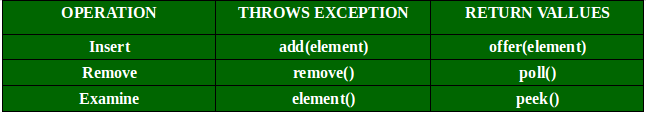
Queue Interface In Java

The Queue interface is available in java.util package and extends the Collection interface. The queue collection is used to hold the elements about to be processed and provides various operations like the insertion, removal etc. It is an ordered list of objects with its use limited to insert elements at the end of the list and deleting elements from the start of list i.e. it follows the FIFO or the First-In-First-Out principle. Being an interface the queue needs a concrete class for the declaration and the most common classes are the [PriorityQueue](https://www.geeksforgeeks.org/priority-queue-class-in-java-2/) and [LinkedList](https://www.geeksforgeeks.org/linked-list-in-java/" \t "_blank) in Java.It is to be noted that both the implementations are not thread safe. *PriorityBlockingQueue* is one alternative implementation if thread safe implementation is needed. Few important characteristics of Queue are:

* The Queue is used to insert elements at the end of the queue and removes from the beginning of the queue. It follows FIFO concept.
* The Java Queue supports all methods of Collection interface including insertion, deletion etc.
* [LinkedList](https://www.geeksforgeeks.org/linked-list-in-java/), ArrayBlockingQueue and [PriorityQueue](https://www.geeksforgeeks.org/priority-queue-class-in-java-2/) are the most frequently used implementations.
* If any null operation is performed on BlockingQueues, NullPointerException is thrown.
* BlockingQueues have thread-safe implementations.
* The Queues which are available in java.util package are Unbounded Queues
* The Queues which are available in java.util.concurrent package are the Bounded Queues.
* All Queues except the Deques supports insertion and removal at the tail and head of the queue respectively. The Deques support element insertion and removal at both ends.

**Methods in Queue:**

1. **add()-** This method is used to add elements at the tail of queue. More specifically, at the last of linked-list if it is used, or according to the priority in case of priority queue implementation.
2. **peek()-** This method is used to view the head of queue without removing it. It returns Null if the queue is empty.
3. **element()-** This method is similar to peek(). It throws *NoSuchElementException* when the queue is empty.
4. **remove()-** This method removes and returns the head of the queue. It throws *NoSuchElementException* when the queue is empty.
5. **poll()-** This method removes and returns the head of the queue. It returns null if the queue is empty.
6. **size()-** This method return the no. of elements in the queue.



Since it is a subtype of Collections class, it inherits all the methods of it namely *size(), isEmpty(), contains() etc.*  
Below is a simple Java program to demonstrate these methods:

|  |
| --- |
| // Java orogram to demonstrate working of Queue  // interface in Java  import java.util.LinkedList;  import java.util.Queue;    public class QueueExample  {    public static void main(String[] args)    {      Queue<Integer> q = new LinkedList<>();        // Adds elements {0, 1, 2, 3, 4} to queue      for (int i=0; i<5; i++)       q.add(i);        // Display contents of the queue.      System.out.println("Elements of queue-"+q);        // To remove the head of queue.      int removedele = q.remove();      System.out.println("removed element-" + removedele);        System.out.println(q);        // To view the head of queue      int head = q.peek();      System.out.println("head of queue-" + head);        // Rest all methods of collection interface,      // Like size and contains can be used with this      // implementation.      int size = q.size();      System.out.println("Size of queue-" + size);    }  } |

**Output:**

Elements of queue-[0, 1, 2, 3, 4]

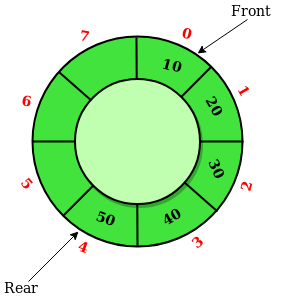
removed element-0

[1, 2, 3, 4]

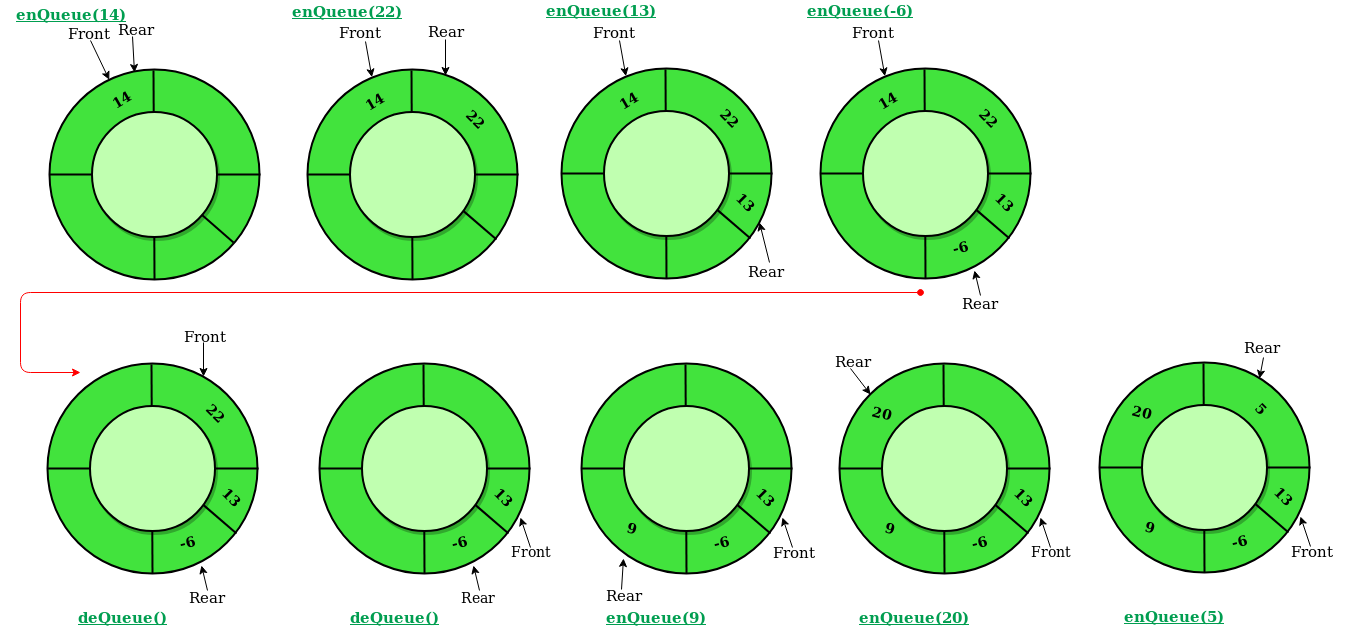
head of queue-1

Size of queue-4

Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle. It is also called **‘Ring Buffer’**.



In a normal Queue, we can insert elements until queue becomes full. But once queue becomes full, we can not insert the next element even if there is a space in front of queue.



Operations on Circular Queue:

* **Front:** Get the front item from queue.
* **Rear:** Get the last item from queue.
* **enQueue(value)**This function is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at Rear position.

**Steps:**

* 1. Check whether queue is Full – Check ((rear == SIZE-1 && front == 0) || (rear == front-1)).
  2. If it is full then display Queue is full. If queue is not full then, check if (rear == SIZE – 1 && front != 0) if it is true then set rear=0 and insert element.
* **deQueue()** This function is used to delete an element from the circular queue. In a circular queue, the element is always deleted from front position.

**Steps:**

* 1. Check whether queue is Empty means check (front==-1).
  2. If it is empty then display Queue is empty. If queue is not empty then step 3
  3. Check if (front==rear) if it is true then set front=rear= -1 else check if (front==size-1), if it is true then set front=0 and return the element.

**Time Complexity:** Time complexity of enQueue(), deQueue() operation is O(1) as there is no loop in any of the operation.

**Applications:**

1. **Memory Management:** The unused memory locations in the case of ordinary queues can be utilized in circular queues.
2. **Traffic system:** In computer controlled traffic system, circular queues are used to switch on the traffic lights one by one repeatedly as per the time set.
3. **CPU Scheduling:** Operating systems often maintain a queue of processes that are ready to execute or that are waiting for a particular event to occur.

class MyCircularQueue {

final int[] a;

int front, rear = -1, len = 0;

public MyCircularQueue(int k) { a = new int[k];}

public boolean enQueue(int val) {

if (!isFull()) {

rear = (rear + 1) % a.length;

a[rear] = val;

len++;

return true;

} else return false;

}

public boolean deQueue() {

if (!isEmpty()) {

front = (front + 1) % a.length;

len--;

return true;

} else return false;

}

public int Front() { return isEmpty() ? -1 : a[front];}

public int Rear() {return isEmpty() ? -1 : a[rear];}

public boolean isEmpty() { return len == 0;}

public boolean isFull() { return len == a.length;}

}