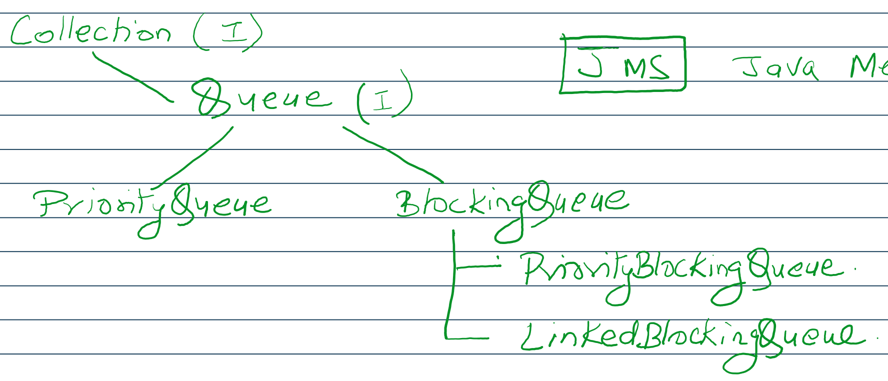
Enhancement in Collections

10 June 2025 18:38



$\boxed{\text{JMS}}$ Java Message Service

Queue interface

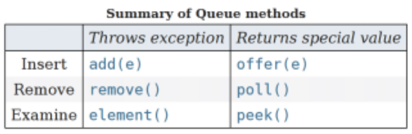
- It is a child interface of Collection.
- If we want to represent a group of individual objects "Prior to process". Then we should go for Queue interface. For example before sending SMS message all mobile numbers we have to store in some data structure. In which order we add mobile number in the same order only message should be delivered. For this FIFO requirement, Queue is the best choice.
- Queue follows FIFO, but based on our requirement we can implement our own priority order also. (Priority queue).
- From. 1.5 version onwards. LinkedList class also implements Queue interface. LinkedList based implementation of Queue always follow FIFO.

Queue (I) Specific methods :-

1. boolean offer (Object o) ← to add object into the queue.
2. Object poll ( ) ← remove and return head element of queue.
3. Object peek ( ) ← it returns the head element of queue. if queue is empty then this method returns null.
4. Object element( ) ← it returns the head element of queue. If queue is empty then this method raises runtime exception i.e, "NoSuchElementException"
5. Object remove ( ) ← it returns the head element of the queue. If queue is empty then this method raises runtime exception i.e "NoSuchElementException"

Official Documentation:

public interface Queue<E>
extends Collection<E>
A collection designed for holding elements prior to processing. Besides basic Collection operations, queues provide additional insertion, extraction, and inspection operations. Each of these methods exists in two forms: one throws an exception if the operation fails, the other returns a special value (either null or false, depending on the operation). The latter form of the insert operation is designed specifically for use with capacity-restricted Queue implementations; in most implementations, insert operations cannot fail.



Summary of Queue methods

	Throws exception	Returns special value
Insert	add(e)	offer(e)
Remove	remove()	poll()
Examine	element()	peek()

Queues typically, but do not necessarily, order elements in a FIFO (first-in-first-out) manner. Among the exceptions are priority queues, which order elements according to a supplied comparator, or the elements' natural ordering, and LIFO queues (or stacks) which order the elements LIFO (last-in-first-out). Whatever the ordering used, the head of the queue is that element which would be removed by a call to remove() or poll(). In a FIFO queue, all new elements are inserted at the tail of the queue. Other kinds of queues may use different placement rules. Every Queue implementation must specify its ordering properties.

The offer method inserts an element if possible, otherwise returning false. This differs from the Collection.add method, which can fail to add an element only by throwing an unchecked exception. The offer method is designed for use when failure is a normal, rather than exceptional occurrence, for example, in fixed-capacity (or "bounded") queues.

The remove() and poll() methods remove and return the head of the queue. Exactly which element is removed from the queue is a function of the queue's ordering policy, which differs from implementation to implementation. The remove() and poll() methods differ only in their behavior when the queue is empty: the remove() method throws an exception, while the poll() method returns null.

The element() and peek() methods return, but do not remove, the head of the queue.

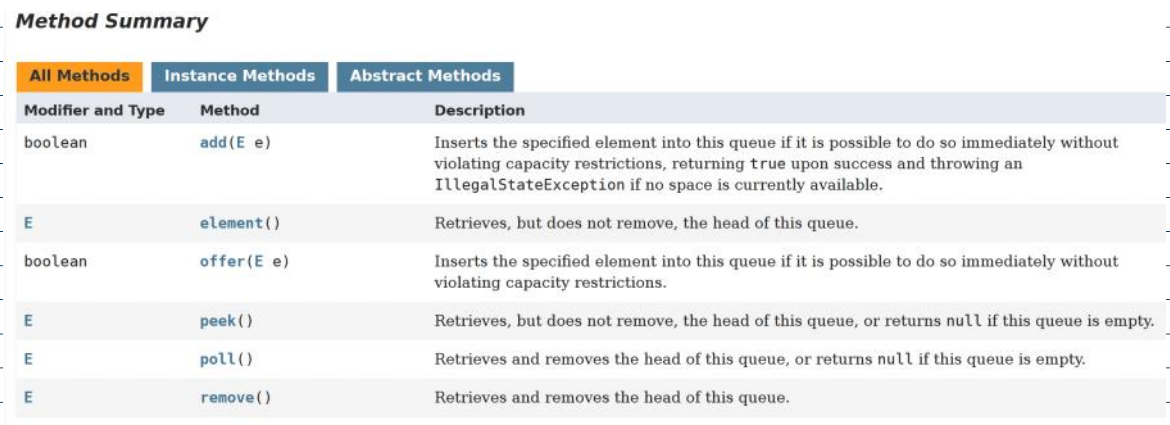
The Queue interface does not define the blocking queue methods, which are common in concurrent programming. These methods, which wait for elements to appear or for space to become available, are defined in the BlockingQueue interface, which extends this interface.

Queue implementations generally do not allow insertion of null elements, although some implementations, such as LinkedList, do not prohibit insertion of null. Even in the implementations that permit it, null should not be inserted into a Queue, as null is also used as a special return value by the poll method to indicate that the queue contains no elements.

Queue implementations generally do not define element-based versions of methods equals and hashCode but instead inherit the identity based versions from class Object, because element-based equality is not always well-defined for queues with the same elements but different ordering properties.

This interface is a member of the Java Collections Framework.

Since:
    1.5



Method Summary

Modifier and Type	Method	Description
boolean	add(E e)	Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions, returning true upon success and throwing an IllegalStateException if no space is currently available.
E	element()	Retrieves, but does not remove, the head of this queue.
boolean	offer(E e)	Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions.
E	peek()	Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.
E	poll()	Retrieves and removes the head of this queue, or returns null if this queue is empty.
E	remove()	Retrieves and removes the head of this queue.

Priority Queue

- **If we want to represent a group of individual objects prior to processing according to some** priority then we should go for priority queue.

- The priority can be either *default natural sorting order* Or *customized sorting order* defined by Comparator.
- Insertion order is not preserved and it is based on some priority.
- If we are depending on default natural sorting order, it is compulsory object should be homogeneous and Comparable otherwise we will get runtime exception saying *"ClassCastException"*.
- If we are defining our own sorting by Comparator, then objects need not be homogeneous and Comparable.
- Null is not allowed at all.

Constructors

PriorityQueue pq = new PriorityQueue(); ==> It creates an empty priority queue with default initial capacity 11 and all objects is inserted according to default natural sorting order.

PriortyQueue pq = new PriorityQueue(int initialCapacity);

PriortyQueue pq = new PriorityQueue(int initialCapacity, Comparator c);

PriortyQueue pq = new PriorityQueue(Comparator c);

PriortyQueue pq = new PriorityQueue(SortedSet s);

PriortyQueue pq = new PriorityQueue(Collection c); } ← interconversion constructor.

Official Documentation:

public class PriorityQueue<E>
extends AbstractQueue<E>
implements Serializable

An unbounded priority queue based on a priority heap. The elements of the priority queue are ordered according to their natural ordering, or by a Comparator provided at queue construction time, depending on which constructor is used. A priority queue does not permit null elements. A priority queue relying on natural ordering also does not permit insertion of non-comparable objects (doing so may result in ClassCastException).

The head of this queue is the least element with respect to the specified ordering. If multiple elements are tied for least value, the head is one of those elements -- ties are broken arbitrarily. The queue retrieval operations poll, remove, peek, and element access the element at the head of the queue.

A priority queue is unbounded, but has an internal capacity governing the size of an array used to store the elements on the queue. It is always at least as large as the queue size. As elements are added to a priority queue, its capacity grows automatically. The details of the growth policy are not specified.

This class and its iterator implement all of the optional methods of the Collection and Iterator interfaces. The Iterator provided in method iterator() and the Spliterator provided in method spliterator() are not guaranteed to traverse the elements of the priority queue in any particular order. If you need ordered traversal, consider using Arrays.sort(pq.toArray()).

Note that this implementation is not synchronized. Multiple threads should not access a PriorityQueue instance concurrently if any of the threads modifies the queue. Instead, use the thread-safe PriorityBlockingQueue class.

Implementation note: this implementation provides O(log(n)) time for the enqueuing and dequeuing methods (offer, poll, remove() and add); linear time for the remove(Object) and contains(Object) methods; and constant time for the retrieval methods (peek, element, and size).

This class is a member of the Java Collections Framework.

Since:
    1.5
https://docs.oracle.com/en/java/javase/24/docs/api//java.base/java/util/PriorityQueue.html

To know more about methods consider the above link.

```java
import java.util.PriorityQueue;
public class PriorityQueueDemo{
    public static void main(String[] args) {
        PriorityQueue pq = new PriorityQueue();
        System.out.println(pq.peek()); //null
        System.out.println(pq.element()); //NoSuchElementException
        for (int i = 0; i < 11; i++) {
            pq.offer(i);
        }
        System.out.println(pq);
    }
}
```

Output:

```
null
Exception in thread "main" java.util.NoSuchElementException
        at java.base/java.util.AbstractQueue.element(AbstractQueue.java:136)
        at PriorityQueueDemo.main(PriorityQueueDemo.java:7)
```

```java
import java.util.PriorityQueue;
import java.util.Random;
import java.util.Scanner;
public class PriorityQueueDemo{
    public static void main(String[] args) {
        PriorityQueue pq = new PriorityQueue();
        System.out.println(pq.peek()); //null
        //System.out.println(pq.element()); //NoSuchElementException
        for (int i = 0; i < 11; i++) {
            int x = new Random().nextInt(10, 100);
            System.out.println(x);
            pq.offer(x);
        }
        System.out.println(pq);
        while(!pq.isEmpty()){
            System.out.println("Press y/n to poll elements from priority queue: ");
            String x = new Scanner(System.in).next().toLowerCase();
            if (x.charAt(0) == 'n'){
                break;
            }
            System.out.println(pq.poll());
        }
    }
}
```

Output:

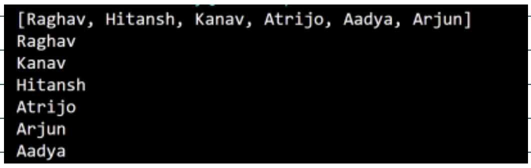
```
null
33
91
87
48
43
83
14
73
18
73
37
[14, 18, 33, 43, 37, 87, 83, 91, 73, 73, 48]
Press y/n to poll elements from priority queue:
y
14
Press y/n to poll elements from priority queue:
y
18
Press y/n to poll elements from priority queue:
y
33
Press y/n to poll elements from priority queue:
y
37
Press y/n to poll elements from priority queue:
y
43
Press y/n to poll elements from priority queue:
y
48
Press y/n to poll elements from priority queue:
y
73
Press y/n to poll elements from priority queue:
y
73
Press y/n to poll elements from priority queue:
y
83
Press y/n to poll elements from priority queue:
y
87
Press y/n to poll elements from priority queue:
y
91
```

Example with Comparator

```java
import java.util.Comparator;
public class Mycomparator implements Comparator{
    @Override
    public int compare(Object o1, Object o2) {
        return o2.toString().compareTo(o2.toString());
    }
}
```

```java
import java.util.PriorityQueue;
public class PriorityQueueComparatorDemo {
    public static void main(String[] args) {
        PriorityQueue pq = new PriorityQueue<>(15, new MyCompartor());
        pq.add("Atrijo");
        pq.add("Raghav");
        pq.add("Arjun");
        pq.add("Hitansh");
        pq.add("Aadya");
        pq.add("Kanav");
        System.out.println(pq);
        while(!pq.isEmpty()){
            System.out.println(pq.poll());
        }
    }
}
```

Output:



```
[Raghav, Hitansh, Kanav, Atrijo, Aadya, Arjun]
Raghav
Kanav
Hitansh
Atrijo
Arjun
Aadya
```