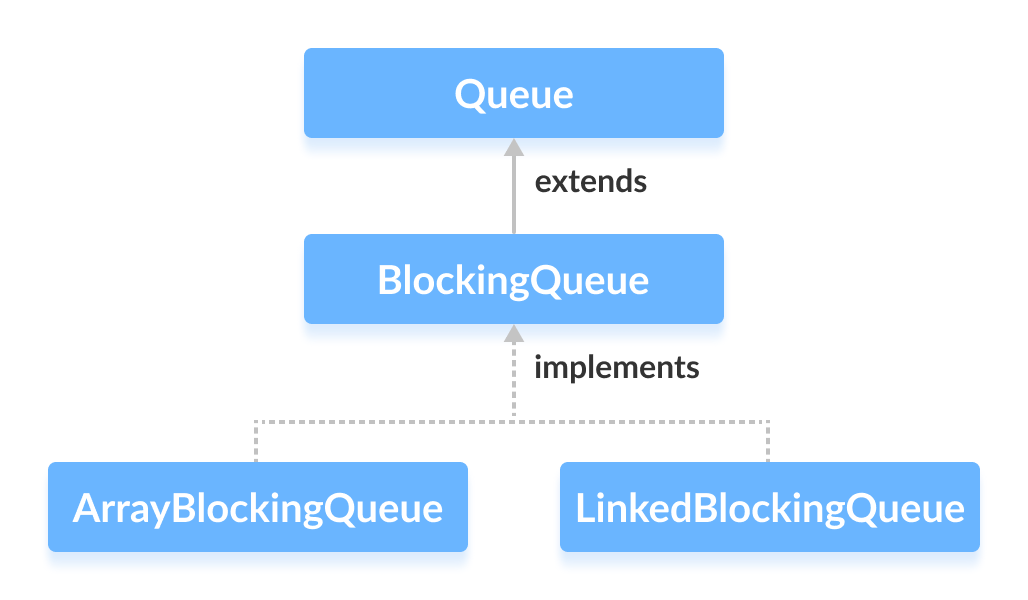
**Java ArrayBlockingQueue**

The ArrayBlockingQueue class of the Java Collections framework provides the blocking queue implementation using an array.

It implements the Java BlockingQueue interface.



## Creating ArrayBlockingQueue

In order to create an array blocking queue, we must import the java.util.concurrent.ArrayBlockingQueue package.

Once we import the package, here is how we can create an array blocking queue in Java:

ArrayBlockingQueue<Type> animal = new ArrayBlockingQueue<>(int capacity);

Here,

* Type - the type of the array blocking queue
* capacity - the size of the array blocking queue

For example,

// Creating String type ArrayBlockingQueue with size 5

ArrayBlockingQueue<String> animals = new ArrayBlockingQueue<>(5);

// Creating Integer type ArrayBlockingQueue with size 5

ArrayBlockingQueue<Integer> age = new ArrayBlockingQueue<>(5);

**Note:** **It is compulsory to provide the size of the array.**

## Methods of ArrayBlockingQueue

The ArrayBlockingQueue class provides the implementation of all the methods in the BlockingQueue interface.

These methods are used to insert, access and delete elements from array blocking queues.

Also, we will learn about two methods put() and take() that support the blocking operation in the array blocking queue.

These two methods distinguish the array blocking queue from other typical queues.

### Insert Elements

* add() - Inserts the specified element to the array blocking queue. It throws an exception if the queue is full.
* offer() - Inserts the specified element to the array blocking queue. It returns false if the queue is full.

For example,

import java.util.concurrent.ArrayBlockingQueue;

class Main {

public static void main(String[] args) {

ArrayBlockingQueue<String> animals = new ArrayBlockingQueue<>(5);

// Using add()

animals.add("Dog");

animals.add("Cat");

// Using offer()

animals.offer("Horse");

System.out.println("ArrayBlockingQueue: " + animals);

}

}

**Output**

ArrayBlockingQueue: [Dog, Cat, Horse]

### Access Elements

* peek() - Returns an element from the front of the array blocking queue. It returns null if the queue is empty.
* iterator() - Returns an iterator object to sequentially access elements from the array blocking queue. It throws an exception if the queue is empty. We must import the java.util.Iterator package to use it.

For example,

import java.util.concurrent.ArrayBlockingQueue;

import java.util.Iterator;

class Main {

public static void main(String[] args) {

ArrayBlockingQueue<String> animals = new ArrayBlockingQueue<>(5);

// Add elements

animals.add("Dog");

animals.add("Cat");

animals.add("Horse");

System.out.println("ArrayBlockingQueue: " + animals);

// Using peek()

String element = animals.peek();

System.out.println("Accessed Element: " + element);

// Using iterator()

Iterator<String> iterate = animals.iterator();

System.out.print("ArrayBlockingQueue Elements: ");

while(iterate.hasNext()) {

System.out.print(iterate.next());

System.out.print(", ");

}

}

}

**Output**

ArrayBlockingQueue: [Dog, Cat, Horse]

Accessed Element: Dog

ArrayBlockingQueue Elements: Dog, Cat, Horse,

### Remove Elements

* remove() - Returns and removes a specified element from the array blocking queue. It throws an exception if the queue is empty.
* poll() - Returns and removes a specified element from the array blocking queue. It returns null if the queue is empty.
* clear() - Removes all the elements from the array blocking queue.

For example,

import java.util.concurrent.ArrayBlockingQueue;

class Main {

public static void main(String[] args) {

ArrayBlockingQueue<String> animals = new ArrayBlockingQueue<>(5);

animals.add("Dog");

animals.add("Cat");

animals.add("Horse");

System.out.println("ArrayBlockingQueue: " + animals);

// Using remove()

String element1 = animals.remove();

System.out.println("Removed Element:");

System.out.println("Using remove(): " + element1);

// Using poll()

String element2 = animals.poll();

System.out.println("Using poll(): " + element2);

// Using clear()

animals.clear();

System.out.println("Updated ArrayBlockingQueue: " + animals);

}

}

**Output**

ArrayBlockingQueue: [Dog, Cat, Horse]

Removed Elements:

Using remove(): Dog

Using poll(): Cat

Updated ArrayBlockingQueue: []

## put() and take() Method

In multithreading processes, we can use put() and take() to block the operation of one thread to synchronize it with another thread. These methods will wait until they can be successfully executed.

### put() method

To add an element to the end of an array blocking queue, we can use the put() method.

If the array blocking queue is full, it waits until there is space in the array blocking queue to add an element.

For example,

import java.util.concurrent.ArrayBlockingQueue;

class Main {

public static void main(String[] args) {

ArrayBlockingQueue<String> animals = new ArrayBlockingQueue<>(5);

try {

// Add elements to animals

animals.put("Dog");

animals.put("Cat");

System.out.println("ArrayBlockingQueue: " + animals);

}

catch(Exception e) {

System.out.println(e);

}

}

}

**Output**

ArrayBlockingQueue: [Dog, Cat]

Here, the put() method may throw an InterruptedException if it is interrupted while waiting. Hence, we must enclose it inside a try..catch block.

### take() Method

To **return and remove** an element from the front of the array blocking queue, we can use the take() method.

If the array blocking queue is empty, it waits until there are elements in the array blocking queue to be deleted.

For example,

import java.util.concurrent.ArrayBlockingQueue;

class Main {

public static void main(String[] args) {

ArrayBlockingQueue<String> animals = new ArrayBlockingQueue<>(5);

try {

//Add elements to animals

animals.put("Dog");

animals.put("Cat");

System.out.println("ArrayBlockingQueue: " + animals);

// Remove an element

String element = animals.take();

System.out.println("Removed Element: " + element);

}

catch(Exception e) {

System.out.println(e);

}

}

}

**Output**

ArrayBlockingQueue: [Dog, Cat]

Removed Element: Dog

Here, the take() method will throw an InterrupedException if it is interrupted while waiting. Hence, we must enclose it inside a try...catch block.

## Other Methods

|  |  |
| --- | --- |
| Methods | Descriptions |
| contains(element) | Searches the array blocking queue for the specified element. If the element is found, it returns true, if not it returns false. |
| size() | Returns the length of the array blocking queue. |
| toArray() | Converts array blocking queue to an array and returns it. |
| toString() | Converts the array blocking queue to string |

## Why use ArrayBlockingQueue?

The ArrayBlockingQueue uses arrays as its internal storage.

It is considered as a **thread-safe** collection. Hence, it is generally used in multi-threading applications.

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

Now, if the first thread is slower than the second thread, then the array blocking queue can make the second thread waits until the first thread completes its operations.