BlockingQueue in Java

Before directly jumping to the topic 'Blocking Queue' let us first understand Queue in brief. Queue is an ordered list of objects where insertions take place at the rear end of the list and deletion of elements takes place from the front end. Therefore, it is also said that Queue is based on FIFO ( First-In-First-Out ) principle.

BlockingQueue is a queue that additionally supports operations that wait for the queue to become non-empty when we are trying to retrieve an element, and wait for the space to be empty when an element is to be inserted in the queue. Java 1.5 provides support for BlockingQueue interface along with other concurrent utility classes.

**Some important points with respect to the Blocking Queue :**

* A BlockingQueue may have a remainingCapacity beyond which we cannot insert any element without blocking.
* All the implementations related to the BlockingQueue are thread-safe. All the methods achieve their events using internal locks or other forms of concurrency control.
* A BlockingQueue does not accept null elements. If we try to add a null value the implementation throws a NullPointerException.
* Java 5 provides the BlockingQueue implementations in the java.util.concurrent package.

BlockingQueue Implementing Classes

There is no provision of directly providing an instance of the BlockingQueue since it as an interface, so to implement BlockingQueue we need to create classes implementing it.

1. ArrayBlockingQueue
2. DelayQueue
3. LinkedBlockingDeque
4. LinkedBlockingQueue
5. LinkedTransferQueue
6. PriorityBlockingQueue
7. SynchronousQueue

LinkedBlockingQueue and ArrayBlockingQueue are the classes used to implement BlockingDeque class. These two classes are a composition of BlockingDeque and linked list data structure and BlockingDeque and array respectively.

**Syntax of using BlockingQueue :**

We make use of import statement in order to use the above classes and import java.util.concurrent.BlockingQueue package.

1. **import** java.util.concurrent.BlockingQueue ;
2. or
3. **import** java.util.concurrent.\* ;

**Declaring a BlockingQueue**

1. **public** **interface** BlockingQueue< E > **extends** Queue< E >

**Creating objects of BlockingQueue class using LinkedBlockingDeque**

1. BlockingQueue< E > objectName = **new** LinkedBlockingDeque< >( ) ;
2. Or
3. LinkedBlockingDeque< E > objectName = **new** LinkedBlockingDeque< >( );

**Creating objects of BlockingQueue class using ArrayBlockingQueue**

1. BlockingQueue< E > objectName = **new** ArrayBlockingQueue< >( ) ;

Methods of the BlockingQueue Interface

There are three categories of methods used in the implementation of BlockingQueue :

**1. Methods that throw an exception**

1. **add( ):** It inserts an element to the BlockingQueue at the end of the queue. It throws an exception when the queue is full.
2. **element( ):** It returns the top or the head of the queue. It throws an exception if the queue is empty.
3. **remove( ):** It removes an element from the BlockingQueue. It throws an exception if the queue is empty.

**2. Methods that return some value**

**i. offer( ):** It inserts the specific element to the BlockingQueue at the rear end of the queue. It returns false if the queue is full. The method can also be used with timeouts, that is time units can be passed as a parameter.

For example:

1. offer( value, 100, milliseconds ) ;

Here, value is the element to be inserted in the queue. The above method will insert an element to the BlockingQueue for 100 milliseconds. If the element cannot be inserted in 100 milliseconds, the method returns false.

**ii. peek( ):** It returns the top or head of the BlockingQueue. It returns null if the queue is empty.

**iii. poll( ):** It removes an element from the BlockingQueue. It returns null if the queue is empty. It can also be used with timeouts, that is time units can be passed as a parameter.

**3. Methods that block the operation**

1. **i. put( ):** The method inserts an element to the BlockingQueue. If in case the queue is full, the put( ) method waits until the queue has some vacant space to insert an element.
2. **ii. take( ):** The method removes and returns an element from the BlockingQueue. If in case the queue is empty, the take( ) method waits until the queue has some elements to be deleted.

BlockingQueue Types

There are two types of BlockingQueue:

**1. Unbounded Queue:** Unbounded blocking queue is the queue that never blocks because its size could be grown to a very large size. The capacity of the BlockingQueue will be set to Integer.MAX\_VALUE. When the elements are added, the size of the Unbounded queue grows.

**Syntax :**

1. BlockingQueue blockingQueue = **new** LinkedBlockingDeque( ) ;

**2. Bounded Queue:** Another type of the blocking queue is the bounded queue. It can be created by passing the capacity of the queue to the constructor of the queue.

**Syntax :**

1. // Creating a Blocking Queue of the bounded nature with capacity 10
2. BlockingQueue blockingQueue = **new** LinkedBlockingDeque( 10 ) ;

**Example :**

Let us consider an example that explains the concept of BlockingQueue.

1. // importing the packages that support BlockingQueue in Java
2. **import** java.util.concurrent.BlockingQueue ;
3. **import** java.util.concurrent.ArrayBlockingQueue ;
4. **public** **class** alphabets {
5. **public** **static** **void** main( String[ ] args ) {
6. // declaring the BlockingQueue of ' bounded ' nature using ArrayBlockingQueue
7. BlockingQueue< String > alph = **new** ArrayBlockingQueue< >( 7 ) ;
8. **try** {
9. // Insert element to blocking queue
10. alph.put( " A " ) ;
11. alph.put( " B " ) ;
12. alph.put( " C " ) ;
13. alph.put( " D " ) ;
14. alph.put( " E " ) ;
15. alph.put( " F " ) ;
16. alph.put( " G " ) ;
17. System.out.println( " Content of BLockingQueue : " + alph ) ;
18. // deleting some elements from the queue
19. String temp = alph.take( ) ;
20. System.out.println( " The number removed is : " + temp ) ;
21. // BlockingQueue after deleting one element
22. System.out.println( " Content of BLockingQueue after deleting one element : " + alph ) ;
23. }
24. **catch**( Exception e ) {
25. e.getStackTrace( ) ;
26. }
27. }
28. }

**Output:**

Content of BLockingQueue : [ A , B , C , D , E , F , G ]

The number removed is : A

Content of BLockingQueue after deleting one element : [ B , C , D , E , F , G ]

Basic Operations

Let us have a broader look over different operations that can be performed on BlockingQueue :

1. Adding elements
2. Accessing elements
3. Deleting elements
4. Iterating through the elements

1. Adding Elements

We can add elements into a ' LinkedBlockedDeque ' in different ways depending on the type of structure we are trying to use it as. The most common method used to add elements in the rear end of the deque is the ' add( ) ' method. There is another function named ' addAll( ) ' method to add an entire collection of the elements to LinkedBlockingDeque. In order to use the deque as a queue function like ' add( ) ' and ' put( ) ' can be added in the program.

**Example :**

1. // program to insert elements in the BlockingQueue
2. // importing packages that support BlockingQueue
3. **import** java.util.concurrent.LinkedBlockingDeque ;
4. **import** java.util.concurrent.BlockingQueue ;
5. **import** java.util.\* ;
6. **public** **class** alphabets {
7. **public** **static** **void** main( String[ ] args )
8. **throws** IllegalStateException
9. {
10. // creating object of the BlockingQueue class
11. BlockingQueue< String > alph
12. = **new** LinkedBlockingDeque< String >( ) ;
13. // adding alphabets to the Blocking Queue using add( ) method
14. alph.add( " A " ) ;
15. alph.add( " B " ) ;
16. alph.add( " C " ) ;
17. alph.add( " D " ) ;
18. alph.add( " E " ) ;
19. BlockingQueue< String > alph2
20. = **new** LinkedBlockingDeque< String >( ) ;
21. // adding collection of elements using addAll( ) method
22. alph2.addAll( alph ) ;
23. // before removing print BlockingQueue
24. System.out.println( " Contents of Blocking Queue : " + alph ) ;
25. System.out.println( " Contents of another Blocking Queue : " + alph2 ) ;
26. }
27. }

**Output:**

Contents of Blocking Queue : [ A , B , C , D , E ]

Contents of another Blocking Queue : [ A , B , C , D , E ]

2. Accessing Elements

We can access the elements of the ' LinkedBlockingDeque ' using methods like contains( ), element( ), peek( ), poll( ).

**Example :**

1. // accessing elements from the Blocking Queue
2. // importing the Dart package that support BlockingQueue
3. **import** java.util.concurrent.\* ;
5. **public** **class** alphabets {
6. **public** **static** **void** main( String[ ] args )
7. {
8. // creating an object of the class LinkedBlockingDeque
9. BlockingQueue< String > alph
10. = **new** LinkedBlockingDeque< String >( ) ;
12. // adding elements to the BlockingQueue using add( ) method
13. alph.add( " A " ) ;
14. alph.add( " B " ) ;
15. alph.add( " C " ) ;
16. alph.add( " D " ) ;
17. alph.add( " E " ) ;
18. // printing the elements of the BlockingQueue
19. System.out.println(
20. " The contents of the Linked Blocking Queue is : " ) ;
21. System.out.println( alph ) ;
22. // retrieving the element alphabet ' C ' from the queue
23. **if** ( alph.contains( " C " ) )
24. System.out.println(
25. " Yayy! Element C successfully founded in the queue " ) ;
26. **else**
27. System.out.println( " No such element exists in the queue. " ) ;
29. // Accessing the first element of the queue using the function element( )
30. String top = alph.element( ) ;
31. System.out.println( " The top element of the queue is : " + top ) ;
32. }
33. }

**Output:**

The contents of the Linked Blocking Queue is :

[ A , B , C , D , E ]

Yayy! Element C successfully founded in the queue

The top element of the queue is : A

3. Deleting Elements

Elements can be deleted from a LinkedBlockingDeque using remove(). Other methods such as take( ) and poll( ) can also be used in a way to remove the first and the last elements.

**Example :**

1. // program to delete elements from the queue
2. // importing the package that support BlockingQueue
3. **import** java.util.concurrent.\* ;
4. **public** **class** alphabets {
5. **public** **static** **void** main( String[] args )
6. {
7. // creating an object of the class LinkedBlockingDeque
8. BlockingQueue< String > alph
9. = **new** LinkedBlockingDeque< String >( ) ;
10. // adding the elements to the BlockingQueue using add( ) method
11. alph.add( " A " ) ;
12. alph.add( " B " ) ;
13. alph.add( " C " ) ;
14. alph.add( " D " ) ;
15. alph.add( " E " ) ;
16. // printing the elements of the blocking queue
17. System.out.println(
18. " The content of LinkedBlockingDeque is : " ) ;
19. System.out.println( alph ) ;
20. // removing the elements from the queue using the remove( ) function
21. alph.remove( " C " ) ;
22. alph.remove( " E " ) ;
23. // let us see what will happen if we try to remove an element that actually does not exist in the queue
24. alph.remove( " Y " ) ;
25. // Print the elements of the alph object of BlockingQueue
26. System.out.println(
27. " The content of the LinkedBlockingDeque after removing elements is : " ) ;
28. System.out.println( alph ) ;
29. }
30. }

**Output:**

The content of LinkedBlockingDeque is :

[ A , B , C , D , E ]

The content of the LinkedBlockingDeque after removing elements is :

[ A , B , D ]

4. Iterating through the Elements

In order to iterate through the elements of a ' LinkedBlockingDeque ' we can create an iterator and use the methods of the Iterable interface, which is the root of the Collection Framework of Java, to access the elements. The ' next( ) ' method of Iterable returns the element of any collection.

1. // program to iterate over the elements of the BlockingQueue
2. // importing the packages that support BlockingQueue
3. **import** java.util.Iterator ;
4. **import** java.util.concurrent.\* ;
5. **public** **class** alphabet {
6. **public** **static** **void** main( String[ ] args ) {
7. // creating an object of the LinkedBlockingDeque
8. BlockingQueue< String > alph = **new** LinkedBlockingDeque< String >( ) ;
9. // adding the elements to the BlockingQueue
10. alph.add( " A " ) ;
11. alph.add( " B " ) ;
12. alph.add( " C " ) ;
13. alph.add( " D " ) ;
14. alph.add( " E " ) ;
15. // Create an iterator to traverse through the BlockingQueue
16. Iterator< String > alphIter = alph.iterator( ) ;
17. // Print the elements of lbdq on to the console
18. System.out.println( " The content of the Linked Blocking Deque is : " ) ;
19. **for**( **int** i = 0 ; i < alph.size( ) ; i++ )
20. {
21. System.out.print( alphIter.next( ) + " ") ;
22. }
23. }
24. }

**Output:**

The content of the Linked Blocking Deque is :

A B C D E

BlockingQueue Methods

|  |  |
| --- | --- |
| **Method** | **Description** |
| **boolean add( E e )** | It inserts the specified element into the specified queue if it is possible to do so immediately without violating capacity restrictions, returning true upon success and throwing an IllegalStateException if no space is currently available. |
| **boolean contains( Object o )** | It returns true if this queue contains the specified element. |
| **int drainTo( Collection< ? super E > c )** | It removes all available elements from this queue and adds them to the given collection. |
| **int drainTo( Collection< ? super E > c, int maxElements )** | It removes at most the given number of available elements from this queue and adds them to the given collection. |
| **boolean offer( E e )** | It inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions, returning true upon success and false if no space is currently available. |
| **boolean offer( E e, long timeout, TimeUnit unit )** | It inserts the specified element into this queue, waiting up to the specified wait time if necessary, for space to become available. |
| **E poll( long timeout, TimeUnit unit )** | It retrieves and removes the head of this queue, waiting up to the specified wait time if necessary, for an element to become available. |
| **void put( E e )** | It inserts the specified element into this queue, waiting if necessary, for space to become available. |
| **int remainingCapacity( )** | It returns the number of additional elements that this queue can ideally (in the absence of memory or resource constraints) accept without blocking, or Integer.MAX\_VALUE if there is no intrinsic limit. |
| **boolean remove( Object o )** | It removes a single instance of the specified element from this queue, if it is present. |
| **E take( )** | It retrieves and removes the head of this queue, waiting if necessary, until an element becomes available. |

**The Behavior of BlockingQueue Methods :**

BlockingQueue provides certain methods for insertion, removal, and examine operations on the Blocking queue. Each of the four sets of methods behaves differently if the requested operation is not satisfied immediately.

1. **Throws Exception :** If the requested operation is not satisfied immediately, an exception will be thrown.
2. **Special value :** A special value is returned if the operation is not satisfied immediately.
3. **Blocks :** The method call is blocked if the attempted operation is not satisfied immediately and it waits until it gets executed.
4. **Times out :** The special value is returned to determine whether the operation was successful or not. If the requested operation does not happen immediately, the method calls for blocks until it occurs, but does not wait for more than the specified time.