**BlockingQueue**

* [BlockingQueue Usage](http://tutorials.jenkov.com/java-util-concurrent/blockingqueue.html#blockingqueue-usage)
* [BlockingQueue Implementations](http://tutorials.jenkov.com/java-util-concurrent/blockingqueue.html#blockingqueue-implementations)
* [Java BlockingQueue Example](http://tutorials.jenkov.com/java-util-concurrent/blockingqueue.html#java-blockingqueue-example)

The Java BlockingQueue interface in the java.util.concurrent package represents a queue which is thread safe to put into, and take instances from. In this text I will show you how to use this BlockingQueue.

This text will not discuss how to implement a BlockingQueue in Java yourself. If you are interested in that, I have a text on [**Blocking Queues**](http://tutorials.jenkov.com/java-concurrency/blocking-queues.html) in my more theoretical [**Java Concurrency Tutorial**](http://tutorials.jenkov.com/java-concurrency/index.html).

**BlockingQueue Usage**

A BlockingQueue is typically used to have on thread produce objects, which another thread consumes. Here is a diagram that illustrates this principle:

|  |
| --- |
| A BlockingQueue with one thread putting into it, and another thread taking from it. |
| **A BlockingQueue with one thread putting into it, and another thread taking from it.** |

The producing thread will keep producing new objects and insert them into the queue, until the queue reaches some upper bound on what it can contain. It's limit, in other words. If the blocking queue reaches its upper limit, the producing thread is blocked while trying to insert the new object. It remains blocked until a consuming thread takes an object out of the queue.

The consuming thread keeps taking objects out of the blocking queue, and processes them. If the consuming thread tries to take an object out of an empty queue, the consuming thread is blocked until a producing thread puts an object into the queue.

**BlockingQueue Methods**

A BlockingQueue has 4 different sets of methods for inserting, removing and examining the elements in the queue. Each set of methods behaves differently in case the requested operation cannot be carried out immediately. Here is a table of the methods:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Throws Exception** | **Special Value** | **Blocks** | **Times Out** |
| **Insert** | add(o) | offer(o) | put(o) | offer(o, timeout, timeunit) |
| **Remove** | remove(o) | poll() | take() | poll(timeout, timeunit) |
| **Examine** | element() | peek() |  |  |

The 4 different sets of behaviour means this:

1. **Throws Exception**:   
   If the attempted operation is not possible immediately, an exception is thrown.
2. **Special Value**:   
   If the attempted operation is not possible immediately, a special value is returned (often true / false).
3. **Blocks**:   
   If the attempted operation is not possible immedidately, the method call blocks until it is.
4. **Times Out**:   
   If the attempted operation is not possible immedidately, the method call blocks until it is, but waits no longer than the given timeout. Returns a special value telling whether the operation succeeded or not (typically true / false).

It is not possible to insert null into a BlockingQueue. If you try to insert null, the BlockingQueue will throw a NullPointerException.

It is also possible to access all the elements inside a BlockingQueue, and not just the elements at the start and end. For instance, say you have queued an object for processing, but your application decides to cancel it. You can then call e.g. remove(o) to remove a specific object in the queue. However, this is not done very efficiently, so you should not use these Collection methods unless you really have to.

**BlockingQueue Implementations**

Since BlockingQueue is an interface, you need to use one of its implementations to use it. The java.util.concurrent package has the following implementations of the BlockingQueue interface (in Java 6):

* [**ArrayBlockingQueue**](http://tutorials.jenkov.com/java-util-concurrent/arrayblockingqueue.html)
* [**DelayQueue**](http://tutorials.jenkov.com/java-util-concurrent/delayqueue.html)
* [**LinkedBlockingQueue**](http://tutorials.jenkov.com/java-util-concurrent/linkedblockingqueue.html)
* [**PriorityBlockingQueue**](http://tutorials.jenkov.com/java-util-concurrent/priorityblockingqueue.html)
* [**SynchronousQueue**](http://tutorials.jenkov.com/java-util-concurrent/synchronousqueue.html)

Click the links in the list to read more about each implementation. If a link cannot be clicked, that implementation has not yet been described. Check back again in the future, or check out the JavaDoc's for more detail.

**Java BlockingQueue Example**

Here is a Java BlockingQueue example. The example uses the ArrayBlockingQueue implementation of the BlockingQueue interface.

First, the BlockingQueueExample class which starts a Producer and a Consumer in separate threads. The Producer inserts strings into a shared BlockingQueue, and the Consumer takes them out.

public class BlockingQueueExample {

public static void main(String[] args) throws Exception {

BlockingQueue queue = new ArrayBlockingQueue(1024);

Producer producer = new Producer(queue);

Consumer consumer = new Consumer(queue);

new Thread(producer).start();

new Thread(consumer).start();

Thread.sleep(4000);

}

}

Here is the Producer class. Notice how it sleeps a second between each put() call. This will cause the Consumer to block, while waiting for objects in the queue.

public class Producer implements Runnable{

protected BlockingQueue queue = null;

public Producer(BlockingQueue queue) {

this.queue = queue;

}

public void run() {

try {

queue.put("1");

Thread.sleep(1000);

queue.put("2");

Thread.sleep(1000);

queue.put("3");

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

Here is the Consumer class. It just takes out the objects from the queue, and prints them to System.out.

public class Consumer implements Runnable{

protected BlockingQueue queue = null;

public Consumer(BlockingQueue queue) {

this.queue = queue;

}

public void run() {

try {

System.out.println(queue.take());

System.out.println(queue.take());

System.out.println(queue.take());

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

# ArrayBlockingQueue

|  |
| --- |
|  |

The ArrayBlockingQueue class implements the **[BlockingQueue](http://tutorials.jenkov.com/java-util-concurrent/blockingqueue.html)** interface. Read the **[BlockingQueue](http://tutorials.jenkov.com/java-util-concurrent/blockingqueue.html)** text for more information about the interface.

ArrayBlockingQueue is a bounded, blocking queue that stores the elements internally in an array. That it is bounded means that it cannot store unlimited amounts of elements. There is an upper bound on the number of elements it can store at the same time. You set the upper bound at instantiation time, and after that it cannot be changed.

The ArrayBlockingQueue stores the elements internally in FIFO (First In, First Out) order. The head of the queue is the element which has been in queue the longest time, and the tail of the queue is the element which has been in the queue the shortest time.

Here is how to instantiate and use an ArrayBlockingQueue:

BlockingQueue queue = new ArrayBlockingQueue(1024);

queue.put("1");

Object object = queue.take();

Here is a BlockingQueue example that uses Java Generics. Notice how you can put and take String's instead of :

BlockingQueue<String> queue = new ArrayBlockingQueue<String>(1024);

queue.put("1");

String string = queue.take();

# LinkedBlockingQueue

|  |
| --- |
|  |

The LinkedBlockingQueue class implements the **[BlockingQueue](http://tutorials.jenkov.com/java-util-concurrent/blockingqueue.html)** interface. Read the **[BlockingQueue](http://tutorials.jenkov.com/java-util-concurrent/blockingqueue.html)** text for more information about the interface.

The LinkedBlockingQueue keeps the elements internally in a linked structure (linked nodes). This linked structure can optionally have an upper bound if desired. If no upper bound is specified, Integer.MAX\_VALUE is used as the upper bound.

The LinkedBlockingQueue stores the elements internally in FIFO (First In, First Out) order. The head of the queue is the element which has been in queue the longest time, and the tail of the queue is the element which has been in the queue the shortest time.

Here is how to instantiate and use a LinkedBlockingQueue:

BlockingQueue<String> unbounded = new LinkedBlockingQueue<String>();

BlockingQueue<String> bounded = new LinkedBlockingQueue<String>(1024);

bounded.put("Value");

String value = bounded.take();

# PriorityBlockingQueue

|  |
| --- |
|  |

The PriorityBlockingQueue class implements the **[BlockingQueue](http://tutorials.jenkov.com/java-util-concurrent/blockingqueue.html)** interface. Read the **[BlockingQueue](http://tutorials.jenkov.com/java-util-concurrent/blockingqueue.html)** 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 Comparableimplementation.

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();