# Concurrent Collections

# Concurrent Collections

The java.util.concurrent package includes a number of additions to the Java Collections Framework. These are most easily categorized by the collection interfaces provided:

* [BlockingQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html) defines a first-in-first-out data structure that blocks or times out when you attempt to add to a full queue, or retrieve from an empty queue.
* [ConcurrentMap](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentMap.html) is a subinterface of [java.util.Map](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html" \t "_blank) that defines useful atomic operations. These operations remove or replace a key-value pair only if the key is present, or add a key-value pair only if the key is absent. Making these operations atomic helps avoid synchronization. The standard general-purpose implementation of ConcurrentMap is[ConcurrentHashMap](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html), which is a concurrent analog of [HashMap](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html" \t "_blank).
* [ConcurrentNavigableMap](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentNavigableMap.html) is a subinterface of ConcurrentMap that supports approximate matches. The standard general-purpose implementation of ConcurrentNavigableMap is [ConcurrentSkipListMap](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentSkipListMap.html" \t "_blank), which is a concurrent analog of [TreeMap](https://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html" \t "_blank).

All of these collections help avoid [Memory Consistency Errors](https://docs.oracle.com/javase/tutorial/essential/concurrency/memconsist.html) by defining a happens-before relationship between an operation that adds an object to the collection with subsequent operations that access or remove that objec

## Interface BlockingQueue<E>

**All Known Subinterfaces:**

[BlockingDeque](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingDeque.html)<E>, [TransferQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/TransferQueue.html" \o "interface in java.util.concurrent)<E>

**All Known Implementing Classes:**

[ArrayBlockingQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ArrayBlockingQueue.html), [DelayQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/DelayQueue.html), [LinkedBlockingDeque](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/LinkedBlockingDeque.html), [LinkedBlockingQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/LinkedBlockingQueue.html), [LinkedTransferQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/LinkedTransferQueue.html), [PriorityBlockingQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/PriorityBlockingQueue.html), [SynchronousQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/SynchronousQueue.html)

A [Queue](https://docs.oracle.com/javase/8/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:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 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 remainingCapacity beyond which no additional elements can be put without blocking. A BlockingQueuewithout any intrinsic capacity constraints always reports a remaining capacity of Integer.MAX\_VALUE.

2) BlockingQueue can be bounded or unbounded.

A bounded BlockingQueue is one which is initialized with initial capacity and call to put() will be blocked if BlockingQueue is full

and size is equal to capacity. This bounding nature makes it ideal to use a shared queue between multiple threads like in most common Producer consumer solutions in Java. An unbounded Queue is one which is initialized without capacity, actually by default it initialized with Integer.MAX\_VALUE.

most common example of BlockingQueue uses bounded BlockingQueue as shown in below example.

ArrayBlockingQueue and LinkedBlockingQueue in Java

ArrayBlockingQueue and LinkedBlockingQueue are common implementation of BlockingQueue<E> interface. ArrayBlockingQueue is backed by array and Queue impose orders as FIFO. head of the queue is the oldest element in terms of time and tail of the queue is youngest element. ArrayBlockingQueue is also fixed size bounded buffer on the other hand LinkedBlockingQueue is an optionally bounded queue built on top of Linked nodes. In terms of throughput LinkedBlockingQueue provides higher throughput than ArrayBlockingQueue in Java.

4. Deque and BlockingDeque

Deque interface is added in Java 6 and it extends Queue interface to support insertion and removal from both end of Queue referred as head and tail. Java6 also provides concurrent implementation of Deque like ArrayDeque and LinkedBlockingDeque. Deque Can be used efficiently to increase parallelism in program by allowing set of worker thread to help each other by taking some of work load from other thread by utilizing Deque double end consumption property. So if all Thread has there own set of task Queue and they are consuming from head; helper thread can also share some work load via consumption from tail.

The PriorityBlockingQueue class implements the BlockingQueue interface. Read the BlockingQueue text for more information about the interface.

The PriorityBlockingQueue is an unbounded concurrent queue. It uses the same ordering rules as the java.util.PriorityQueue class. You cannot insert null into this queue.

All elements inserted into the PriorityBlockingQueue must implement the java.lang.Comparable interface. The elements thus order themselves according to whatever priority you decide in your Comparable implementation.

Notice that the PriorityBlockingQueue does not enforce any specific behaviour for elements that have equal priority (compare() == 0).

Also notice, that in case you obtain an Iterator from a PriorityBlockingQueue, the Iterator does not guarantee to iterate the elements in priority order.

Here is an example of how to use the PriorityBlockingQueue:

BlockingQueue queue = new PriorityBlockingQueue();

//String implements java.lang.Comparable

queue.put("Value");

String value = queue.take();

public class **LinkedBlockingDeque<E>**

extends [AbstractQueue](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractQueue.html)<E>

implements [BlockingDeque](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingDeque.html)<E>, [Serializable](https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html)

An optionally-bounded [blocking deque](https://docs.oracle.com/javase/8/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](https://docs.oracle.com/javase/8/docs/api/java/lang/Integer.html" \l "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](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/LinkedBlockingDeque.html#remove-java.lang.Object-), [removeFirstOccurrence](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/LinkedBlockingDeque.html#removeFirstOccurrence-java.lang.Object-), [removeLastOccurrence](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/LinkedBlockingDeque.html#removeLastOccurrence-java.lang.Object-), [contains](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/LinkedBlockingDeque.html#contains-java.lang.Object-), [iterator.remove()](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/LinkedBlockingDeque.html#iterator--), 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](https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html) and [Iterator](https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html" \o "interface in java.util) interfaces.

This class is a member of the [Java Collections Framework](https://docs.oracle.com/javase/8/docs/technotes/guides/collections/index.html).

java.util.concurrent

## Interface ConcurrentMap<K,V>

**All Known Subinterfaces:**

[ConcurrentNavigableMap](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentNavigableMap.html)<K,V>

**All Known Implementing Classes:**

[ConcurrentHashMap](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html), [ConcurrentSkipListMap](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentSkipListMap.html" \o "class in java.util.concurrent)

public class **ConcurrentHashMap<K,V>**

extends [AbstractMap](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractMap.html)<K,V>

implements [ConcurrentMap](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentMap.html)<K,V>, [Serializable](https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html)

A hash table supporting full concurrency of retrievals and high expected concurrency for updates. This class obeys the same functional specification as [Hashtable](https://docs.oracle.com/javase/8/docs/api/java/util/Hashtable.html" \o "class in java.util), and includes versions of methods corresponding to each method of Hashtable. However, even though all operations are thread-safe, retrieval operations do *not* entail locking, and there is *not* any support for locking the entire table in a way that prevents all access. This class is fully interoperable with Hashtable in programs that rely on its thread safety but not on its synchronization details.