BlockingQueue

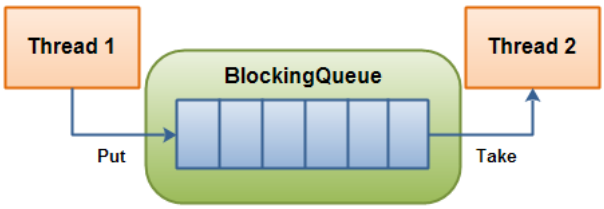
Jakob Jenkov
Last update: 2014-06-23

The Java `BlockingQueue` interface in the `java.util.concurrent` package represents a queue wh 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 intereste that, I have a text on **Blocking Queues** in my more theoretical **Java Concurrency Tutorial**.

BlockingQueue Usage

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



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 q 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 re 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 th consuming thread tries to take an object out of an empty queue, the consuming thread is blocke 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 elem the queue. Each set of methods behaves differently in case the requested operation cannot be c 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 tru 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, bu
   no longer than the given timeout. Returns a special value telling whether the operation succ
   or not (typically true / false).

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

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

BlockingQueue Implementations

Since `BlockingQueue` is an interface, you need to use one of its implementations to use it. The

- **ArrayBlockingQueue**
- **DelayQueue**
- **LinkedBlockingQueue**
- **PriorityBlockingQueue**
- **SynchronousQueue**

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 Jav for more detail.

Java BlockingQueue Example

Here is a Java `BlockingQueue` example. The example uses the `ArrayBlockingQueue` implementa the `BlockingQueue` interface.

First, the `BlockingQueueExample` class which starts a `Producer` and a `Consumer` in separate threa 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 cau `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 `Syste`

```
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();
        }
    }
}
```

Next: **ArrayBlockingQueue**

Jakob Jenkov

All Trails | Trail TOC | Page TOC | Previous | Next