

**java.util.Queue Interface**

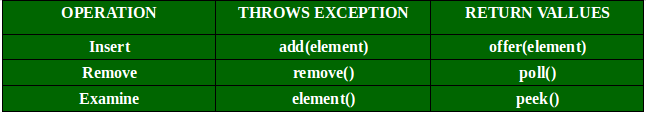
* The Queue interface is available in java.util package and extends the Collection interface.
* A collection designed for holding elements prior to processing.
* Besides basic Collection operations, queues provide additional insertion, extraction, and inspection operations.
* Queues typically, but do not necessarily, order elements in a FIFO (first-in-first-out) manner.

**Few important characteristics of Queue are: -**

* 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](https://www.geeksforgeeks.org/linked-list-in-java/), ArrayBlockingQueue and [PriorityQueue](https://www.geeksforgeeks.org/priority-queue-class-in-java-2/) are the most frequently used implementations.
* If any null operation is performed on BlockingQueues, NullPointerException is thrown.
* BlockingQueues have thread-safe implementations.
* 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.
* BlockingQueues do not accept null elements. If we perform any null related operation, it throws NullPointerException.
* BlockingQueues are used to implement Producer/Consumer based applications.
* BlockingQueues are thread-safe.
* All Queues which are available in java.util package are Unbounded Queues and Queues which are available in java.util.concurrent package are Bounded Queues.
* All Deques are not thread-safe.
* ConcurrentLinkedQueue is an unbounded thread-safe Queue based on linked nodes.
* All Queues supports insertion at the tail of the queue and removal at the head of the queue, except Deques.
* Deques are queues but they support element insertion and removal at both ends.

**Methods in Queue:**

1. **add ()-** This method is used to add elements at the tail of queue. More specifically, at the last of LinkedList if it is used, or according to the priority in case of priority queue implementation.
2. **peek ()-** This method is used to view the head of queue without removing it. It returns **Null** if the queue is empty.
3. **element ()-** This method is similar to peek (). It throws ***NoSuchElementException*** when the queue is empty.
4. **remove ()-** This method removes and returns the head of the queue. It throws ***NoSuchElementException*** when the queue is empty.
5. **poll ()-** This method removes and returns the head of the queue. It returns null if the queue is empty.
6. **size ()-** This method returns the no. of elements in the queue.



Since it is a subtype of Collections class, it inherits all the methods of it namely *size (), isEmpty (), contains () etc.*

**Some More Points: -**

* **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.

**Because the Queue interface extends the Collection interface, all Queue implementations have basic operations of a collection: -**

* **Single operations:** add(e), contains(e), iterator (), clear (), isEmpty(), size() and toArray().
* **Bulk operations:** addAll(), containsAll(), removeAll() and retainAll().

**Queue Categories: -**

In Java, we can find many Queue implementations. We can broadly categorize them into the following two types: -

1. **Bounded Queues**
2. **Unbounded Queues**

**Bounded Queues: -**Bounded Queues are queues which are bounded by capacity that means we need to provide the max size of the queue at the time of creation. For example, **ArrayBlockingQueue.** Queues which are available in java.util.concurrent package are Bounded Queues.

**Unbounded Queues: -**Unbounded Queues are queues which are NOT bounded by capacity that means we should not provide the size of the queue. For example, LinkedList (see previous example). All Queues which are available in java.util package are Unbounded Queues.

**In other ways, W can broadly categorize them into the following two types: -**

1. **Blocking Queues: -** All Queues which implement BlockingQueue interface are BlockingQueues. BlockingQueues blocks until it finishes its job or time out.
2. **Non-Blocking Queues: -** All Queues which does not implement BlockingQueue interface are Non-Blocking Queues. Non-BlockingQueues do not block any operations.

**Sub-interfaces of Queues: -**

Queue is the super interface of the queue branch in the Java Collection Framework. Under it, there are the following sub interfaces:

1. **java.util.Deque interface: -** abstracts a queue that has two heads. A deque allows adding or removing elements at both ends.
2. **java.util.concurrent.BlockingQueue interface: -**abstracts a type of queues that waits for the queue to be non-empty when retrieving an element and waits for space to become available in the queue when storing an element.
3. **java.util.concurrent.BlockingDeque interface: -**is similar to BlockingQueue, but for double ended queues. It is sub interface of the BlockingQueue.

**Note: -**And since Java 7, the BlockingQueue interface has a new sub interface called **TransferQueue**, which is a specialized BlockingQueue, which waits for another thread to retrieve an element in the queue.

**Queue Implementations: -**

The Queue implementations are grouped into general-purpose and concurrent implementations.

**General-Purpose Queue Implementations: -**

1. **LinkedList: -** This class implements both List and Deque interface, thus having hybrid characteristics and behaviors of list and queue. Consider using a LinkedList when you want fast adding and fast removing elements at both ends, plus accessing elements by index.
2. **PriorityQueue:-** This queue orders elements according to their natural ordering, or by a Comparator provided at construction time. Consider using a PriorityQueue when you want to take advantages of natural ordering and fast adding elements to the tail and fast removing elements at the head of the queue.
3. **ArrayDeque:-** A simple implementation of the Deque interface. Consider using an ArrayDeque when you want to utilize features of a double ended queue without list-based ones (simpler than a LinkedList).

**Concurrent Queue Implementations.**

1. **ArrayBlockingQueue: -T**his is a blocking queue backed by an array. Consider using an ArrayBlockingQueue when you want to use a simple blocking queue that has limited capacity (bounded).
2. **PriorityBlockingQueue: -**Use this class when you want to take advantages of both PriorityQueue and BlockingQueue.
3. **DelayQueue: -** Time-based scheduling blocking queue. Elements added to this queue must implement the Delayed interface. That means an element can only be taken from the head of the queue when its delay has expired.
4. **LinkedBlockingQueue: -**an optionally bounded FIFO blocking queue backed by linked nodes
5. **SynchronousQueue: -**a simple rendezvous mechanism that uses the BlockingQueue interface
6. **LinkedTransferQueue: -**an unbounded TransferQueue based on linked nodes.

**java.util.concurrent. BlockingQueue Interface.**

**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:

1. one throws an exception,
2. the second returns a special value (either null or false, depending on the operation),
3. the third blocks the current thread indefinitely until the operation can succeed,
4. 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)](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html#add-E-) | [offer(e)](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html#offer-E-) | [put(e)](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html#put-E-) | [offer(e, time, unit)](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html#offer-E-long-java.util.concurrent.TimeUnit-) |
| **Remove** | [remove()](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html#remove-java.lang.Object-) | [poll()](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html#poll-long-java.util.concurrent.TimeUnit-) | [take()](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html#take--) | [poll(time, unit)](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html#poll-long-java.util.concurrent.TimeUnit-) |
| **Examine** | [element()](https://docs.oracle.com/javase/8/docs/api/java/util/Queue.html#element--) | [peek()](https://docs.oracle.com/javase/8/docs/api/java/util/Queue.html#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 remaining Capacity 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](https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html) 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.

## ***BlockingQueue* Types**

We can distinguish two types of *BlockingQueue*:

* unbounded queue – can grow almost indefinitely
* bounded queue – with maximal capacity defined

### **Unbounded Queue**

Creating unbounded queues is simple:

|  |  |
| --- | --- |
| 1 | BlockingQueue<String> blockingQueue = **new** LinkedBlockingDeque<> (); |

The Capacity of *blockingQueue* will be set to *Integer.MAX\_VALUE.* All operations that add an element to the unbounded queue will never block, thus it could grow to a very large size.

### **Bounded Queue**

The second type of queues is the bounded queue. We can create such queues by passing the capacity as an argument to a constructor:

|  |
| --- |
| BlockingQueue<String> blockingQueue = **new** LinkedBlockingDeque<> (10); |

Here we have a *blockingQueue* that has a capacity equal to 10. It means that when a consumer tries to add an element to an already full queue, depending on a method that was used to add it (*offer ()*, *add()* or *put()*), it will block until space for inserting object becomes available. Otherwise, the operations will fail.

Using bounded queue is a good way to design concurrent programs because when we insert an element to an already full queue, that operations need to wait until consumers catch up and make some space available in the queue. It gives us throttling without any effort on our part.

## ***BlockingQueue* API**

There are two types of methods in the *BlockingQueue* interface *–* methods responsible for adding elements to a queue and methods that retrieve those elements. Each method from those two groups behaves differently in case the queue is full/empty.

### **Adding Elements**

* ***add () –***returns *true* if insertion was successful, otherwise throws an ***IllegalStateException***
* ***put () –*** inserts the specified element into a queue, waiting for a free slot if necessary
* ***offer () –***returns *true* if insertion was successful, otherwise *false*
* ***offer (E e, long timeout, TimeUnit unit) –*** tries to insert element into a queue and waits for an available slot within a specified timeout

### **Retrieving Elements**

* ***take ()* –** waits for a head element of a queue and removes it. If the queue is empty, it blocks and waits for an element to become available
* ***poll (long timeout, TimeUnit unit) –*** retrieves and removes the head of the queue, waiting up to the specified wait time if necessary, for an element to become available. Returns *null* after a timeout

**java.util.Deque interface**

**public interface Deque<E> extends Queue<E>**

* A linear collection that supports element insertion and removal at both ends.
* The name *deque* is short for "double ended queue" and is usually pronounced "deck".
* Most Deque implementations place no fixed limits on the number of elements they may contain, but this interface supports capacity-restricted deques as well as those with no fixed size limit.
* This interface defines methods to access the elements at both ends of the deque.
* Methods are provided to insert, remove, and examine the element.
* 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 Deque implementations; in most implementations, insert operations cannot fail.

**Summary of Deque methods**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **First Element (Head)** |  | **Last Element (Tail)** |  |
|  | *Throws exception* | *Special value* | *Throws exception* | *Special value* |
| **Insert** | [addFirst(e)](https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html#addFirst-E-) | [offerFirst(e)](https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html#offerFirst-E-) | [addLast(e)](https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html#addLast-E-) | [offerLast(e)](https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html#offerLast-E-) |
| **Remove** | [removeFirst()](https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html#removeFirst--) | [pollFirst()](https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html#pollFirst--) | [removeLast()](https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html#removeLast--) | [pollLast()](https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html#pollLast--) |
| **Examine** | [getFirst()](https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html#getFirst--) | [peekFirst()](https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html#peekFirst--) | [getLast()](https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html#getLast--) | [peekLast()](https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html#peekLast--) |

* This interface extends the [Queue](https://docs.oracle.com/javase/8/docs/api/java/util/Queue.html) interface.
* When a deque is used as a queue, FIFO (First-In-First-Out) behavior results. Elements are added at the end of the deque and removed from the beginning. The methods inherited from the Queue interface are precisely equivalent to Deque methods as indicated in the following table:

**Comparison of Queue and Deque methods**

|  |  |
| --- | --- |
| **Queue Method** | **Equivalent Deque Method** |
| [add(e)](https://docs.oracle.com/javase/8/docs/api/java/util/Queue.html#add-E-) | [addLast(e)](https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html#addLast-E-) |
| [offer(e)](https://docs.oracle.com/javase/8/docs/api/java/util/Queue.html#offer-E-) | [offerLast(e)](https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html#offerLast-E-) |
| [remove()](https://docs.oracle.com/javase/8/docs/api/java/util/Queue.html#remove--) | [removeFirst()](https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html#removeFirst--) |
| [poll()](https://docs.oracle.com/javase/8/docs/api/java/util/Queue.html#poll--) | [pollFirst()](https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html#pollFirst--) |
| [element()](https://docs.oracle.com/javase/8/docs/api/java/util/Queue.html#element--) | [getFirst()](https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html#getFirst--) |
| [peek()](https://docs.oracle.com/javase/8/docs/api/java/util/Queue.html#peek--) | [peekFirst()](https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html#peek--) |

* Deques can also be used as LIFO (Last-In-First-Out) stacks.
* This interface should be used in preference to the legacy [Stack](https://docs.oracle.com/javase/8/docs/api/java/util/Stack.html) class. When a deque is used as a stack, elements are pushed and popped from the beginning of the deque.
* Stack methods are precisely equivalent to Deque methods as indicated in the table below:

**Comparison of Stack and Deque methods**

|  |  |
| --- | --- |
| **Stack Method** | **Equivalent Deque Method** |
| [push(e)](https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html#push-E-) | [addFirst(e)](https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html#addFirst-E-) |
| [pop()](https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html#pop--) | [removeFirst()](https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html#removeFirst--) |
| [peek()](https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html#peek--) | [peekFirst()](https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html#peekFirst--) |

**Note** that the [peek](https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html#peek--) method works equally well when a deque is used as a queue or a stack; in either case, elements are drawn from the beginning of the deque.

**This interface provides two methods to remove interior elements,**

1. [removeFirstOccurrence](https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html#removeFirstOccurrence-java.lang.Object-) and
2. [removeLastOccurrence](https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html#removeLastOccurrence-java.lang.Object-).

* Unlike the [List](https://docs.oracle.com/javase/8/docs/api/java/util/List.html) interface, this interface does not provide support for indexed access to elements.
* While Deque implementations are not strictly required to prohibit the insertion of null elements, they are strongly encouraged to do so. Users of any Deque implementations that do allow null elements are strongly encouraged *not* to take advantage of the ability to insert nulls. This is so because null is used as a special return value by various methods to indicate that the deque is empty.
* Deque implementations generally do not define element-based versions of the equals and hashCode methods, but instead inherit the identity-based versions from class Object.

**Few important features of Deque are:**

* It provides the support of resizable array and helps in restriction-free capacity, so to grow the array according to the usage.
* Array deques prohibit the use of Null elements and do not accept any such elements.
* Any concurrent access by multiple threads is not supported.
* In the absence of external synchronization, Deque is not thread-safe.

**java.util.concurrent.BlockingDeque**

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

A [Deque](https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html) 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:

1. one throws an exception,
2. the second returns a special value (either null or false, depending on the operation),
3. the third blocks the current thread indefinitely until the operation can succeed,
4. 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)](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingDeque.html#addFirst-E-) | [offerFirst(e)](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingDeque.html#offerFirst-E-) | [putFirst(e)](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingDeque.html#putFirst-E-) | [offerFirst(e, time, unit)](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingDeque.html#offerFirst-E-long-java.util.concurrent.TimeUnit-) |
| **Remove** | [removeFirst()](https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html#removeFirst--) | [pollFirst()](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingDeque.html#pollFirst-long-java.util.concurrent.TimeUnit-) | [takeFirst()](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingDeque.html#takeFirst--) | [pollFirst(time, unit)](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingDeque.html#pollFirst-long-java.util.concurrent.TimeUnit-) |
| **Examine** | [getFirst()](https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html#getFirst--) | [peekFirst()](https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html#peekFirst--) | *not applicable* | *not applicable* |
| **Last Element (Tail)** |  |  |  |  |
|  | *Throws exception* | *Special value* | *Blocks* | *Times out* |
| **Insert** | [addLast(e)](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingDeque.html#addLast-E-) | [offerLast(e)](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingDeque.html#offerLast-E-) | [putLast(e)](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingDeque.html#putLast-E-) | [offerLast(e, time, unit)](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingDeque.html#offerLast-E-long-java.util.concurrent.TimeUnit-) |
| **Remove** | [removeLast()](https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html#removeLast--) | [pollLast()](https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html#pollLast--) | [takeLast()](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingDeque.html#takeLast--) | [pollLast(time, unit)](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingDeque.html#pollLast-long-java.util.concurrent.TimeUnit-) |
| **Examine** | [getLast()](https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html#getLast--) | [peekLast()](https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html#peekLast--) | *not applicable* | *not applicable* |

* Like any [BlockingQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html), 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.

## **BlockingDeque Extends BlockingQueue**

The BlockingDeque interface extends the BlockingQueue interface. That means that you can use a BlockingDeque as a BlockingQueue. If you do so, the various inserting methods will add the elements to the end of the deque, and the removing methods will remove the elements from the beginning of the deque. The inserting and removing methods of the BlockingQueue interface, that is. The methods inherited from the BlockingQueue interface are precisely equivalent to BlockingDeque methods as indicated in the following table:

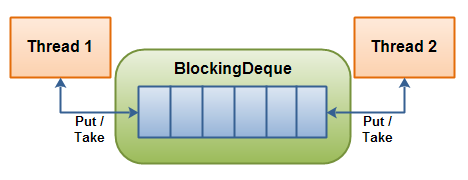
**Comparison of BlockingQueue and BlockingDeque methods**

|  |  |
| --- | --- |
| **BlockingQueue Method** | **Equivalent BlockingDeque Method** |
| **Insert** |  |
| [add(e)](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingDeque.html#add-E-) | [addLast(e)](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingDeque.html#addLast-E-) |
| [offer(e)](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingDeque.html#offer-E-) | [offerLast(e)](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingDeque.html#offerLast-E-) |
| [put(e)](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingDeque.html#put-E-) | [putLast(e)](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingDeque.html#putLast-E-) |
| [offer(e, time, unit)](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingDeque.html#offer-E-long-java.util.concurrent.TimeUnit-) | [offerLast(e, time, unit)](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingDeque.html#offerLast-E-long-java.util.concurrent.TimeUnit-) |
| **Remove** |  |
| [remove()](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingDeque.html#remove--) | [removeFirst()](https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html#removeFirst--) |
| [poll()](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingDeque.html#poll--) | [pollFirst()](https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html#pollFirst--) |
| [take()](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingDeque.html#take--) | [takeFirst()](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingDeque.html#takeFirst--) |
| [poll(time, unit)](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingDeque.html#poll-long-java.util.concurrent.TimeUnit-) | [pollFirst(time, unit)](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingDeque.html#pollFirst-long-java.util.concurrent.TimeUnit-) |
| **Examine** |  |
| [element()](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingDeque.html#element--) | [getFirst()](https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html#getFirst--) |
| [peek()](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingDeque.html#peek--) | [peekFirst()](https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html#peekFirst--) |

**Memory consistency effects:** As with other concurrent collections, actions in a thread prior to placing an object into a BlockingDeque [*happen-before*](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html#MemoryVisibility) *actions subsequent to the access or removal of that element from the* BlockingDeque in another thread.

## **BlockingDeque Usage**

A BlockingDeque could be used if threads are both producing and consuming elements of the same queue. It could also just be used if the producting thread needs to insert at both ends of the queue, and the consuming thread needs to remove from both ends of the queue. Here is an illustration of that:



**A BlockingDeque - threads can put and take from both ends of the deque.**

A thread will produce elements and insert them into either end of the queue. If the deque is currently full, the inserting thread will be blocked until a removing thread takes an element out of the deque. If the deque is currently empty, a removing thread will be blocked until an inserting thread inserts an element into the deque.

**java.util.concurrent.TransferQueue Interface**

**public interface TransferQueue<E> extends BlockingQueue<E>**

* A [BlockingQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html) in which producers may wait for consumers to receive elements.
* A TransferQueue may be useful for example in message passing applications in which producers sometimes (using method [transfer(E)](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/TransferQueue.html#transfer-E-)) await receipt of elements by consumers invoking take or poll, while at other times enqueue elements (via method put) without waiting for receipt.
* [Non-blocking](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/TransferQueue.html#tryTransfer-E-) and [time-out](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/TransferQueue.html#tryTransfer-E-long-java.util.concurrent.TimeUnit-) versions of try Transfer are also available.
* A TransferQueue may also be queried, via [hasWaitingConsumer()](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/TransferQueue.html#hasWaitingConsumer--), whether there are any threads waiting for items, which is a converse analogy to a peek operation.
* Like other blocking queues, a TransferQueue may be capacity bounded. If so, an attempted transfer operation may initially block waiting for available space, and/or subsequently block waiting for reception by a consumer. Note that in a queue with zero capacity, such as [SynchronousQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/SynchronousQueue.html), put and transfer are effectively synonymous.
* The *TransferQueue* can be very useful when we do not want an over-producing producer that will flood the queue with messages, resulting in the *OutOfMemory* errors. In such design, the consumer will be dictating the speed at which the producer will produce messages.

**java.util.AbstractQueue**

**public abstract class AbstractQueue<E> extends AbstractCollection<E> implements Queue<E>**

* This class provides skeletal implementations of some Queue operations.
* The implementations in this class are appropriate when the base implementation does not allow null elements.
* Methods add, remove, and element are based on offer, poll, and peek, respectively, but throw exceptions instead of indicating failure via false or null returns.
* A Queue implementation that extends this class must minimally define a method Queue.offer(E) which does not permit insertion of null elements, along with methods Queue.peek(), Queue.poll(), Collection.size(), and Collection.iterator().
* Typically, additional methods will be overridden as well. If these requirements cannot be met, consider instead subclassing AbstractCollection.

**java.util.PriorityQueue**

**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() is 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).

**Few important points on Priority Queue are as follows:**

* PriorityQueue doesn’t permit NULL pointers.
* We can’t create PriorityQueue of Objects that are non-comparable
* PriorityQueue are unbound queues.
* 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.
* It inherits methods from AbstractQueue, AbstractCollection, Collection and Object class.

|  |  |
| --- | --- |
| **Modifier and Type** | **Method and Description** |
| boolean | [**add**](https://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html#add-E-)**(**[**E**](https://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html) **e)**  Inserts the specified element into this priority queue. |
| void | [**clear**](https://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html#clear--)**()**  Removes all of the elements from this priority queue. |
| [**Comparator**](https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html)**<? super** [**E**](https://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html)**>** | [**comparator**](https://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html#comparator--)**()**  Returns the comparator used to order the elements in this queue, or null if this queue is sorted according to the [**natural ordering**](https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html) **of its elements.** |
| boolean | [**contains**](https://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html#contains-java.lang.Object-)**(**[**Object**](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html) **o)**  Returns true if this queue contains the specified element. |
| [**Iterator**](https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html)**<**[**E**](https://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html)**>** | [**iterator**](https://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html#iterator--)**()**  Returns an iterator over the elements in this queue. |
| boolean | [**offer**](https://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html#offer-E-)**(**[**E**](https://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html) **e)**  Inserts the specified element into this priority queue. |
| [**E**](https://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html) | [**peek**](https://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html#peek--)**()**  Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty. |
| [**E**](https://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html) | [**poll**](https://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html#poll--)**()**  Retrieves and removes the head of this queue, or returns null if this queue is empty. |
| boolean | [**remove**](https://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html#remove-java.lang.Object-)**(**[**Object**](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html) **o)**  Removes a single instance of the specified element from this queue, if it is present. |
| int | [**size**](https://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html#size--)**()**  Returns the number of elements in this collection. |
| [**Spliterator**](https://docs.oracle.com/javase/8/docs/api/java/util/Spliterator.html)**<**[**E**](https://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html)**>** | [**spliterator**](https://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html#spliterator--)**()**  Creates a [***late-binding***](https://docs.oracle.com/javase/8/docs/api/java/util/Spliterator.html#binding) ***and*** *fail-fast* [**Spliterator**](https://docs.oracle.com/javase/8/docs/api/java/util/Spliterator.html) **over the elements in this queue.** |
| [**Object**](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html)**[]** | [**toArray**](https://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html#toArray--)**()**  Returns an array containing all of the elements in this queue. |
| <T> T[] | [**toArray**](https://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html#toArray-T:A-)**(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. |

**java.util.ArrayDeque<E>**

**public class ArrayDeque<E> extends AbstractCollection<E> implements Deque<E>, Cloneable, Serializable**

* Resizable-array implementation of the Deque interface.
* Array deques have no capacity restrictions; they grow as necessary to support usage.
* They are not thread-safe; in the absence of external synchronization, they do not support concurrent access by multiple threads.
* Null elements are prohibited.
* This class is likely to be faster than Stack when used as a stack, and faster than LinkedList when used as a queue.
* Most ArrayDeque operations run in amortized constant time.
* Exceptions include remove, removeFirstOccurrence, removeLastOccurrence, contains, iterator.remove(), and the bulk operations, all of which run in linear time.
* The iterators returned by this class's iterator method are fail-fast: If the deque is modified at any time after the iterator is created, in any way except through the iterator's own remove method, the iterator will generally throw a **ConcurrentModificationException**. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future. Note that the fail-fast behavior of an iterator cannot be guaranteed as it is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw **ConcurrentModificationException** on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness: the fail-fast behavior of iterators should be used only to detect bugs.
* This class and its iterator implement all of the optional methods of the Collection and Iterator interfaces.

ArrayDeque in Java provides a way to apply resizable-array in addition to the implementation of the Deque interface. It is also known as Array Double Ended Queue or Array Deck. This is a special kind of array that grows and allows users to add or remove an element from both the sides of the queue.

**Few important features of ArrayDeque are as follows:**

* Array deques have no capacity restrictions and they grow as necessary to support usage.
* They are not thread-safe which means that in the absence of external synchronization, ArrayDeque does not support concurrent access by multiple threads.
* Null elements are prohibited in the ArrayDeque.
* ArrayDeque class is likely to be faster than Stack when used as a stack.
* ArrayDeque class is likely to be faster than LinkedList when used as a queue.

**java.util.concurrent.ArrayBlockingQueue<E>**

**public class ArrayBlockingQueue<E> extends AbstractQueue<E> implements BlockingQueue<E>, Serializable**

* A bounded blocking queue backed by an array.
* This queue orders elements FIFO (first-in-first-out). The head of the queue is that element that has been on the queue the longest time. The tail of the queue is that element that has been on the queue the shortest time. New elements are inserted at the tail of the queue, and the queue retrieval operations obtain elements at the head of the queue.
* This is a classic "bounded buffer", in which a fixed-sized array holds elements inserted by producers and extracted by consumers. Once created, the capacity cannot be changed. Attempts to put an element into a full queue will result in the operation blocking; attempts to take an element from an empty queue will similarly block.
* This class supports an optional fairness policy for ordering waiting producer and consumer threads. By default, this ordering is not guaranteed. However, a queue constructed with fairness set to true grants threads access in FIFO order. Fairness generally decreases throughput but reduces variability and avoids starvation.
* This class and its iterator implement all of the optional methods of the Collection and Iterator interfaces.

**java.util.concurrent.ConcurrentLinkedQueue<E>**

**public class ConcurrentLinkedQueue<E> extends AbstractQueue<E> implements Queue<E>, Serializable**

* An unbounded thread-safe queue based on linked nodes.
* This queue orders elements FIFO (first-in-first-out). The head of the queue is that element that has been on the queue the longest time. The tail of the queue is that element that has been on the queue the shortest time. New elements are inserted at the tail of the queue, and the queue retrieval operations obtain elements at the head of the queue.
* A **ConcurrentLinkedQueue** is an appropriate choice when many threads will share access to a common collection.
* Like most other concurrent collection implementations, t**his class does not permit the use of null elements.**
* This implementation employs an efficient non-blocking algorithm based on one described in Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms by Maged M. Michael and Michael L. Scott.
* Iterators are weakly consistent, returning elements reflecting the state of the queue at some point at or since the creation of the iterator. They do not throw ConcurrentModificationException, and may proceed concurrently with other operations. Elements contained in the queue since the creation of the iterator will be returned exactly once.
* **Beware that, unlike in most collections, the size method is NOT a constant-time operation.** Because of the asynchronous nature of these queues, determining the current number of elements requires a traversal of the elements, and so may report inaccurate results if this collection is modified during traversal.
* Additionally, the bulk operations addAll, removeAll, retainAll, containsAll, equals, and toArray are not guaranteed to be performed atomically. For example, an iterator operating concurrently with an addAll operation might view only some of the added elements.
* This class and its iterator implement all of the optional methods of the Queue and Iterator interfaces.
* Memory consistency effects: As with other concurrent collections, actions in a thread prior to placing an object into a ConcurrentLinkedQueue happen-before actions subsequent to the access or removal of that element from the ConcurrentLinkedQueue in another thread.

**java.util.concurrent.ConcurrentLinkedDeque<E>**

**public class ConcurrentLinkedDeque<E> extends AbstractCollection<E> implements Deque<E>, Serializable**

* An unbounded concurrent deque based on linked nodes.
* Concurrent insertion, removal, and access operations execute safely across multiple threads.
* A **ConcurrentLinkedDeque** is an appropriate choice when many threads will share access to a common collection.
* Like most other concurrent collection implementations, **this class does not permit the use of null elements.**
* Iterators and spliterators are weakly consistent.
* **Beware that, unlike in most collections, the size method is NOT a constant-time operation.** Because of the asynchronous nature of these deques, determining the current number of elements requires a traversal of the elements, and so may report inaccurate results if this collection is modified during traversal.
* **Additionally**, the bulk operations addAll, removeAll, retainAll, containsAll, equals, and toArray are not guaranteed to be performed atomically. For example, an iterator operating concurrently with an addAll operation might view only some of the added elements.
* This class and its iterator implement all of the optional methods of the Deque and Iterator interfaces.
* Memory consistency effects: As with other concurrent collections, actions in a thread prior to placing an object into a ConcurrentLinkedDeque happen-before actions subsequent to the access or removal of that element from the ConcurrentLinkedDeque in another thread.

**java.util.concurrent.DelayQueue<E extends Delayed>**

**public class DelayQueue<E extends Delayed> extends AbstractQueue<E> implements BlockingQueue<E>**

An unbounded blocking queue of Delayed elements, in which an element can only be taken when its delay has expired. The head of the queue is that Delayed element whose delay expired furthest in the past. If no delay has expired there is no head and poll will return null. Expiration occurs when an element's getDelay(TimeUnit.NANOSECONDS) method returns a value less than or equal to zero. Even though unexpired elements cannot be removed using take or poll, they are otherwise treated as normal elements. For example, the size method returns the count of both expired and unexpired elements. This queue does not permit null elements.

This class and its iterator implement all of the optional methods of the Collection and Iterator interfaces. The Iterator provided in method iterator() is not guaranteed to traverse the elements of the DelayQueue in any particular order.

**java.util.concurrent.LinkedBlockingQueue<E>**

**public class LinkedBlockingQueue<E> extends AbstractQueue<E> implements BlockingQueue<E>, Serializable**

* An optionally-bounded blocking queue based on linked nodes.
* This queue orders elements FIFO (first-in-first-out). The head of the queue is that element that has been on the queue the longest time. The tail of the queue is that element that has been on the queue the shortest time. New elements are inserted at the tail of the queue, and the queue retrieval operations obtain elements at the head of the queue.
* Linked queues typically have higher throughput than array-based queues but less predictable performance in most concurrent applications.
* The optional capacity bound constructor argument serves as a way to prevent excessive queue expansion. The capacity, if unspecified, is equal to Integer.MAX\_VALUE. Linked nodes are dynamically created upon each insertion unless this would bring the queue above capacity.
* This class and its iterator implement all of the optional methods of the Collection and Iterator interfaces.

**java.util.concurrent.LinkedBlockingDeque<E>**

**public class LinkedBlockingDeque<E> extends AbstractQueue<E> implements BlockingDeque<E>, Serializable**

* An optionally-bounded blocking deque based on linked nodes.
* The optional capacity bound constructor argument serves as a way to prevent excessive expansion. The capacity, if unspecified, is equal to Integer.MAX\_VALUE. Linked nodes are dynamically created upon each insertion unless this would bring the deque above capacity.
* Most operations run in constant time (ignoring time spent blocking). Exceptions include remove, removeFirstOccurrence, removeLastOccurrence, contains, iterator.remove(), and the bulk operations, all of which run in linear time.
* This class and its iterator implement all of the optional methods of the Collection and Iterator interfaces.

**java.util.concurrent.LinkedTransferQueue<E>**

**public class LinkedTransferQueue<E> extends AbstractQueue<E> implements TransferQueue<E>, Serializable**

* An unbounded TransferQueue based on linked nodes.
* This queue orders elements FIFO (first-in-first-out) with respect to any given producer. The head of the queue is that element that has been on the queue the longest time for some producer. The tail of the queue is that element that has been on the queue the shortest time for some producer.
* Beware that, unlike in most collections, the size method is NOT a constant-time operation. Because of the asynchronous nature of these queues, determining the current number of elements requires a traversal of the elements, and so may report inaccurate results if this collection is modified during traversal. Additionally, the bulk operations addAll, removeAll, retainAll, containsAll, equals, and toArray are not guaranteed to be performed atomically. For example, an iterator operating concurrently with an addAll operation might view only some of the added elements.
* This class and its iterator implement all of the optional methods of the Collection and Iterator interfaces.
* Memory consistency effects: As with other concurrent collections, actions in a thread prior to placing an object into a LinkedTransferQueue happen-before actions subsequent to the access or removal of that element from the LinkedTransferQueue in another thread.

**java.util.concurrent.PriorityBlockingQueue<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() is 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.

**java.util.concurrent.SynchronousQueue<E>**

**public class SynchronousQueue<E> extends AbstractQueue<E> implements BlockingQueue<E>, Serializable**

* A blocking queue in which each insert operation must wait for a corresponding remove operation by another thread, and vice versa.
* A synchronous queue does not have any internal capacity, not even a capacity of one.
* You cannot peek at a synchronous queue because an element is only present when you try to remove it;
* you cannot insert an element (using any method) unless another thread is trying to remove it;
* you cannot iterate as there is nothing to iterate.
* The head of the queue is the element that the first queued inserting thread is trying to add to the queue; if there is no such queued thread then no element is available for removal and poll() will return null.
* For purposes of other Collection methods (for example contains), a SynchronousQueue acts as an empty collection.
* This queue does not permit null elements.
* Synchronous queues are similar to rendezvous channels used in CSP and Ada. They are well suited for handoff designs, in which an object running in one thread must sync up with an object running in another thread in order to hand it some information, event, or task.
* This class supports an optional fairness policy for ordering waiting producer and consumer threads.
* By default, this ordering is not guaranteed. However, a queue constructed with fairness set to true grants threads access in FIFO order.
* This class and its iterator implement all of the optional methods of the Collection and Iterator interfaces.