**Java BlockingQueue**

The BlockingQueue interface of the Java Collections framework extends the Queue interface. It allows any operation to wait until it can be successfully performed.

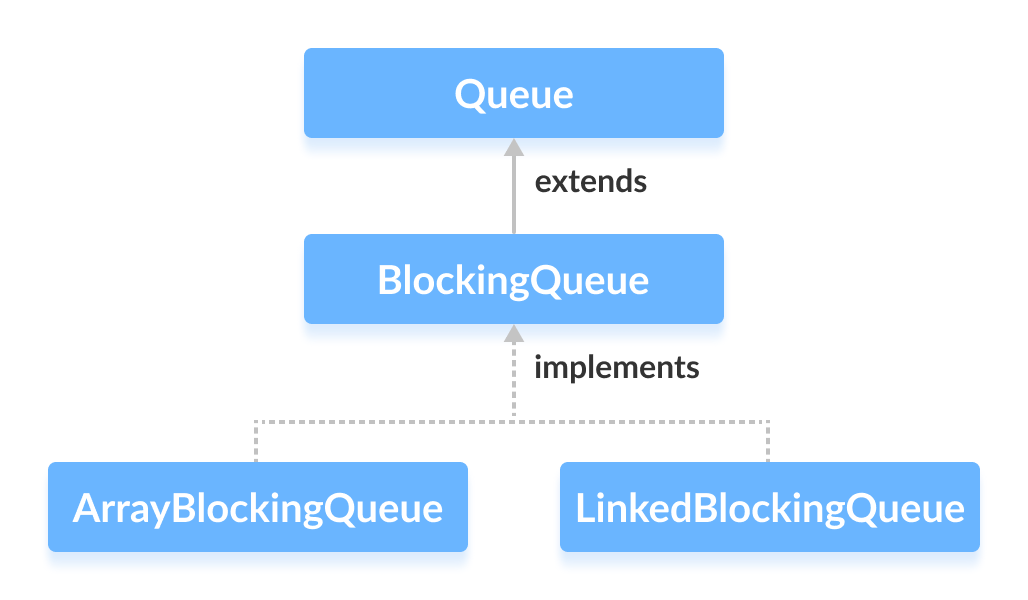
For example, if we want to delete an element from an empty queue, then the blocking queue allows the delete operation to wait until the queue contains some elements to be deleted.

## Classes that Implement BlockingQueue

Since BlockingQueue is an interface, we cannot provide the direct implementation of it.

In order to use the functionality of the BlockingQueue, we need to use classes that implement it.

* [ArrayBlockingQueue](https://www.programiz.com/java-programming/arrayblockingqueue)
* [LinkedBlockingQueue](https://www.programiz.com/java-programming/linkedblockingqueue)



## How to use blocking queues?

We must import the java.util.concurrent.BlockingQueue package in order to use BlockingQueue.

// Array implementation of BlockingQueue

BlockingQueue<String> animal1 = new ArraryBlockingQueue<>();

// LinkedList implementation of BlockingQueue

BlockingQueue<String> animal2 = new LinkedBlockingQueue<>();

Here, we have created objects animal1 and animal2 of classes ArrayBlockingQueue and LinkedBlockingQueue, respectively. These objects can use the functionalities of the BlockingQueue interface.

## Methods of BlockingQueue

Based on whether a queue is full or empty, methods of a blocking queue can be divided into 3 categories:

### Methods that throw an exception

* add() - Inserts an element to the blocking queue at the end of the queue. Throws an exception if the queue is full.
* element() - Returns the head of the blocking queue. Throws an exception if the queue is empty.
* remove() - Removes an element from the blocking queue. Throws an exception if the queue is empty.

### Methods that return some value

* offer() - Inserts the specified element to the blocking queue at the end of the queue. Returns false if the queue is full.
* peek() - Returns the head of the blocking queue. Returns null if the queue is empty.
* poll() - Removes an element from the blocking queue. Returns null if the queue is empty.

**More on offer() and poll()**

The offer() and poll() method can be used with timeouts. That is, we can pass time units as a parameter. For example,

offer(value, 100, milliseconds)

Here,

* value is the element to be inserted to the queue
* And we have set a timeout of 100 milliseconds

This means the offer() method will try to insert an element to the blocking queue for 100 milliseconds. If the element cannot be inserted in 100 milliseconds, the method returns false.

**Note:** Instead of milliseconds, we can also use these time units: days, hours, minutes, seconds, microseconds and nanoseconds in offer() and poll() methods.

### Methods that blocks the operation

The BlockingQueue also provides methods to block the operations and wait if the queue is full or empty.

* put() - Inserts an element to the blocking queue. If the queue is full, it will wait until the queue has space to insert an element.
* take() - Removes and returns an element from the blocking queue. If the queue is empty, it will wait until the queue has elements to be deleted.

Suppose, we want to insert elements into a queue. If the queue is full then the put() method will wait until the queue has space to insert elements.

Similarly, if we want to delete elements from a queue. If the queue is empty then the take() method will wait until the queue contains elements to be deleted.

## Implementation of BlockingQueue in ArrayBlockingQueue

import java.util.concurrent.BlockingQueue;

import java.util.concurrent.ArrayBlockingQueue;

class Main {

public static void main(String[] args) {

// Create a blocking queue using the ArrayBlockingQueue

BlockingQueue<Integer> numbers = new ArrayBlockingQueue<>(5);

try {

// Insert element to blocking queue

numbers.put(2);

numbers.put(1);

numbers.put(3);

System.out.println("BLockingQueue: " + numbers);

// Remove Elements from blocking queue

int removedNumber = numbers.take();

System.out.println("Removed Number: " + removedNumber);

}

catch(Exception e) {

e.getStackTrace();

}

}

}

**Output**

BlockingQueue: [2, 1, 3]

Removed Element: 2

To learn more about ArrayBlockingQueue, visit Java ArrayBlockingQueue.

## Why BlockingQueue?

In Java, BlockingQueue is considered as the **thread-safe** collection. It is because it can be helpful in multi-threading operations.

Suppose one thread is inserting elements to the queue and another thread is removing elements from the queue.

Now, if the first thread runs slower, then the blocking queue can make the second thread wait until the first thread completes its operation.