Queue

```
public class ArrayQueue {
    private int[] queue;
    private int front;
    private int rear;
    private int capacity;

    public ArrayQueue(int size) {
        capacity = size;
        queue = new int[capacity];
        front = -1;
        rear = -1;
    }

    // Enqueue operation
    public void enqueue(int value) {
        if (isFull()) {
            System.out.println("Queue is full");
            return;
        }
        if (isEmpty()) {
            front = 0;
        }
        rear = (rear + 1) % capacity;
        queue[rear] = value;
        System.out.println("Enqueued: " + value);
    }
```

```
// Dequeue operation
public int dequeue() {
    if (isEmpty()) {
        System.out.println("Queue is empty");
        return -1;
    }
    int result = queue[front];
    if (front == rear) {
        front = rear = -1;  // Reset queue after last element is dequeued
    } else {
        front = (front + 1) % capacity;
    }
    System.out.println("Dequeued: " + result);
    return result;
}

// Get front element
public int getFront() {
    if (isEmpty()) {
        System.out.println("Queue is empty");
        return -1;
    }
    return queue[front];
}

// Check if queue is empty
public boolean isEmpty() {
    return front == -1;
}
```

```
// Check if queue is full
private boolean isFull() {
    return (rear + 1) % capacity == front;
}

public static void main(String[] args) {
    ArrayQueue queue = new ArrayQueue(5); // Initialize the queue with capacity 5

    // Enqueue elements
    queue.enqueue(10);
    queue.enqueue(20);
    queue.enqueue(30);
    queue.enqueue(40);
    queue.enqueue(50);

    // Try to enqueue another element
    queue.enqueue(60); // This should print "Queue is full"

    // Dequeue a few elements
    queue.dequeue(); // Dequeues 10
    queue.dequeue(); // Dequeues 20

    // Peek at the front element
    System.out.println("Front element: " + queue.getFront()); // Should show 30

    // Enqueue another element
    queue.enqueue(60); // This should now be possible
```

```
        // Check if the queue is empty
        System.out.println("Is the queue empty? " + queue.isEmpty()); // Should return false

        // Empty the queue
        while (!queue.isEmpty()) {
            queue.dequeue();
        }

        // Check if the queue is empty
        System.out.println("Is the queue empty? " + queue.isEmpty()); // Should return true
    }
}
```

Output

```
Enqueued: 10

Enqueued: 20

Enqueued: 30

Enqueued: 40

Enqueued: 50

Queue is full

Dequeued: 10

Dequeued: 20

Front element: 30

Enqueued: 60

Is the queue empty? false

Dequeued: 30

Dequeued: 40

Dequeued: 50

Dequeued: 60
```

```
Is the queue empty? True
```

Queue Using LinkedList

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

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

        // Enqueue elements
        queue.offer("Alice");
        queue.offer("Bob");
        queue.offer("Charlie");
        queue.offer("Diana");
        queue.offer("Evan");

        // Dequeue a few elements
        System.out.println("Dequeued: " + queue.poll());
        System.out.println("Dequeued: " + queue.poll());

        // Peek at the front element
        System.out.println("Front element: " + queue.peek());

        // Check if the queue is empty
        System.out.println("Is the queue empty? " + queue.isEmpty());

        // Continue to empty the queue
        while (!queue.isEmpty()) {
            System.out.println("Dequeued: " + queue.poll());
        }

        // Check if the queue is empty after all operations
        System.out.println("Is the queue empty? " + queue.isEmpty());
    }
}
```

Output

```
Dequeued: Alice
Dequeued: Bob
Front element: Charlie
Is the queue empty? false
Dequeued: Charlie
Dequeued: Diana
Dequeued: Evan
Is the queue empty? true
```

ArrayDeque

```
import java.util.ArrayDeque;
import java.util.Deque;

public class DequeDemo {
    public static void main(String[] args) {
        // Create a new deque using ArrayDeque
        Deque<String> deque = new ArrayDeque<>();
```

```
// Adding elements to the front and back
deque.addFirst("Element at Front 1");  // Add to front
deque.addLast("Element at Back");      // Add to back
deque.addFirst("Element at Front 2");  // Another element to front

// Display all elements in the deque
System.out.println("Current Deque: " + deque);

// Peeking at elements from the front and back
System.out.println("First Element: " + deque.peekFirst());
System.out.println("Last Element: " + deque.peekLast());

// Removing elements from the front and back
System.out.println("Removed from Front: " + deque.removeFirst());
System.out.println("Removed from Back: " + deque.removeLast());

// Display the final state of the deque
System.out.println("Deque after removals: " + deque);

// Check if deque is empty
System.out.println("Is the deque empty? " + deque.isEmpty());

// Try to remove elements from an empty deque
deque.removeFirst(); // Removing from front
System.out.println("Removed one element from front, remaining: " + deque);

// Add more elements and clear the deque
deque.addFirst("New Front");
deque.addLast("New Back");
```

```
        System.out.println("Deque before clearing: " + deque);

        deque.clear();

        System.out.println("Deque after clearing: " + deque);

    }

}
```

Output

```
Current Deque: [Element at Front 2, Element at Front 1, Element at Back]

First Element: Element at Front 2

Last Element: Element at Back

Removed from Front: Element at Front 2

Removed from Back: Element at Back

Deque after removals: [Element at Front 1]

Is the deque empty? false

Removed one element from front, remaining: []

Deque before clearing: [New Front, New Back]

Deque after clearing: []
```

Priority Queue

```
public class Task implements Comparable<Task> {

    private int priority;

    private String description;


    public Task(int priority, String description) {

        this.priority = priority;

        this.description = description;

    }
    @Override
    public int compareTo(Task other) {
        // Lower values have higher priority
        return Integer.compare(this.priority, other.priority);
    }

    @Override
    public String toString() {
        return description + " (Priority: " + priority + ")";
    }
}


import java.util.PriorityQueue;

public class TaskManager {
    public static void main(String[] args) {
        PriorityQueue<Task> taskQueue = new PriorityQueue<>();
        // Adding tasks to the priority queue
        taskQueue.add(new Task(5, "Complete the quarterly report"));
        taskQueue.add(new Task(1, "Emergency meeting with the team"));
        taskQueue.add(new Task(3, "Schedule annual review"));
        taskQueue.add(new Task(2, "Update project roadmap"));
        taskQueue.add(new Task(4, "Reply to client emails"));

        // Processing tasks based on their priority
        System.out.println("Processing tasks based on priority:");
        while (!taskQueue.isEmpty()) {
            System.out.println(taskQueue.poll());
        }
    }
}
```

Output

```
Processing tasks based on priority:

Emergency meeting with the team (Priority: 1)

Update project roadmap (Priority: 2)

Schedule annual review (Priority: 3)

Reply to client emails (Priority: 4)

Complete the quarterly report (Priority: 5)
```

Circular Queue

```
public class CircularQueue {
    private int[] data;
    private int front;
    private int rear;
    private int size;
    private int capacity;

    // Constructor to initialize the queue
        •   public CircularQueue(int capacity) {
        this.capacity = capacity;
        data = new int[capacity];
        front = 0;
        rear = 0;
        size = 0;
    }
```

```
// Enqueue elements to the rear
public boolean enqueue(int value) {
    if (isFull()) {
        System.out.println("Queue is full");
        return false;
    }
    data[rear] = value;
    rear = (rear + 1) % capacity;
    size++;
    return true;
}

// Dequeue elements from the front
public Integer dequeue() {
    if (isEmpty()) {
        System.out.println("Queue is empty");
        return null;
    }
    int result = data[front];
    front = (front + 1) % capacity;
    size--;
    return result;
}

// Check if the queue is full
public boolean isFull() {
    return size == capacity;
}

// Check if the queue is empty
```

```
public boolean isEmpty() {
    return size == 0;
}

// Display the contents of the queue
public void displayQueue() {
    if (isEmpty()) {
        System.out.println("Queue is empty");
        return;
    }
    System.out.print("Queue contents: ");
    int i = front;
    for (int count = 0; count < size; count++) {
        System.out.print(data[i] + " ");
        i = (i + 1) % capacity;
    }
    System.out.println();
}

public static void main(String[] args) {
    CircularQueue queue = new CircularQueue(5);

    // Enqueue elements
    queue.enqueue(1);
    queue.enqueue(2);
    queue.enqueue(3);
    queue.enqueue(4);
    queue.enqueue(5);

    // Trying to add another element which should fail
```

```
        queue.enqueue(6);

        // Display current Queue
        queue.displayQueue();

        // Dequeue elements
        System.out.println("Dequeued: " + queue.dequeue());
        System.out.println("Dequeued: " + queue.dequeue());

        // Enqueue more elements
        queue.enqueue(6);
        queue.enqueue(7);

        // Display current Queue
        queue.displayQueue();

        // Continue dequeue to empty
        System.out.println("Dequeued: " + queue.dequeue());
        System.out.println("Dequeued: " + queue.dequeue());
        System.out.println("Dequeued: " + queue.dequeue());
        System.out.println("Dequeued: " + queue.dequeue());
        // Attempt to dequeue from empty queue
        System.out.println("Dequeued: " + queue.dequeue());
    }
}
```

Output

```
Queue is full
Queue contents: 1 2 3 4 5
Dequeued: 1
```

```
Dequeued: 2
Queue contents: 3 4 5 6 7
Dequeued: 3
Dequeued: 4
Dequeued: 5
Dequeued: 6
Dequeued: 7
Queue is empty
Dequeued: null
```

Compare To Methoid

```
public class Person implements Comparable<Person> {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public int compareTo(Person other) {
        return this.age - other.age;  // This will sort persons by age
    }

    @Override
    public String toString() {
        return name + ", age " + age;
    }
}
```