Module java.base
Package java.util.concurrent

Interface BlockingQueue<E>

Type Parameters:

E - the type of elements held in this queue

All Superinterfaces:

Collection<E>, Iterable<E>, Queue<E>

All Known Subinterfaces:

BlockingDeque<E>, TransferQueue<E>

All Known Implementing Classes:

ArrayBlockingQueue, DelayQueue, LinkedBlockingDeque, LinkedBlockingQueue, LinkedTransferQueue, PriorityBlockingQueue, SynchronousQueue

---

```
public interface BlockingQueue<E>
extends Queue<E>
```

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

BlockingQueue 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 BlockingQueue methods

	Throws exception	*Special value*	*Blocks*	*Times out*
Insert	add(e)	offer(e)	put(e)	offer(e, time, unit)
Remove	remove()	poll()	take()	poll(time, unit)
Examine	element()	peek()	*not applicable*	*not applicable*

A BlockingQueue does not accept null elements. Implementations throw NullPointerException on attempts to add, put or offer a null. A null is used as a sentinel value to indicate failure of poll operations.

A BlockingQueue may be capacity bounded. At any given time it may have a remainingCapacity beyond which no additional elements can be put without blocking. A BlockingQueue without any intrinsic capacity constraints always reports a remaining capacity of Integer.MAX_VALUE.

BlockingQueue implementations are designed to be used primarily for producer-consumer queues, but additionally support the Collection interface. So, for example, it is possible to remove an arbitrary element from a queue using remove(x). However, such operations are in general *not* performed very efficiently, and are intended for only occasional use, such as when a queued message is cancelled.

`BlockingQueue` implementations are thread-safe. All queuing methods achieve their effects atomically using internal locks or other forms of concurrency control. However, the *bulk* Collection operations `addAll`, `containsAll`, `retainAll` and `removeAll` are *not* necessarily performed atomically unless specified otherwise in an implementation. So it is possible, for example, for `addAll(c)` to fail (throwing an exception) after adding only some of the elements in `c`.

A `BlockingQueue` does *not* intrinsically support any kind of "close" or "shutdown" operation to indicate that no more items will be added. The needs and usage of such features tend to be implementation-dependent. For example, a common tactic is for producers to insert special *end-of-stream* or *poison* objects, that are interpreted accordingly when taken by consumers.

Usage example, based on a typical producer-consumer scenario. Note that a `BlockingQueue` can safely be used with multiple producers and multiple consumers.

```
class Producer implements Runnable {
  private final BlockingQueue queue;
  Producer(BlockingQueue q) { queue = q; }
  public void run() {
    try {
      while (true) { queue.put(produce()); }
    } catch (InterruptedException ex) { ... handle ...}
  }
  Object produce() { ... }
}

class Consumer implements Runnable {
  private final BlockingQueue queue;
  Consumer(BlockingQueue q) { queue = q; }
  public void run() {
    try {
      while (true) { consume(queue.take()); }
    } catch (InterruptedException ex) { ... handle ...}
  }
  void consume(Object x) { ... }
}

class Setup {
  void main() {
    BlockingQueue q = new SomeQueueImplementation();
    Producer p = new Producer(q);
    Consumer c1 = new Consumer(q);
    Consumer c2 = new Consumer(q);
    new Thread(p).start();
    new Thread(c1).start();
    new Thread(c2).start();
  }
}
```

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

This interface is a member of the Java Collections Framework.

Since:

1.5

Method Summary

All Methods Instance Methods Abstract Methods

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.
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.
boolean	**offer**(**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 `false` if no space is currently available.
boolean	**offer**(**E** e, long timeout, **TimeUnit** unit)	Inserts the specified element into this queue, waiting up to the specified wait time if necessary for space to become available.
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 queue, waiting if

		necessary for space to become available.
int	**remainingCapacity**()	Returns the number of additional elements that this queue can ideally (in the absence of memory or resource constraints) accept without blocking, or `Integer.MAX_VALUE` if there is no intrinsic limit.
boolean	**remove**(**Object** o)	Removes a single instance of the specified element from this queue, if it is present.
E	**take**()	Retrieves and removes the head of this queue, waiting if necessary until an element becomes available.

Methods declared in interface java.util.Collection

addAll, clear, containsAll, equals, hashCode, isEmpty, iterator, parallelStream, removeAll, removeIf, retainAll, size, spliterator, stream, toArray, toArray, toArray

Methods declared in interface java.lang.Iterable

forEach

Methods declared in interface java.util.Queue

element, peek, poll, remove

Method Details

add

```
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. When using a capacity-restricted queue, it is generally preferable to use `offer`.

Specified by:

add in interface Collection<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 queue

`NullPointerException` - if the specified element is null

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

offer

```
boolean offer(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 `false` if no space is currently available. When using a capacity-restricted queue, this method is generally preferable to `add(E)`, which can fail to insert an element only by throwing an exception.

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 queue

`NullPointerException` - if the specified element is null

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

put

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

Inserts the specified element into this queue, 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 queue

`NullPointerException` - if the specified element is null

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

offer

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

Inserts the specified element into this queue, 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 queue

`NullPointerException` - if the specified element is null

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

take

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

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

Returns:

the head of this queue

Throws:

`InterruptedException` - if interrupted while waiting

poll

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

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

Throws:

`InterruptedException` - if interrupted while waiting

remainingCapacity

```
int remainingCapacity()
```

Returns the number of additional elements that this queue can ideally (in the absence of memory or resource constraints) accept without blocking, or `Integer.MAX_VALUE` if there is no intrinsic limit.

Note that you *cannot* always tell if an attempt to insert an element will succeed by inspecting `remainingCapacity` because it may be the case that another thread is about to insert or remove an element.

Returns:

the remaining capacity

remove

```
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 this queue contained the specified element (or equivalently, if this queue changed as a result of the call).

Specified by:

`remove` in interface `Collection<E>`

Parameters:

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

Returns:

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

Throws:

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

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

contains

```
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 `Collection<E>`

Parameters:

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

Returns:

`true` if this queue contains the specified element

Throws:

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

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

drainTo

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

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.

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

```
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. 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.

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

---