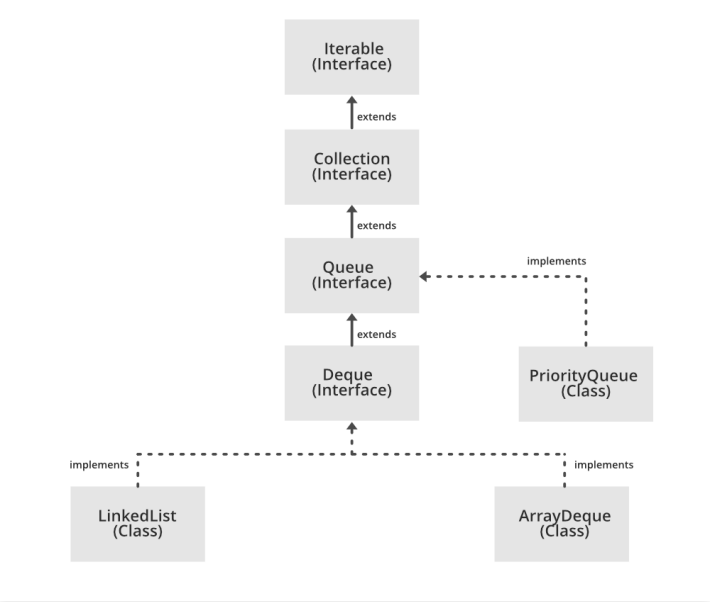
Queue Interface In Java

Difficulty Level : Easy Last Updated : 22 Nov, 2021

The Queue interface present in the java.util package and extends the Collection interface is used to hold the elements about to be processed in FIFO(First In First Out) order. 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 the 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 and LinkedList in Java. Note that neither of these implementations are thread safe. PriorityBlockingQueue is one alternative implementation if thread safe implementation is needed.

Declaration: The Queue interface is declared as:

public interface Queue extends Collection

Creating Queue Objects

Since *Queue* is an interface, objects cannot be created of the type queue. We always need a class which extends this list in order to create an object. And also, after the

introduction of Generics in Java 1.5, it is possible to restrict the type of object that can be stored in the Queue. This type-safe queue can be defined as:

```
// Obj is the type of the object to be stored in Queue
Queue<Obj> queue = new PriorityQueue<Obj> ();
```

Example of a Queue:

```
// Java program to demonstrate a Queue

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
        // the 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
```

Operations on Queue Interface

Let's see how to perform a few frequently used operations on the queue using the Priority Queue class.

1. Adding Elements: In order to add an element in a queue, we can use the add() method. The insertion order is not retained in the PriorityQueue. The elements are stored based on the priority order which is ascending by default.

```
// Java program to add elements
// to a Queue

import java.util.*;

public class GFG {

    public static void main(String args[])
    {
        Queue<String> pq = new PriorityQueue<>();

        pq.add("Geeks");
        pq.add("For");
```

```
        pq.add("Geeks");

        System.out.println(pq);
    }
}
```

Output:

```
[For, Geeks, Geeks]
```

2. Removing Elements: In order to remove an element from a queue, we can use the remove() method. If there are multiple such objects, then the first occurrence of the object is removed. Apart from that, poll() method is also used to remove the head and return it.

```
// Java program to remove elements
// from a Queue

import java.util.*;

public class GFG {

    public static void main(String args[])
    {
        Queue<String> pq = new PriorityQueue<>();

        pq.add("Geeks");
        pq.add("For");
        pq.add("Geeks");

        System.out.println("Initial Queue " + pq);

        pq.remove("Geeks");

        System.out.println("After Remove " + pq);

        System.out.println("Poll Method " + pq.poll());

        System.out.println("Final Queue " + pq);
    }
```

```
}
```

Output:

```
Initial Queue [For, Geeks, Geeks]
After Remove [For, Geeks]
Poll Method For
Final Queue [Geeks]
```

3. Iterating the Queue: There are multiple ways to iterate through the Queue. The most famous way is converting the queue to the array and traversing using the for loop. However, the queue also has an inbuilt iterator which can be used to iterate through the queue.

```
// Java program to iterate elements
// to a Queue

import java.util.*;

public class GFG {

    public static void main(String args[])
    {
        Queue<String> pq = new PriorityQueue<>();

        pq.add("Geeks");
        pq.add("For");
        pq.add("Geeks");

        Iterator iterator = pq.iterator();

        while (iterator.hasNext()) {
            System.out.print(iterator.next() + " ");
        }
    }
}
```

Output:

```
For Geeks Geeks
```

Characteristics of a Queue: The following are the characteristics of the queue:

- 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, ArrayBlockingQueue and PriorityQueue are the most frequently used implementations.
- If any null operation is performed on BlockingQueues, NullPointerException is thrown.
- 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.

Classes which 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 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. Let's see how to create a queue object using this class.

```
// 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. Let's see how to create a queue object using this class.

```
// 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. Let's see how to create a queue object using this class.

```
// 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 list. When a single parameter is passed, it simply adds the element at the end of the list.
addAll(int index, Collection collection)	This method is used to add all the elements in the given collection to the list. When a single parameter is passed, it adds all the elements of the given collection at the end of the list.
size()	This method is used to return the size of the list.
clear()	This method is used to remove all the elements in the list. However, the reference of the list created is still stored.
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 the first occurrence of the given element in the list.
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 list.
lastIndexOf(element)	This method returns the last occurrence of the given element or *-1* if the element is not present in the list.

Method	Description
equals(element)	This method is used to compare the equality of the given element with the elements of the list.
hashCode()	This method is used to return the hashcode value of the given list.
isEmpty()	This method is used to check if the list is empty or not. It returns true if the list is empty, else false.
contains(element)	This method is used to check if the list contains the given element or not. It returns true if the list contains the element.
containsAll(Collection collection)	This method is used to check if the list contains all the collection of elements.
sort(Comparator comp)	This method is used to sort the elements of the list on the basis of the given comparator.