Module java.base
Package java.util.concurrent

Interface BlockingDeque<E>

Type Parameters:

E - the type of elements held in this deque

All Superinterfaces:

`BlockingQueue<E>`, `Collection<E>`, `Deque<E>`, `Iterable<E>`, `Queue<E>`

All Known Implementing Classes:

`LinkedBlockingDeque`

```
public interface BlockingDeque<E>
extends BlockingQueue<E>, Deque<E>
```

A `Deque` that additionally supports blocking operations that wait for the deque to become non-empty when retrieving an element, and wait for space to become available in the deque when storing an element.

`BlockingDeque` methods come in four forms, with different ways of handling operations that cannot be satisfied immediately, but may be satisfied at some point in the future: one throws an exception, the second returns a special value (either `null` or `false`, depending on the operation), the third blocks the current thread indefinitely until the operation can succeed, and the fourth blocks for only a given maximum time limit before giving up. These methods are summarized in the following table:

Summary of BlockingDeque methods

First Element (Head)				
	Throws exception	*Special value*	*Blocks*	*Times out*
Insert	`addFirst(e)`	`offerFirst(e)`	`putFirst(e)`	`offerFirst(e, time, unit)`
Remove	`removeFirst()`	`pollFirst()`	`takeFirst()`	`pollFirst(time, unit)`
Examine	`getFirst()`	`peekFirst()`	*not applicable*	*not applicable*
Last Element (Tail)				
	Throws exception	*Special value*	*Blocks*	*Times out*
Insert	`addLast(e)`	`offerLast(e)`	`putLast(e)`	`offerLast(e, time, unit)`
Remove	`removeLast()`	`pollLast()`	`takeLast()`	`pollLast(time, unit)`
Examine	`getLast()`	`peekLast()`	*not applicable*	*not applicable*

Like any `BlockingQueue`, a `BlockingDeque` is thread safe, does not permit null elements, and may (or may not) be capacity-constrained.

A `BlockingDeque` implementation may be used directly as a FIFO `BlockingQueue`. The methods inherited from the `BlockingQueue` interface are precisely equivalent to

BlockingDeque methods as indicated in the following table:

Comparison of BlockingQueue and BlockingDeque methods

	BlockingQueue Method	**Equivalent BlockingDeque Method**
Insert	`add(e)`	`addLast(e)`
	`offer(e)`	`offerLast(e)`
	`put(e)`	`putLast(e)`
	`offer(e, time, unit)`	`offerLast(e, time, unit)`
Remove	`remove()`	`removeFirst()`
	`poll()`	`pollFirst()`
	`take()`	`takeFirst()`
	`poll(time, unit)`	`pollFirst(time, unit)`
Examine	`element()`	`getFirst()`
	`peek()`	`peekFirst()`

Memory consistency effects: As with other concurrent collections, actions in a thread prior to placing an object into a `BlockingDeque` *happen-before* actions subsequent to the access or removal of that element from the `BlockingDeque` in another thread.

This interface is a member of the Java Collections Framework.

Since:

1.6

Method Summary

All Methods Instance Methods Abstract Methods

Modifier and Type	**Method**	**Description**
`boolean`	**`add`**`(E e)`	Inserts the specified element into the queue represented by this deque (in other words, at the tail of this deque) 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.
`void`	**`addFirst`**`(E e)`	Inserts the specified element at the front of this deque if it is possible to do so immediately without violating capacity restrictions, throwing an `IllegalStateException` if no space is currently available.
`void`	**`addLast`**`(E e)`	Inserts the specified element at the end of this deque if it is

		possible to do so immediately without violating capacity restrictions, throwing an `IllegalStateException` if no space is currently available.
boolean	`contains(Object o)`	Returns `true` if this deque contains the specified element.
E	`element()`	Retrieves, but does not remove, the head of the queue represented by this deque (in other words, the first element of this deque).
Iterator<E>	`iterator()`	Returns an iterator over the elements in this deque in proper sequence.
boolean	`offer(E e)`	Inserts the specified element into the queue represented by this deque (in other words, at the tail of this deque) if it is possible to do so immediately without violating capacity restrictions, returning `true` upon success and `false` if no space is currently available.
boolean	`offer(E e, long timeout, TimeUnit unit)`	Inserts the specified element into the queue represented by this deque (in other words, at the tail of this deque), waiting up to the specified wait time if necessary for space to become available.
boolean	`offerFirst(E e)`	Inserts the specified element at the front of this deque if it is possible to do so immediately without violating capacity restrictions, returning `true` upon success and `false` if no space is currently available.
boolean	`offerFirst(E e, long timeout, TimeUnit unit)`	Inserts the specified element at the front of this deque, waiting up to the specified wait time if necessary for space to become available.
boolean	`offerLast(E e)`	Inserts the specified element at the end of this deque if it is possible to do so immediately

		without violating capacity restrictions, returning `true` upon success and `false` if no space is currently available.
boolean	**offerLast**(E e, long timeout, TimeUnit unit)	Inserts the specified element at the end of this deque, waiting up to the specified wait time if necessary for space to become available.
E	**peek**()	Retrieves, but does not remove, the head of the queue represented by this deque (in other words, the first element of this deque), or returns `null` if this deque is empty.
E	**poll**()	Retrieves and removes the head of the queue represented by this deque (in other words, the first element of this deque), or returns `null` if this deque is empty.
E	**poll**(long timeout, TimeUnit unit)	Retrieves and removes the head of the queue represented by this deque (in other words, the first element of this deque), waiting up to the specified wait time if necessary for an element to become available.
E	**pollFirst**(long timeout, TimeUnit unit)	Retrieves and removes the first element of this deque, waiting up to the specified wait time if necessary for an element to become available.
E	**pollLast**(long timeout, TimeUnit unit)	Retrieves and removes the last element of this deque, waiting up to the specified wait time if necessary for an element to become available.
void	**push**(E e)	Pushes an element onto the stack represented by this deque (in other words, at the head of this deque) if it is possible to do so immediately without violating capacity restrictions, throwing an `IllegalStateException` if no space is currently available.

void	put(E e)	Inserts the specified element into the queue represented by this deque (in other words, at the tail of this deque), waiting if necessary for space to become available.
void	putFirst(E e)	Inserts the specified element at the front of this deque, waiting if necessary for space to become available.
void	putLast(E e)	Inserts the specified element at the end of this deque, waiting if necessary for space to become available.
E	remove()	Retrieves and removes the head of the queue represented by this deque (in other words, the first element of this deque).
boolean	remove(Object o)	Removes the first occurrence of the specified element from this deque.
boolean	removeFirstOccurrence (Object o)	Removes the first occurrence of the specified element from this deque.
boolean	removeLastOccurrence (Object o)	Removes the last occurrence of the specified element from this deque.
int	size()	Returns the number of elements in this deque.
E	take()	Retrieves and removes the head of the queue represented by this deque (in other words, the first element of this deque), waiting if necessary until an element becomes available.
E	takeFirst()	Retrieves and removes the first element of this deque, waiting if necessary until an element becomes available.
E	takeLast()	Retrieves and removes the last element of this deque, waiting if necessary until an element becomes available.

Methods declared in interface java.util.concurrent.BlockingQueue

`drainTo, drainTo, remainingCapacity`

Methods declared in interface java.util.Collection

`clear, containsAll, equals, hashCode, isEmpty, parallelStream, removeAll, removeIf, retainAll, spliterator, stream, toArray, toArray, toArray`

Methods declared in interface java.util.Deque

`addAll, descendingIterator, getFirst, getLast, peekFirst, peekLast, pollFirst, pollLast, pop, removeFirst, removeLast`

Methods declared in interface java.lang.Iterable

`forEach`

Method Details

addFirst

```
void addFirst(E e)
```

Inserts the specified element at the front of this deque if it is possible to do so immediately without violating capacity restrictions, throwing an `IllegalStateException` if no space is currently available. When using a capacity-restricted deque, it is generally preferable to use `offerFirst`.

Specified by:

`addFirst` in interface `Deque<E>`

Parameters:

e - the element to add

Throws:

`IllegalStateException` - if the element cannot be added at this time due to capacity restrictions

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

`NullPointerException` - if the specified element is null

`IllegalArgumentException` - if some property of the specified element prevents it from being added to this deque

addLast

```
void addLast(E e)
```

Inserts the specified element at the end of this deque if it is possible to do so immediately without violating capacity restrictions, throwing an `IllegalStateException` if no space is currently available. When using a capacity-restricted deque, it is generally preferable to use `offerLast`.

Specified by:

`addLast` in interface `Deque<E>`

Parameters:

`e` - the element to add

Throws:

`IllegalStateException` - if the element cannot be added at this time due to capacity restrictions

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

`NullPointerException` - if the specified element is null

`IllegalArgumentException` - if some property of the specified element prevents it from being added to this deque

offerFirst

```
boolean offerFirst(E e)
```

Inserts the specified element at the front of this deque if it is possible to do so immediately without violating capacity restrictions, returning `true` upon success and `false` if no space is currently available. When using a capacity-restricted deque, this method is generally preferable to the `addFirst` method, which can fail to insert an element only by throwing an exception.

Specified by:

`offerFirst` in interface `Deque<E>`

Parameters:

`e` - the element to add

Returns:

`true` if the element was added to this deque, else `false`

Throws:

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

`NullPointerException` - if the specified element is null

`IllegalArgumentException` - if some property of the specified element prevents it from being added to this deque

offerLast

```
boolean offerLast(E e)
```

Inserts the specified element at the end of this deque if it is possible to do so immediately without violating capacity restrictions, returning `true` upon success and `false` if no space is currently available. When using a capacity-restricted deque, this method is generally preferable to the `addLast` method, which can fail to insert an element only by throwing an exception.

Specified by:

`offerLast` in interface `Deque<E>`

Parameters:

`e` - the element to add

Returns:

`true` if the element was added to this deque, else `false`

Throws:

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

`NullPointerException` - if the specified element is null

`IllegalArgumentException` - if some property of the specified element prevents it from being added to this deque

putFirst

```
void putFirst(E e)
       throws InterruptedException
```

Inserts the specified element at the front of this deque, waiting if necessary for space to become available.

Parameters:

`e` - the element to add

Throws:

`InterruptedException` - if interrupted while waiting

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

`NullPointerException` - if the specified element is null

`IllegalArgumentException` - if some property of the specified element prevents it from being added to this deque

putLast

```
void putLast(E e)
      throws InterruptedException
```

Inserts the specified element at the end of this deque, waiting if necessary for space to become available.

Parameters:

`e` - the element to add

Throws:

`InterruptedException` - if interrupted while waiting

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

`NullPointerException` - if the specified element is null

`IllegalArgumentException` - if some property of the specified element prevents it from being added to this deque

offerFirst

```
boolean offerFirst(E e,
                   long timeout,
                   TimeUnit unit)
            throws InterruptedException
```

Inserts the specified element at the front of this deque, waiting up to the specified wait time if necessary for space to become available.

Parameters:

`e` - the element to add

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

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

Returns:

`true` if successful, or `false` if the specified waiting time elapses before space is available

Throws:

`InterruptedException` - if interrupted while waiting

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

`NullPointerException` - if the specified element is null

`IllegalArgumentException` - if some property of the specified element prevents it from being added to this deque

offerLast

```
boolean offerLast(E e,
                  long timeout,
                  TimeUnit unit)
           throws InterruptedException
```

Inserts the specified element at the end of this deque, waiting up to the specified wait time if necessary for space to become available.

Parameters:

`e` - the element to add

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

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

Returns:

`true` if successful, or `false` if the specified waiting time elapses before space is available

Throws:

`InterruptedException` - if interrupted while waiting

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

`NullPointerException` - if the specified element is null

`IllegalArgumentException` - if some property of the specified element prevents it from being added to this deque

takeFirst

```
E takeFirst()
     throws InterruptedException
```

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

Returns:

the head of this deque

Throws:

`InterruptedException` - if interrupted while waiting

takeLast

```
E takeLast()
    throws InterruptedException
```

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

Returns:

the tail of this deque

Throws:

`InterruptedException` - if interrupted while waiting

pollFirst

```
E pollFirst(long timeout,
            TimeUnit unit)
     throws InterruptedException
```

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

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 deque, or `null` if the specified waiting time elapses before an element is available

Throws:

`InterruptedException` - if interrupted while waiting

pollLast

```
E pollLast(long timeout,
           TimeUnit unit)
    throws InterruptedException
```

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

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

Throws:

`InterruptedException` - if interrupted while waiting

removeFirstOccurrence

```
boolean removeFirstOccurrence(Object o)
```

Removes the first occurrence of the specified element from this deque. If the deque does not contain the element, it is unchanged. More formally, removes the first element `e` such that `o.equals(e)` (if such an element exists). Returns `true` if this deque contained the specified element (or equivalently, if this deque changed as a result of the call).

Specified by:

`removeFirstOccurrence` in interface `Deque<E>`

Parameters:

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

Returns:

`true` if an element was removed as a result of this call

Throws:

`ClassCastException` - if the class of the specified element is incompatible with this deque (optional)

`NullPointerException` - if the specified element is null (optional)

removeLastOccurrence

```
boolean removeLastOccurrence(Object o)
```

Removes the last occurrence of the specified element from this deque. If the deque does not contain the element, it is unchanged. More formally, removes the last element `e` such that `o.equals(e)` (if such an element exists). Returns `true` if this deque contained the specified element (or equivalently, if this deque changed as a result of the call).

Specified by:

`removeLastOccurrence` in interface `Deque<E>`

Parameters:

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

Returns:

`true` if an element was removed as a result of this call

Throws:

`ClassCastException` - if the class of the specified element is incompatible with this deque (optional)

`NullPointerException` - if the specified element is null (optional)

add

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

Inserts the specified element into the queue represented by this deque (in other words, at the tail of this deque) 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. When using a capacity-restricted deque, it is generally preferable to use `offer`.

This method is equivalent to `addLast`.

Specified by:

`add` in interface `BlockingQueue<E>`

Specified by:

`add` in interface `Collection<E>`

Specified by:

`add` in interface `Deque<E>`

Specified by:

`add` in interface `Queue<E>`

Parameters:

`e` - the element to add

Returns:

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

Throws:

`IllegalStateException` - if the element cannot be added at this time due to capacity restrictions

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

`NullPointerException` - if the specified element is null

`IllegalArgumentException` - if some property of the specified element prevents it from being added to this deque

offer

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

Inserts the specified element into the queue represented by this deque (in other words, at the tail of this deque) if it is possible to do so immediately without violating capacity restrictions, returning `true` upon success and `false` if no space is currently available. When using a capacity-restricted deque, this method is generally preferable to the `add(E)` method, which can fail to insert an element only by throwing an exception.

This method is equivalent to `offerLast`.

Specified by:

`offer` in interface `BlockingQueue<E>`

Specified by:

`offer` in interface `Deque<E>`

Specified by:

`offer` in interface `Queue<E>`

Parameters:

`e` - the element to add

Returns:

`true` if the element was added to this queue, else `false`

Throws:

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

`NullPointerException` - if the specified element is null

`IllegalArgumentException` - if some property of the specified element prevents it from being added to this deque

put

```
void put(E e)
  throws InterruptedException
```

Inserts the specified element into the queue represented by this deque (in other words, at the tail of this deque), waiting if necessary for space to become available.

This method is equivalent to `putLast`.

Specified by:

`put` in interface `BlockingQueue<E>`

Parameters:

`e` - the element to add

Throws:

`InterruptedException` - if interrupted while waiting

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

`NullPointerException` - if the specified element is null

`IllegalArgumentException` - if some property of the specified element prevents it from being added to this deque

offer

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

Inserts the specified element into the queue represented by this deque (in other words, at the tail of this deque), waiting up to the specified wait time if necessary for space to become available.

This method is equivalent to `offerLast`.

Specified by:

`offer` in interface `BlockingQueue<E>`

Parameters:

`e` - the element to add

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

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

Returns:

`true` if the element was added to this deque, else `false`

Throws:

`InterruptedException` - if interrupted while waiting

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

`NullPointerException` - if the specified element is null

`IllegalArgumentException` - if some property of the specified element prevents it from being added to this deque

remove

```
E remove()
```

Retrieves and removes the head of the queue represented by this deque (in other words, the first element of this deque). This method differs from `poll()` only in that it throws an exception if this deque is empty.

This method is equivalent to `removeFirst`.

Specified by:

`remove` in interface `Deque<E>`

Specified by:

`remove` in interface `Queue<E>`

Returns:

the head of the queue represented by this deque

Throws:

`NoSuchElementException` - if this deque is empty

poll

```
E poll()
```

Retrieves and removes the head of the queue represented by this deque (in other words, the first element of this deque), or returns `null` if this deque is empty.

This method is equivalent to `Deque.pollFirst()`.

Specified by:

`poll` in interface `Deque<E>`

Specified by:

`poll` in interface `Queue<E>`

Returns:

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

take

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

Retrieves and removes the head of the queue represented by this deque (in other words, the first element of this deque), waiting if necessary until an element becomes available.

This method is equivalent to `takeFirst`.

Specified by:

`take` in interface `BlockingQueue<E>`

Returns:

the head of this deque

Throws:

`InterruptedException` - if interrupted while waiting

poll

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

Retrieves and removes the head of the queue represented by this deque (in other words, the first element of this deque), waiting up to the specified wait time if necessary for an element to become available.

This method is equivalent to `pollFirst`.

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 deque, or `null` if the specified waiting time elapses before an element is available

Throws:

`InterruptedException` - if interrupted while waiting

element

```
E element()
```

Retrieves, but does not remove, the head of the queue represented by this deque (in other words, the first element of this deque). This method differs from `peek` only in that it throws an exception if this deque is empty.

This method is equivalent to `getFirst`.

Specified by:

`element` in interface `Deque<E>`

Specified by:

`element` in interface `Queue<E>`

Returns:

the head of this deque

Throws:

`NoSuchElementException` - if this deque is empty

peek

```
E peek()
```

Retrieves, but does not remove, the head of the queue represented by this deque (in other words, the first element of this deque), or returns `null` if this deque is empty.

This method is equivalent to `peekFirst`.

Specified by:

`peek` in interface `Deque<E>`

Specified by:

`peek` in interface `Queue<E>`

Returns:

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

remove

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

Removes the first occurrence of the specified element from this deque. If the deque does not contain the element, it is unchanged. More formally, removes the first element `e` such that `o.equals(e)` (if such an element exists). Returns `true` if this deque contained the specified element (or equivalently, if this deque changed as a result of the call).

This method is equivalent to `removeFirstOccurrence`.

Specified by:

`remove` in interface `BlockingQueue<E>`

Specified by:

`remove` in interface `Collection<E>`

Specified by:

`remove` in interface `Deque<E>`

Parameters:

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

Returns:

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

Throws:

`ClassCastException` - if the class of the specified element is incompatible with this deque (optional)

`NullPointerException` - if the specified element is null (optional)

contains

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

Returns `true` if this deque contains the specified element. More formally, returns `true` if and only if this deque 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>`

Specified by:

`contains` in interface `Deque<E>`

Parameters:

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

Returns:

`true` if this deque contains the specified element

Throws:

`ClassCastException` - if the class of the specified element is incompatible with this deque (optional)

`NullPointerException` - if the specified element is null (optional)

size

```
int size()
```

Returns the number of elements in this deque.

Specified by:

`size` in interface `Collection<E>`

Specified by:

`size` in interface `Deque<E>`

Returns:

the number of elements in this deque

iterator

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

Returns an iterator over the elements in this deque in proper sequence. The elements will be returned in order from first (head) to last (tail).

Specified by:

`iterator` in interface `Collection<E>`

Specified by:

`iterator` in interface `Deque<E>`

Specified by:

`iterator` in interface `Iterable<E>`

Returns:

an iterator over the elements in this deque in proper sequence

push

```
void push(E e)
```

Pushes an element onto the stack represented by this deque (in other words, at the head of this deque) if it is possible to do so immediately without violating capacity restrictions, throwing an `IllegalStateException` if no space is currently available.

This method is equivalent to `addFirst`.

Specified by:

`push` in interface `Deque<E>`

Parameters:

e - the element to push

Throws:

`IllegalStateException` - if the element cannot be added at this time due to capacity restrictions

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

`NullPointerException` - if the specified element is null

`IllegalArgumentException` - if some property of the specified element prevents it from being added to this deque

Report a bug or suggest an enhancement
For further API reference and developer documentation see the Java SE Documentation, which contains more detailed, developer-targeted descriptions with conceptual overviews, definitions of terms, workarounds, and working code examples. Other versions.
Java is a trademark or registered trademark of Oracle and/or its affiliates in the US and other countries.
Copyright © 1993, 2022, Oracle and/or its affiliates, 500 Oracle Parkway, Redwood Shores, CA 94065 USA.
All rights reserved. Use is subject to license terms and the documentation redistribution policy. Modify Preferências de Cookies. Modify Ad Choices.