

        return queue.remove();
    }
```

5. Use a queue to simulate Round Robin Scheduling.

```
public static void roundRobinScheduling(int[] processes, int[] burstTimes, int timeQuantum) {
    Queue<Integer> queue = new ArrayDeque<>();
    int n = processes.length;
    int[] remainingTimes = new int[n];
    int totalTime = 0;
```

```
    for (int i = 0; i < n; i++) {
        queue.add(processes[i]);
        remainingTimes[i] = burstTimes[i];
        totalTime += burstTimes[i];
    }

    while (!queue.isEmpty()) {
        int process = queue.remove();
        int executionTime = Math.min(timeQuantum, remainingTimes[process]);
        remainingTimes[process] -= executionTime;
        totalTime -= executionTime;
        System.out.println("Process " + process + " executed for " + executionTime + " units.");

        if (remainingTimes[process] > 0) {
            queue.add(process);
        }

        if (totalTime > 0) {
            queue.add((process + 1) % n);
        }
    }
}
```