QUEUE INTERFACE IN JAVA(PART 2)

Classes that implement the Queue Interface:

1. PriorityQueue:

PriorityQueue class which is implemented in the collection framework provides us a way to process the objects based on the priority. It is known that a queue follows the First-In-First-Out algorithm, but sometimes the elements of the queue are needed to be processed according to the priority, that’s when the PriorityQueue comes into play

Example: Java program to demonstrate the creation of queue object using the PriorityQueue class

```
import java.util.*;


class GfG {


    public static void main(String args[])

    {

        // Creating empty priority queue

        Queue<Integer> pQueue

            = new PriorityQueue<Integer>();


        // Adding items to the pQueue

        // using add()

        pQueue.add(10);

        pQueue.add(20);
        pQueue.add(15);

        // Printing the top element of
        // the PriorityQueue
        System.out.println(pQueue.peek());

        // Printing the top element and removing it
        // from the PriorityQueue container
        System.out.println(pQueue.poll());

        // Printing the top element again
        System.out.println(pQueue.peek());
    }
}
```

Output

```
10
10
15
```

2. LinkedList:

LinkedList is a class which is implemented in the collection framework which inherently implements the linked list data structure. It is a linear data structure where the elements are not stored in contiguous locations and every element is a separate object with a data part and address part. The elements are linked using pointers and addresses. Each element is known as a node.

Due to the dynamicity and ease of insertions and deletions, they are preferred over the arrays or queues.

Example: Java program to demonstrate the creation of queue object using the LinkedList class

```
import java.util.*;


class GfG {


    public static void main(String args[])
    {
        // Creating empty LinkedList
        Queue<Integer> ll
            = new LinkedList<Integer>();


        // Adding items to the ll
        // using add()
        ll.add(10);
        ll.add(20);
        ll.add(15);


        // Printing the top element of
        // the LinkedList
        System.out.println(ll.peek());


        // Printing the top element and removing it
        // from the LinkedList container
        System.out.println(ll.poll());


        // Printing the top element again
        System.out.println(ll.peek());
    }
}
```

```
Output
10
10
20
```

3. PriorityBlockingQueue:

It is to be noted that both the implementations, the PriorityQueue and LinkedList are not thread-safe. PriorityBlockingQueue is one alternative implementation if thread-safe implementation is needed. PriorityBlockingQueue is an unbounded blocking queue that uses the same ordering rules as class PriorityQueue and supplies blocking retrieval operations.

Since it is unbounded, adding elements may sometimes fail due to resource exhaustion resulting in OutOfMemoryError.

Example: Java program to demonstrate the creation of queue object using the PriorityBlockingQueue class

```
import java.util.concurrent.PriorityBlockingQueue;
import java.util.*;
```

```
class GfG {

    public static void main(String args[])

    {

        // Creating empty priority

        // blocking queue

        Queue<Integer> pbq

            = new PriorityBlockingQueue<Integer>();


        // Adding items to the pbq

        // using add()

        pbq.add(10);

        pbq.add(20);

        pbq.add(15);


        // Printing the top element of

        // the PriorityBlockingQueue

        System.out.println(pbq.peek());


        // Printing the top element and

        // removing it from the

        // PriorityBlockingQueue

        System.out.println(pbq.poll());


        // Printing the top element again

        System.out.println(pbq.peek());
    }
}
```

Output

```
10
10
15
```

Methods of Queue Interface

The queue interface inherits all the methods present in the collections interface while implementing the following methods:

Method	Description
add(int index, element)	This method is used to add an element at a particular index in the queue. When a single parameter is passed, it simply adds the element at the end of the queue.
addAll(int index, Collection collection)	This method is used to add all the elements in the given collection to the queue. When a single parameter is passed, it adds all the elements of the given collection at the end of the queue.
size()	This method is used to return the size of the queue.
clear()	This method is used to remove all the elements in the queue. However, the reference of the queue created is still stored.

Method	Description
remove()	This method is used to remove the element from the front of the queue.
remove(int index)	This method removes an element from the specified index. It shifts subsequent elements(if any) to left and decreases their indexes by 1.
remove(element)	This method is used to remove and return the first occurrence of the given element in the queue.
get(int index)	This method returns elements at the specified index.
set(int index, element)	This method replaces elements at a given index with the new element. This function returns the element which was just replaced by a new element.
indexOf(element)	This method returns the first occurrence of the given element or -*1* if the element is not present in the queue.
lastIndexOf(element)	This method returns the last occurrence of the given element or -*1* if the element is not present in the queue.
equals(element)	This method is used to compare the equality of the given element with the elements of the queue.

Method	Description
hashCode()	This method is used to return the hashcode value of the given queue.
isEmpty()	This method is used to check if the queue is empty or not. It returns true if the queue is empty, else false.
contains(element)	This method is used to check if the queue contains the given element or not. It returns true if the queue contains the element.
containsAll(Collection collection)	This method is used to check if the queue contains all the collection of elements.
sort(Comparator comp)	This method is used to sort the elements of the queue on the basis of the given comparator.
boolean add(object)	This method is used to insert the specified element into a queue and return true upon success.
boolean offer(object)	This method is used to insert the specified element into the queue.
Object poll()	This method is used to retrieve and removes the head of the queue, or returns null if the queue is empty.

Method	Description
Object element()	This method is used to retrieves, but does not remove, the head of queue.
Object peek()	This method is used to retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.

Advantages of using the Queue Interface in Java

- Order preservation: The Queue interface provides a way to store and retrieve elements in a specific order, following the first-in, first-out (FIFO) principle.
- Flexibility: The Queue interface is a subtype of the Collection interface, which means that it can be used with many different data structures and algorithms, depending on the requirements of the application.
- Thread–safety: Some implementations of the Queue interface, such as the java.util.concurrent.ConcurrentLinkedQueue class, are thread-safe, which means that they can be accessed by multiple threads simultaneously without causing conflicts.
- Performance: The Queue interface provides efficient implementations for adding, removing, and inspecting elements, making it a useful tool for managing collections of elements in performance-critical applications.

Disadvantages of using the Queue Interface in Java

- Limited functionality: The Queue interface is designed specifically for managing collections of elements in a specific order, which means that it may not be suitable for more complex data structures or algorithms.
- Size restrictions: Some implementations of the Queue interface, such as the ArrayDeque class, have a fixed size, which means that they cannot grow beyond a certain number of elements.

- Memory usage: Depending on the implementation, the Queue interface may require more memory than other data structures, especially if it needs to store additional information about the order of the elements.
- Complexity: The Queue interface can be difficult to use and understand for novice programmers, especially if they are not familiar with the principles of data structures and algorithms.