Module java.base
Package java.util.concurrent

Class PriorityBlockingQueue<E>

java.lang.Object
    java.util.AbstractCollection<E>
        java.util.AbstractQueue<E>
            java.util.concurrent.PriorityBlockingQueue<E>

Type Parameters:

E - the type of elements held in this queue

All Implemented Interfaces:

Serializable, Iterable<E>, Collection<E>, BlockingQueue<E>, Queue<E>

```
public class PriorityBlockingQueue<E>
extends AbstractQueue<E>
implements BlockingQueue<E>, Serializable
```

An unbounded blocking queue that uses the same ordering rules as class `PriorityQueue` and supplies blocking retrieval operations. While this queue is logically unbounded, attempted additions may fail due to resource exhaustion (causing `OutOfMemoryError`). This class does not permit `null` elements. A priority queue relying on natural ordering also does not permit insertion of non-comparable objects (doing so results in `ClassCastException`).

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 PriorityBlockingQueue in any particular order. If you need ordered traversal, consider using `Arrays.sort(pq.toArray())`. Also, method `drainTo` can be used to *remove* some or all elements in priority order and place them in another collection.

Operations on this class make no guarantees about the ordering of elements with equal priority. If you need to enforce an ordering, you can define custom classes or comparators that use a secondary key to break ties in primary priority values. For example, here is a class that applies first-in-first-out tie-breaking to comparable elements. To use it, you would insert a `new FIFOEntry(anEntry)` instead of a plain entry object.

```
class FIFOEntry<E extends Comparable<? super E>>
    implements Comparable<FIFOEntry<E>> {
  static final AtomicLong seq = new AtomicLong();
  final long seqNum;
  final E entry;
  public FIFOEntry(E entry) {
    seqNum = seq.getAndIncrement();
    this.entry = entry;
  }
  public E getEntry() { return entry; }
  public int compareTo(FIFOEntry<E> other) {
    int res = entry.compareTo(other.entry);
    if (res == 0 && other.entry != this.entry)
```

```
        res = (seqNum < other.seqNum ? -1 : 1);
      return res;
    }
  }
```

This class is a member of the Java Collections Framework.

Since:

1.5

See Also:

Serialized Form

Constructor Summary

Constructors

Constructor	Description
`PriorityBlockingQueue()`	Creates a `PriorityBlockingQueue` with the default initial capacity (11) that orders its elements according to their natural ordering.
`PriorityBlockingQueue(int initialCapacity)`	Creates a `PriorityBlockingQueue` with the specified initial capacity that orders its elements according to their natural ordering.
`PriorityBlockingQueue(int initialCapacity, Comparator<? super E> comparator)`	Creates a `PriorityBlockingQueue` with the specified initial capacity that orders its elements according to the specified comparator.
`PriorityBlockingQueue(Collection<? extends E> c)`	Creates a `PriorityBlockingQueue` containing the elements in the specified collection.

Method Summary

All Methods Instance Methods Concrete Methods

Modifier and Type	Method	Description
`boolean`	`add(E e)`	Inserts the specified element into this priority queue.
`void`	`clear()`	Atomically removes all of the elements from this queue.
`Comparator<? super E>`	`comparator()`	Returns the comparator used to order the elements in this queue, or `null` if this queue

		uses the natural ordering of its elements.
boolean	**contains**(**Object** o)	Returns `true` if this queue contains the specified element.
int	**drainTo**(**Collection**<? super **E**> c)	Removes all available elements from this queue and adds them to the given collection.
int	**drainTo**(**Collection**<? super **E**> c, int maxElements)	Removes at most the given number of available elements from this queue and adds them to the given collection.
void	**forEach**(**Consumer**<? super **E**> action)	Performs the given action for each element of the `Iterable` until all elements have been processed or the action throws an exception.
Iterator<**E**>	**iterator**()	Returns an iterator over the elements in this queue.
boolean	**offer**(**E** e)	Inserts the specified element into this priority queue.
boolean	**offer**(**E** e, long timeout, **TimeUnit** unit)	Inserts the specified element into this priority queue.
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	**poll**(long timeout, **TimeUnit** unit)	Retrieves and removes the head of this queue, waiting up to the specified wait time if necessary for an element to become available.
void	**put**(**E** e)	Inserts the specified element into this priority queue.
int	**remainingCapacity**()	Always returns `Integer.MAX_VALUE` because

		a PriorityBlockingQueue is not capacity constrained.
boolean	**remove**(**Object** o)	Removes a single instance of the specified element from this queue, if it is present.
boolean	**removeAll**(**Collection**<?> c)	Removes all of this collection's elements that are also contained in the specified collection (optional operation).
boolean	**removeIf**(**Predicate**<? super **E**> filter)	Removes all of the elements of this collection that satisfy the given predicate.
boolean	**retainAll**(**Collection**<?> c)	Retains only the elements in this collection that are contained in the specified collection (optional operation).
int	**size**()	Returns the number of elements in this collection.
Spliterator<**E**>	**spliterator**()	Returns a Spliterator over the elements in this queue.
E	**take**()	Retrieves and removes the head of this queue, waiting if necessary until an element becomes available.
Object[]	**toArray**()	Returns an array containing all of the elements in this queue.
<T> T[]	**toArray**(T[] a)	Returns an array containing all of the elements in this queue; the runtime type of the returned array is that of the specified array.

Methods declared in class java.util.AbstractQueue

addAll, element, remove

Methods declared in class java.util.AbstractCollection

containsAll, isEmpty, toString

Methods declared in class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Methods declared in interface java.util.Collection

addAll, containsAll, equals, hashCode, isEmpty, parallelStream, stream, toArray

Methods declared in interface java.util.Queue

element, remove

Constructor Details

PriorityBlockingQueue

```
public PriorityBlockingQueue()
```

Creates a PriorityBlockingQueue with the default initial capacity (11) that orders its elements according to their natural ordering.

PriorityBlockingQueue

```
public PriorityBlockingQueue(int initialCapacity)
```

Creates a PriorityBlockingQueue with the specified initial capacity that orders its elements according to their natural ordering.

Parameters:

initialCapacity - the initial capacity for this priority queue

Throws:

IllegalArgumentException - if initialCapacity is less than 1

PriorityBlockingQueue

```
public PriorityBlockingQueue(int initialCapacity,
                             Comparator<? super E> comparator)
```

Creates a PriorityBlockingQueue with the specified initial capacity that orders its elements according to the specified comparator.

Parameters:

initialCapacity - the initial capacity for this priority queue

comparator - the comparator that will be used to order this priority queue. If null, the natural ordering of the elements will be used.

Throws:

IllegalArgumentException - if initialCapacity is less than 1

PriorityBlockingQueue

```
public PriorityBlockingQueue(Collection<? extends E> c)
```

Creates a `PriorityBlockingQueue` containing the elements in the specified collection. If the specified collection is a `SortedSet` or a `PriorityBlockingQueue`, this priority queue will be ordered according to the same ordering. Otherwise, this priority queue will be ordered according to the natural ordering of its elements.

Parameters:

`c` - the collection whose elements are to be placed into this priority queue

Throws:

`ClassCastException` - if elements of the specified collection cannot be compared to one another according to the priority queue's ordering

`NullPointerException` - if the specified collection or any of its elements are null

Method Details

add

```
public boolean add(E e)
```

Inserts the specified element into this priority queue.

Specified by:

`add` in interface `BlockingQueue<E>`

Specified by:

`add` in interface `Collection<E>`

Specified by:

`add` in interface `Queue<E>`

Overrides:

`add` in class `AbstractQueue<E>`

Parameters:

`e` - the element to add

Returns:

`true` (as specified by `Collection.add(E)`)

Throws:

`ClassCastException` - if the specified element cannot be compared with elements currently in the priority queue according to the priority queue's ordering

`NullPointerException` - if the specified element is null

offer

```
public boolean offer(E e)
```

Inserts the specified element into this priority queue. As the queue is unbounded, this method will never return `false`.

Specified by:

`offer` in interface `BlockingQueue<E>`

Specified by:

`offer` in interface `Queue<E>`

Parameters:

`e` - the element to add

Returns:

`true` (as specified by `Queue.offer(E)`)

Throws:

`ClassCastException` - if the specified element cannot be compared with elements currently in the priority queue according to the priority queue's ordering

`NullPointerException` - if the specified element is null

put

```
public void put(E e)
```

Inserts the specified element into this priority queue. As the queue is unbounded, this method will never block.

Specified by:

`put` in interface `BlockingQueue<E>`

Parameters:

`e` - the element to add

Throws:

`ClassCastException` - if the specified element cannot be compared with elements currently in the priority queue according to the priority queue's ordering

`NullPointerException` - if the specified element is null

offer

```
public boolean offer(E e,
                     long timeout,
                     TimeUnit unit)
```

Inserts the specified element into this priority queue. As the queue is unbounded, this method will never block or return `false`.

Specified by:

`offer` in interface `BlockingQueue<E>`

Parameters:

`e` - the element to add

`timeout` - This parameter is ignored as the method never blocks

`unit` - This parameter is ignored as the method never blocks

Returns:

`true` (as specified by `BlockingQueue.offer`)

Throws:

`ClassCastException` - if the specified element cannot be compared with elements currently in the priority queue according to the priority queue's ordering

`NullPointerException` - if the specified element is null

poll

```
public E poll()
```

Description copied from interface: `Queue`

Retrieves and removes the head of this queue, or returns `null` if this queue is empty.

Specified by:

`poll` in interface `Queue<E>`

Returns:

the head of this queue, or `null` if this queue is empty

take

```
public E take()
       throws InterruptedException
```

Description copied from interface: `BlockingQueue`

Retrieves and removes the head of this queue, waiting if necessary until an element becomes available.

Specified by:

`take` in interface `BlockingQueue<E>`

Returns:

the head of this queue

Throws:

`InterruptedException` - if interrupted while waiting

poll

```
public E poll(long timeout,
              TimeUnit unit)
       throws InterruptedException
```

Description copied from interface: `BlockingQueue`

Retrieves and removes the head of this queue, waiting up to the specified wait time if necessary for an element to become available.

Specified by:

`poll` in interface `BlockingQueue<E>`

Parameters:

`timeout` - how long to wait before giving up, in units of `unit`

`unit` - a `TimeUnit` determining how to interpret the `timeout` parameter

Returns:

the head of this queue, or `null` if the specified waiting time elapses before an element is available

Throws:

`InterruptedException` - if interrupted while waiting

peek

```
public E peek()
```

Description copied from interface: `Queue`

Retrieves, but does not remove, the head of this queue, or returns `null` if this queue is empty.

Specified by:

`peek` in interface `Queue<E>`

Returns:

the head of this queue, or `null` if this queue is empty

comparator

```
public Comparator<? super E> comparator()
```

Returns the comparator used to order the elements in this queue, or `null` if this queue uses the natural ordering of its elements.

Returns:

the comparator used to order the elements in this queue, or `null` if this queue uses the natural ordering of its elements

size

```
public int size()
```

Description copied from interface: `Collection`

Returns the number of elements in this collection. If this collection contains more than `Integer.MAX_VALUE` elements, returns `Integer.MAX_VALUE`.

Specified by:

`size` in interface `Collection<E>`

Returns:

the number of elements in this collection

remainingCapacity

```
public int remainingCapacity()
```

Always returns `Integer.MAX_VALUE` because a `PriorityBlockingQueue` is not capacity constrained.

Specified by:

`remainingCapacity` in interface `BlockingQueue<E>`

Returns:

`Integer.MAX_VALUE` always

remove

```
public boolean remove(Object o)
```

Removes a single instance of the specified element from this queue, if it is present. More formally, removes an element `e` such that `o.equals(e)`, if this queue contains one or more such elements. Returns `true` if and only if this queue contained the specified element (or equivalently, if this queue changed as a result of the call).

Specified by:

`remove` in interface `BlockingQueue<E>`

Specified by:

`remove` in interface `Collection<E>`

Overrides:

`remove` in class `AbstractCollection<E>`

Parameters:

`o` - element to be removed from this queue, if present

Returns:

`true` if this queue changed as a result of the call

contains

```
public boolean contains(Object o)
```

Returns `true` if this queue contains the specified element. More formally, returns `true` if and only if this queue contains at least one element `e` such that `o.equals(e)`.

Specified by:

`contains` in interface `BlockingQueue<E>`

Specified by:

`contains` in interface `Collection<E>`

Overrides:

`contains` in class `AbstractCollection<E>`

Parameters:

`o` - object to be checked for containment in this queue

Returns:

`true` if this queue contains the specified element

drainTo

```
public int drainTo(Collection<? super E> c)
```

Description copied from interface: `BlockingQueue`

Removes all available elements from this queue and adds them to the given collection. This operation may be more efficient than repeatedly polling this queue. A failure encountered while attempting to add elements to collection `c` may result in elements being in neither, either or both collections when the associated exception is thrown. Attempts to drain a queue to itself result in `IllegalArgumentException`. Further, the behavior of this operation is undefined if the specified collection is modified while the operation is in progress.

Specified by:

`drainTo` in interface `BlockingQueue<E>`

Parameters:

`c` - the collection to transfer elements into

Returns:

the number of elements transferred

Throws:

`UnsupportedOperationException` - if addition of elements is not supported by the specified collection

`ClassCastException` - if the class of an element of this queue prevents it from being added to the specified collection

`NullPointerException` - if the specified collection is null

`IllegalArgumentException` - if the specified collection is this queue, or some property of an element of this queue prevents it from being added to the specified collection

drainTo

```
public int drainTo(Collection<? super E> c,
                   int maxElements)
```

Description copied from interface: `BlockingQueue`

Removes at most the given number of available elements from this queue and adds them to the given collection. A failure encountered while attempting to add elements to collection `c` may result in elements being in neither, either or both collections when the

associated exception is thrown. Attempts to drain a queue to itself result in `IllegalArgumentException`. Further, the behavior of this operation is undefined if the specified collection is modified while the operation is in progress.

Specified by:

`drainTo` in interface `BlockingQueue<E>`

Parameters:

`c` - the collection to transfer elements into

`maxElements` - the maximum number of elements to transfer

Returns:

the number of elements transferred

Throws:

`UnsupportedOperationException` - if addition of elements is not supported by the specified collection

`ClassCastException` - if the class of an element of this queue prevents it from being added to the specified collection

`NullPointerException` - if the specified collection is null

`IllegalArgumentException` - if the specified collection is this queue, or some property of an element of this queue prevents it from being added to the specified collection

clear

```
public void clear()
```

Atomically removes all of the elements from this queue. The queue will be empty after this call returns.

Specified by:

`clear` in interface `Collection<E>`

Overrides:

`clear` in class `AbstractQueue<E>`

toArray

```
public Object[] toArray()
```

Returns an array containing all of the elements in this queue. The returned array elements are in no particular order.

The returned array will be "safe" in that no references to it are maintained by this queue. (In other words, this method must allocate a new array). The caller is thus free to modify the returned array.

This method acts as bridge between array-based and collection-based APIs.

Specified by:

`toArray` in interface `Collection<E>`

Overrides:

`toArray` in class `AbstractCollection<E>`

Returns:

an array containing all of the elements in this queue

toArray

```
public <T> T[] toArray(T[] a)
```

Returns an array containing all of the elements in this queue; the runtime type of the returned array is that of the specified array. The returned array elements are in no particular order. If the queue fits in the specified array, it is returned therein. Otherwise, a new array is allocated with the runtime type of the specified array and the size of this queue.

If this queue fits in the specified array with room to spare (i.e., the array has more elements than this queue), the element in the array immediately following the end of the queue is set to `null`.

Like the `toArray()` method, this method acts as bridge between array-based and collection-based APIs. Further, this method allows precise control over the runtime type of the output array, and may, under certain circumstances, be used to save allocation costs.

Suppose `x` is a queue known to contain only strings. The following code can be used to dump the queue into a newly allocated array of `String`:

```
 String[] y = x.toArray(new String[0]);
```

Note that `toArray(new Object[0])` is identical in function to `toArray()`.

Specified by:

`toArray` in interface `Collection<E>`

Overrides:

`toArray` in class `AbstractCollection<E>`

Type Parameters:

`T` - the component type of the array to contain the collection

Parameters:

`a` - the array into which the elements of the queue are to be stored, if it is big enough; otherwise, a new array of the same runtime type is allocated for this purpose

Returns:

an array containing all of the elements in this queue

Throws:

`ArrayStoreException` - if the runtime type of the specified array is not a supertype of the runtime type of every element in this queue

`NullPointerException` - if the specified array is null

iterator

```
public Iterator<E> iterator()
```

Returns an iterator over the elements in this queue. The iterator does not return the elements in any particular order.

The returned iterator is *weakly consistent*.

Specified by:

`iterator` in interface `Collection<E>`

Specified by:

`iterator` in interface `Iterable<E>`

Specified by:

`iterator` in class `AbstractCollection<E>`

Returns:

an iterator over the elements in this queue

spliterator

```
public Spliterator<E> spliterator()
```

Returns a `Spliterator` over the elements in this queue. The spliterator does not traverse elements in any particular order (the `ORDERED` characteristic is not reported).

The returned spliterator is *weakly consistent*.

The `Spliterator` reports `Spliterator.SIZED` and `Spliterator.NONNULL`.

Specified by:

`spliterator` in interface `Collection<E>`

Specified by:

`spliterator` in interface `Iterable<E>`

Implementation Note:

The `Spliterator` additionally reports `Spliterator.SUBSIZED`.

Returns:

a `Spliterator` over the elements in this queue

Since:

1.8

removeIf

```
public boolean removeIf(Predicate<? super E> filter)
```

Description copied from interface: `Collection`

Removes all of the elements of this collection that satisfy the given predicate. Errors or runtime exceptions thrown during iteration or by the predicate are relayed to the caller.

Specified by:

`removeIf` in interface `Collection<E>`

Parameters:

`filter` - a predicate which returns `true` for elements to be removed

Returns:

`true` if any elements were removed

Throws:

`NullPointerException` - if the specified filter is null

removeAll

```
public boolean removeAll(Collection<?> c)
```

Description copied from class: `AbstractCollection`

Removes all of this collection's elements that are also contained in the specified collection (optional operation). After this call returns, this collection will contain no elements in common with the specified collection.

Specified by:

`removeAll` in interface `Collection<E>`

Overrides:

`removeAll` in class `AbstractCollection<E>`

Parameters:

`c` - collection containing elements to be removed from this collection

Returns:

`true` if this collection changed as a result of the call

Throws:

`NullPointerException` - if this collection contains one or more null elements and the specified collection does not support null elements (optional), or if the specified collection is null

See Also:

`AbstractCollection.remove(Object)`, `AbstractCollection.contains(Object)`

retainAll

```
public boolean retainAll(Collection<?> c)
```

Description copied from class: `AbstractCollection`

Retains only the elements in this collection that are contained in the specified collection (optional operation). In other words, removes from this collection all of its elements that are not contained in the specified collection.

Specified by:

`retainAll` in interface `Collection<E>`

Overrides:

`retainAll` in class `AbstractCollection<E>`

Parameters:

`c` - collection containing elements to be retained in this collection

Returns:

`true` if this collection changed as a result of the call

Throws:

`NullPointerException` - if this collection contains one or more null elements and the specified collection does not permit null elements (optional), or if the specified collection is null

See Also:

`AbstractCollection.remove(Object)`, `AbstractCollection.contains(Object)`

forEach

```
public void forEach(Consumer<? super E> action)
```

Description copied from interface: `Iterable`

Performs the given action for each element of the `Iterable` until all elements have been processed or the action throws an exception. Actions are performed in the order of iteration, if that order is specified. Exceptions thrown by the action are relayed to the caller.

The behavior of this method is unspecified if the action performs side-effects that modify the underlying source of elements, unless an overriding class has specified a concurrent modification policy.

Specified by:

`forEach` in interface `Iterable<E>`

Parameters:

`action` - The action to be performed for each element

Throws:

`NullPointerException` - if the specified action is null