Advanced Data Structure – Part-I-2017

**java.util.concurrent   
Interface BlockingQueue<E>**

**All Known Implementing Classes:**

[ArrayBlockingQueue](http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/ArrayBlockingQueue.html), [DelayQueue](http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/DelayQueue.html), [LinkedBlockingDeque](http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/LinkedBlockingDeque.html), [LinkedBlockingQueue](http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/LinkedBlockingQueue.html), [PriorityBlockingQueue](http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/PriorityBlockingQueue.html), [SynchronousQueue](http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/SynchronousQueue.html)

A [Queue](http://docs.oracle.com/javase/6/docs/api/java/util/Queue.html) 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:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | *Throws exception* | *Special value* | ***Blocks*** | *Times out* |
| **Insert** | [add(e)](http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/BlockingQueue.html#add(E)) | [offer(e)](http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/BlockingQueue.html#offer(E)) | [put(e)](http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/BlockingQueue.html#put(E)) | [offer(e, time, unit)](http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/BlockingQueue.html#offer(E, long, java.util.concurrent.TimeUnit)) |
| **Remove** | [remove()](http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/BlockingQueue.html#remove(java.lang.Object)) | [poll()](http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/BlockingQueue.html#poll(long, java.util.concurrent.TimeUnit)) | [take()](http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/BlockingQueue.html#take()) | [poll(time, unit)](http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/BlockingQueue.html#poll(long, java.util.concurrent.TimeUnit)) |
| **Examine** | [element()](http://docs.oracle.com/javase/6/docs/api/java/util/Queue.html#element()) | [peek()](http://docs.oracle.com/javase/6/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 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](http://docs.oracle.com/javase/6/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.

# [Difference between ArrayBlockingQueue and LinkedBlockingQueue](https://stackoverflow.com/questions/18375334/what-is-the-difference-between-arrayblockingqueue-and-linkedblockingqueue)

**ArrayBlockingQueue** is backed by an array that size will never change after creation. Setting the capacity to Integer.MAX\_VALUE would create a big array with high costs in space. **ArrayBlockingQueue is always bounded**.

**LinkedBlockingQueue** creates nodes dynamically until the capacity is reached. This is by default Integer.MAX\_VALUE. Using such a big capacity has no extra costs in space. LinkedBlockingQueueis optionally bounded.

## java.util.concurrent  Class LinkedBlockingDeque<E>

An optionally-bounded [blocking deque](http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/BlockingDeque.html) 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](http://docs.oracle.com/javase/6/docs/api/java/lang/Integer.html#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). 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.

## java.util.concurrent  Class PriorityBlockingQueue<E>

**An unbounded**[**blocking queue**](http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/BlockingQueue.html)**that uses the same ordering rules as class [PriorityQueue](http://docs.oracle.com/javase/6/docs/api/java/util/PriorityQueue.html" \o "class in java.util)**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](http://docs.oracle.com/javase/6/docs/api/java/lang/Comparable.html) 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](http://docs.oracle.com/javase/6/docs/api/java/util/Collection.html) and [Iterator](http://docs.oracle.com/javase/6/docs/api/java/util/Iterator.html) interfaces. **The Iterator provided in method**[**iterator()**](http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/PriorityBlockingQueue.html#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. For example, here is a class that applies first-in-first-out tie-breaking to comparable elements. To use it, you would insert a new FIFOEntry(anEntry) instead of a plain entry object.

## java.util.concurrent  Class SynchronousQueue<E>

**A**[**blocking queue**](http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/BlockingQueue.html)**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.