Java Queue Interface

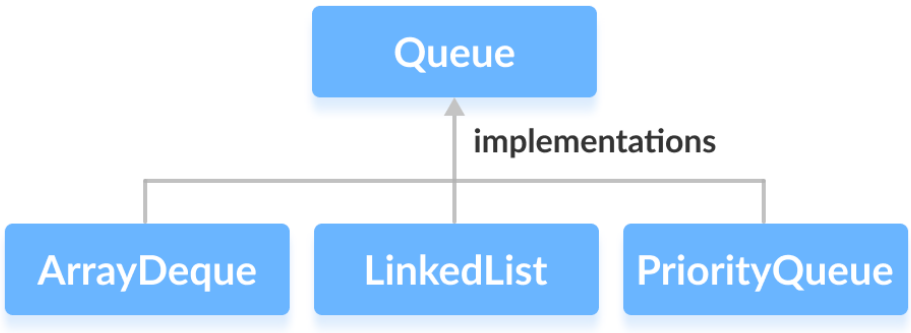
The Queue interface of the Java collections framework provides the functionality of the queue data structure. It extends the Collection interface.

Classes that Implement Queue

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

In order to use the functionalities of Queue, we need to use classes that implement it:

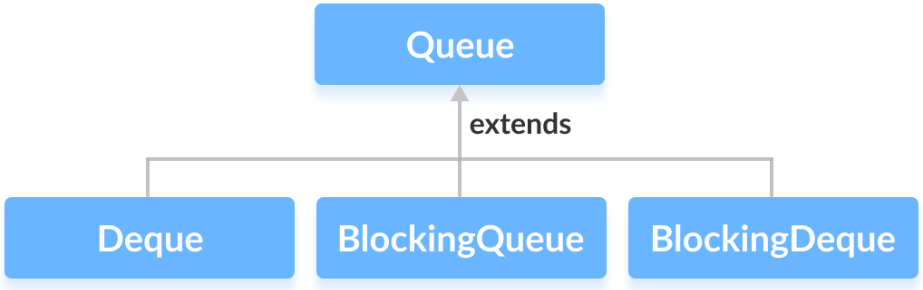
- ArrayDeque
- LinkedList
- PriorityQueue



Interfaces that extend Queue

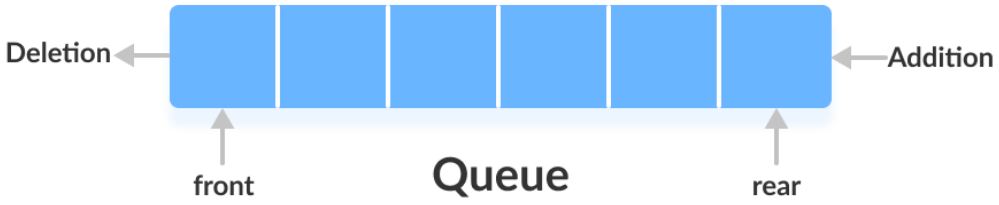
The Queue interface is also extended by various subinterfaces:

- Deque
- BlockingQueue
- BlockingDeque



Working of Queue Data Structure

In queues, elements are stored and accessed in **First In, First Out** manner. That is, elements are **added from the behind** and **removed from the front**.



How to use Queue?

In Java, we must import java.util.Queue package in order to use Queue.

```
// LinkedList implementation of Queue
Queue<String> animal1 = new LinkedList<>();

// Array implementation of Queue
Queue<String> animal2 = new ArrayDeque<>();

// Priority Queue implementation of Queue
Queue<String> animal 3 = new PriorityQueue<>();
```

Here, we have created objects *animal1*, *animal2* and *animal3* of classes LinkedList, ArrayDeque and PriorityQueue respectively. These objects can use the functionalities of the Queue interface.

Methods of Queue

The Queue interface includes all the methods of the Collection interface. It is because Collection is the super interface of Queue.

Some of the commonly used methods of the Queue interface are:

- **add()** - Inserts the specified element into the queue. If the task is successful, `add()` returns `true`, if not it throws an exception.
- **offer()** - Inserts the specified element into the queue. If the task is successful, `offer()` returns `true`, if not it returns `false`.
- **element()** - Returns the head of the queue. Throws an exception if the queue is empty.
- **peek()** - Returns the head of the queue. Returns `null` if the queue is empty.
- **remove()** - Returns and removes the head of the queue. Throws an exception if the queue is empty.
- **poll()** - Returns and removes the head of the queue. Returns `null` if the queue is empty.

1. Implementing the LinkedList Class

```
import java.util.Queue;
import java.util.LinkedList;

class Main {

    public static void main(String[] args) {
        // Creating Queue using the LinkedList class
        Queue<Integer> numbers = new LinkedList<>();

        // offer elements to the Queue
        numbers.offer(1);
        numbers.offer(2);
        numbers.offer(3);
        System.out.println("Queue: " + numbers);

        // Access elements of the Queue
        int accessedNumber = numbers.peek();
        System.out.println("Accessed Element: " + accessedNumber);

        // Remove elements from the Queue
```

```
        int removedNumber = numbers.poll();
        System.out.println("Removed Element: " + removedNumber);

        System.out.println("Updated Queue: " + numbers);
    }
}
```

Output

```
Queue: [1, 2, 3]
Accessed Element: 1
Removed Element: 1
Updated Queue: [2, 3]
```

2. Implementing the PriorityQueue Class

```
import java.util.Queue;
import java.util.PriorityQueue;

class Main {

    public static void main(String[] args) {
        // Creating Queue using the PriorityQueue class
        Queue<Integer> numbers = new PriorityQueue<>();

        // offer elements to the Queue
        numbers.offer(5);
        numbers.offer(1);
        numbers.offer(2);
        System.out.println("Queue: " + numbers);

        // Access elements of the Queue
        int accessedNumber = numbers.peek();
        System.out.println("Accessed Element: " + accessedNumber);

        // Remove elements from the Queue
        int removedNumber = numbers.poll();
        System.out.println("Removed Element: " + removedNumber);

        System.out.println("Updated Queue: " + numbers);
    }
}
```

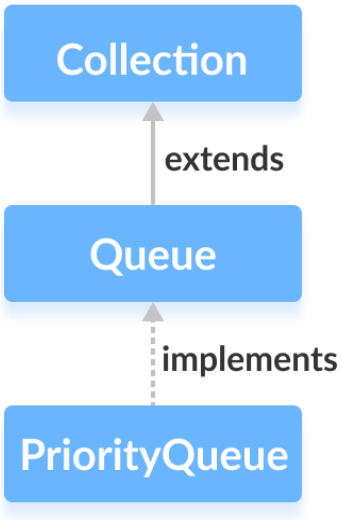
Output

```
Queue: [1, 5, 2]
Accessed Element: 1
Removed Element: 1
Updated Queue: [2, 5]
```

Java PriorityQueue

The PriorityQueue class provides the functionality of the heap data structure.

It implements the Queue interface.



Unlike normal queues, priority queue elements are retrieved in sorted order.

Suppose, we want to retrieve elements in the ascending order. In this case, the head of the priority queue will be the smallest element. Once this element is retrieved, the next smallest element will be the head of the queue.

It is important to note that the elements of a priority queue may not be sorted. However, elements are always retrieved in sorted order.

Creating PriorityQueue

In order to create a priority queue, we must import the java.util.PriorityQueue package. Once we import the package, here is how we can create a priority queue in Java.

```
PriorityQueue<Integer> numbers = new PriorityQueue<>();
```

Here, we have created a priority queue without any arguments. In this case, the head of the priority queue is the smallest element of the queue. And elements are removed in ascending order from the queue.

However, we can customize the ordering of elements with the help of the Comparator interface.

Methods of PriorityQueue

The PriorityQueue class provides the implementation of all the methods present in the Queue interface.

Insert Elements to PriorityQueue

- `add()` - Inserts the specified element to the queue. If the queue is full, it throws an exception.
- `offer()` - Inserts the specified element to the queue. If the queue is full, it returns `false`.

For example,

```
import java.util.PriorityQueue;

class Main {
    public static void main(String[] args) {

        // Creating a priority queue
        PriorityQueue<Integer> numbers = new PriorityQueue<>();

        // Using the add() method
        numbers.add(4);
        numbers.add(2);
        System.out.println("PriorityQueue: " + numbers);

        // Using the offer() method
        numbers.offer(1);
        System.out.println("Updated PriorityQueue: " + numbers);
    }
}
```

Output

```
PriorityQueue: [2, 4]
Updated PriorityQueue: [1, 4, 2]
```

Here, we have created a priority queue named *numbers*. We have inserted 4 and 2 to the queue.

Although 4 is inserted before 2, the head of the queue is 2. It is because the head of the priority queue is the smallest element of the queue.

We have then inserted 1 to the queue. The queue is now rearranged to store the smallest element 1 to the head of the queue.

Access PriorityQueue Elements

To access elements from a priority queue, we can use the peek() method. This method returns the head of the queue. For example,

```
import java.util.PriorityQueue;

class Main {
    public static void main(String[] args) {

        // Creating a priority queue
        PriorityQueue<Integer> numbers = new PriorityQueue<>();
        numbers.add(4);
        numbers.add(2);
        numbers.add(1);
        System.out.println("PriorityQueue: " + numbers);

        // Using the peek() method
        int number = numbers.peek();
        System.out.println("Accessed Element: " + number);
    }
}
```

Output

```
PriorityQueue: [1, 4, 2]
Accessed Element: 1
```

Remove PriorityQueue Elements

- `remove()` - removes the specified element from the queue
- `poll()` - returns and removes the head of the queue

For example,

```
import java.util.PriorityQueue;
```

```
class Main {
    public static void main(String[] args) {

        // Creating a priority queue
        PriorityQueue<Integer> numbers = new PriorityQueue<>();
        numbers.add(4);
        numbers.add(2);
        numbers.add(1);
        System.out.println("PriorityQueue: " + numbers);

        // Using the remove() method
        boolean result = numbers.remove(2);
        System.out.println("Is the element 2 removed? " + result);

        // Using the poll() method
        int number = numbers.poll();
        System.out.println("Removed Element Using poll(): " + number);
    }
}
```

Output

```
PriorityQueue: [1, 4, 2]
Is the element 2 removed? true
Removed Element Using poll(): 1
```

Iterating Over a PriorityQueue

To iterate over the elements of a priority queue, we can use the `iterator()` method. In order to use this method, we must import the `java.util.Iterator` package. For example,

```
import java.util.PriorityQueue;
import java.util.Iterator;

class Main {
    public static void main(String[] args) {

        // Creating a priority queue
        PriorityQueue<Integer> numbers = new PriorityQueue<>();
        numbers.add(4);
        numbers.add(2);
        numbers.add(1);
        System.out.print("PriorityQueue using iterator(): ");

        //Using the iterator() method
        Iterator<Integer> iterate = numbers.iterator();
        while(iterate.hasNext()) {
```

```
            System.out.print(iterate.next());
            System.out.print(", ");
        }
    }
}
```

Output

```
PriorityQueue using iterator(): 1, 4, 2,
```

Other PriorityQueue Methods

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

PriorityQueue Comparator

In all the examples above, priority queue elements are retrieved in the natural order (ascending order). However, we can customize this ordering.

For this, we need to create our own comparator class that implements the `Comparator` interface. For example,

```
import java.util.PriorityQueue;
import java.util.Comparator;
class Main {
    public static void main(String[] args) {

        // Creating a priority queue
        PriorityQueue<Integer> numbers = new PriorityQueue<>(new
CustomComparator());
        numbers.add(4);
        numbers.add(2);
        numbers.add(1);
        numbers.add(3);
        System.out.print("PriorityQueue: " + numbers);
    }
}

class CustomComparator implements Comparator<Integer> {
```

```
    @Override
    public int compare(Integer number1, Integer number2) {
        int value =  number1.compareTo(number2);
        // elements are sorted in reverse order
        if (value > 0) {
            return -1;
        }
        else if (value < 0) {
            return 1;
        }
        else {
            return 0;
        }
    }
}
```

Output

```
PriorityQueue: [4, 3, 1, 2]
```

In the above example, we have created a priority queue passing *CustomComparator* class as an argument.

The *CustomComparator* class implements the `Comparator` interface.

We then override the `compare()` method. The method now causes the head of the element to be the largest number.

Java Deque Interface

The Deque interface of the Java collections framework provides the functionality of a double-ended queue. It extends the Queue interface.

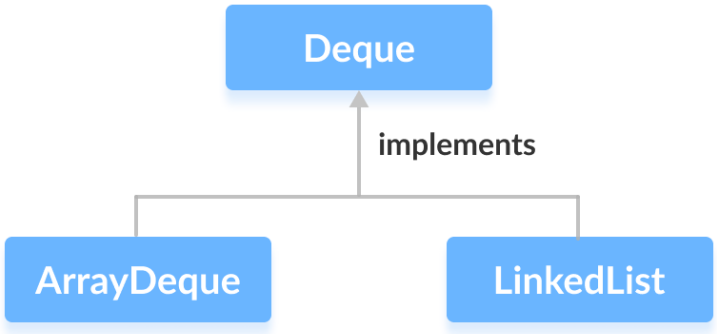
Working of Deque

In a regular queue, elements are added from the rear and removed from the front. However, in a deque, we can **insert and remove elements from both front and rear**.

Classes that implement Deque

In order to use the functionalities of the Deque interface, we need to use classes that implement it:

- ArrayDeque
- LinkedList



How to use Deque?

In Java, we must import the `java.util.Deque` package to use `Deque`.

```
// Array implementation of Deque
Deque<String> animal1 = new ArrayDeque<>();

// LinkedList implementation of Deque
Deque<String> animal2 = new LinkedList<>();
```

Here, we have created objects *animal1* and *animal2* of classes *ArrayDeque* and *LinkedList*, respectively. These objects can use the functionalities of the `Deque` interface.

Methods of Deque

Since Deque extends the Queue interface, it inherits all the methods of the Queue interface.

Besides methods available in the Queue interface, the Deque interface also includes the following methods:

- **addFirst()** - Adds the specified element at the beginning of the deque. Throws an exception if the deque is full.
- **addLast()** - Adds the specified element at the end of the deque. Throws an exception if the deque is full.
- **offerFirst()** - Adds the specified element at the beginning of the deque. Returns `false` if the deque is full.
- **offerLast()** - Adds the specified element at the end of the deque. Returns `false` if the deque is full.
- **getFirst()** - Returns the first element of the deque. Throws an exception if the deque is empty.
- **getLast()** - Returns the last element of the deque. Throws an exception if the deque is empty.
- **peekFirst()** - Returns the first element of the deque. Returns `null` if the deque is empty.
- **peekLast()** - Returns the last element of the deque. Returns `null` if the deque is empty.
- **removeFirst()** - Returns and removes the first element of the deque. Throws an exception if the deque is empty.
- **removeLast()** - Returns and removes the last element of the deque. Throws an exception if the deque is empty.
- **pollFirst()** - Returns and removes the first element of the deque. Returns `null` if the deque is empty.
- **pollLast()** - Returns and removes the last element of the deque. Returns `null` if the deque is empty.

Deque as Stack Data Structure

The Stack class of the Java Collections framework provides the implementation of the stack.

However, it is recommended to use Deque as a stack instead of the Stack class. It is because methods of Stack are synchronized.

Here are the methods the Deque interface provides to implement stack:

- `push()` - adds an element at the beginning of deque
- `pop()` - removes an element from the beginning of deque
- `peek()` - returns an element from the beginning of deque

Implementation of Deque in ArrayDeque Class

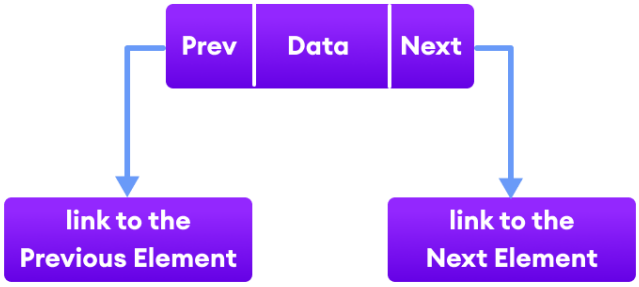
```
import java.util.Deque;
import java.util.ArrayDeque;

class Main {

    public static void main(String[] args) {
        // Creating Deque using the ArrayDeque class
        Deque<Integer> numbers = new ArrayDeque<>();

        // add elements to the Deque
        numbers.offer(1);
        numbers.offerLast(2);
        numbers.offerFirst(3);
        System.out.println("Deque: " + numbers);

        // Access elements of the Deque
        int firstElement = numbers.peekFirst();
        System.out.println("First Element: " + firstElement);

        int lastElement = numbers.peekLast();
        System.out.println("Last Element: " + lastElement);

        // Remove elements from the Deque
        int removedNumber1 = numbers.pollFirst();
        System.out.println("Removed First Element: " +
removedNumber1);

        int removedNumber2 = numbers.pollLast();
        System.out.println("Removed Last Element: " + removedNumber2);

        System.out.println("Updated Deque: " + numbers);
    }
}
```

Output

```
Deque: [3, 1, 2]
First Element: 3
Last Element: 2
Removed First Element: 3
Removed Last Element: 2
Updated Deque: [1]
```

Java LinkedList

The LinkedList class of the Java collections framework provides the functionality of the linked list data structure (doubly linkedlist).



Java Doubly LinkedList

Each element in a linked list is known as a node. It consists of 3 fields:

- **Prev** - stores an address of the previous element in the list. It is null for the first element
- **Next** - stores an address of the next element in the list. It is null for the last element
- **Data** - stores the actual data

Creating a Java LinkedList

Here is how we can create linked lists in Java:

```
LinkedList<Type> linkedList = new LinkedList<>();
```

Here, *Type* indicates the type of a linked list. For example,

```
// create Integer type linked list
LinkedList<Integer> linkedList = new LinkedList<>();

// create String type linked list
LinkedList<String> linkedList = new LinkedList<>();
```

Example: Create LinkedList in Java

```
import java.util.LinkedList;

class Main {
  public static void main(String[] args){
```

```
    // create linkedlist
    LinkedList<String> animals = new LinkedList<>();

    // Add elements to LinkedList
    animals.add("Dog");
    animals.add("Cat");
    animals.add("Cow");
    System.out.println("LinkedList: " + animals);
  }
}
```

Output

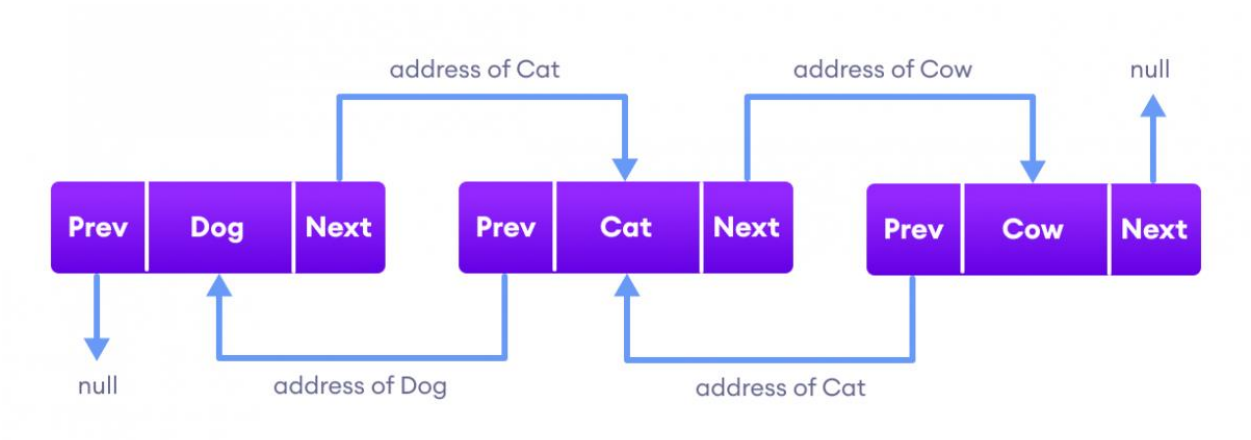
```
LinkedList: [Dog, Cat, Cow]
```

In the above example, we have created a LinkedList named *animals*.

Here, we have used the add() method to add elements to the LinkedList. We will learn more about the add() method later in this tutorial.

Working of a Java LinkedList

Elements in linked lists are not stored in sequence. Instead, they are scattered and connected through links (**Prev** and **Next**).



Java LinkedList Implementation

Here we have 3 elements in a linked list.

- *Dog* - it is the first element that holds *null* as previous address and the address of *Cat* as the next address
- *Cat* - it is the second element that holds an address of *Dog* as the previous address and the address of *Cow* as the next address

- *Cow* - it is the last element that holds the address of *Cat* as the previous address and *null* as the next element

Methods of Java LinkedList

LinkedList provides various methods that allow us to perform different operations in linked lists. We will look at four commonly used LinkedList Operators in this tutorial:

- Add elements
- Access elements
- Change elements
- Remove elements

1. Add elements to a LinkedList

We can use the add() method to add an element (node) at the end of the LinkedList. For example,

```
import java.util.LinkedList;

class Main {
  public static void main(String[] args){
    // create linkedlist
    LinkedList<String> animals = new LinkedList<>();

    // add() method without the index parameter
    animals.add("Dog");
    animals.add("Cat");
    animals.add("Cow");
    System.out.println("LinkedList: " + animals);

    // add() method with the index parameter
    animals.add(1, "Horse");
    System.out.println("Updated LinkedList: " + animals);
  }
}
```

Output

```
LinkedList: [Dog, Cat, Cow]
Updated LinkedList: [Dog, Horse, Cat, Cow]
```

In the above example, we have created a LinkedList named *animals*. Here, we have used the add() method to add elements to *animals*.

Notice the statement,

```
animals.add(1, "Horse");
```

Here, we have used the **index number** parameter. It is an optional parameter that specifies the position where the new element is added.

2. Access LinkedList elements

The get() method of the LinkedList class is used to access an element from the LinkedList. For example,

```
import java.util.LinkedList;

class Main {
  public static void main(String[] args) {
    LinkedList<String> languages = new LinkedList<>();

    // add elements in the linked list
    languages.add("Python");
    languages.add("Java");
    languages.add("JavaScript");
    System.out.println("LinkedList: " + languages);

    // get the element from the linked list
    String str = languages.get(1);
    System.out.print("Element at index 1: " + str);
  }
}
```

Output

```
LinkedList: [Python, Java, JavaScript]
Element at index 1: Java
```

In the above example, we have used the get() method with parameter **1**. Here, the method returns the element at index **1**.

We can also access elements of the LinkedList using the iterator() and the listIterator() method.

3. Change Elements of a LinkedList

The `set()` method of `LinkedList` class is used to change elements of the LinkedList. For example,

```
import java.util.LinkedList;

class Main {
  public static void main(String[] args) {
    LinkedList<String> languages = new LinkedList<>();

    // add elements in the linked list
    languages.add("Java");
    languages.add("Python");
    languages.add("JavaScript");
    languages.add("Java");
    System.out.println("LinkedList: " + languages);

    // change elements at index 3
    languages.set(3, "Kotlin");
    System.out.println("Updated LinkedList: " + languages);
  }
}
```

Output

```
LinkedList: [Java, Python, JavaScript, Java]
Updated LinkedList: [Java, Python, JavaScript, Kotlin]
```

In the above example, we have created a LinkedList named languages. Notice the line,

```
languages.set(3, "Kotlin");
```

Here, the `set()` method changes the element at index **3** to *Kotlin*.

4. Remove element from a LinkedList

The remove() method of the LinkedList class is used to remove an element from the LinkedList. For example,

```
import java.util.LinkedList;

class Main {
  public static void main(String[] args) {
    LinkedList<String> languages = new LinkedList<>();

    // add elements in LinkedList
    languages.add("Java");
    languages.add("Python");
    languages.add("JavaScript");
```

```
    languages.add("Kotlin");
    System.out.println("LinkedList: " + languages);

    // remove elements from index 1
    String str = languages.remove(1);
    System.out.println("Removed Element: " + str);

    System.out.println("Updated LinkedList: " + languages);
  }
}
```

Output

```
LinkedList: [Java, Python, JavaScript, Kotlin]
Removed Element: Python
New LinkedList: [Java, JavaScript, Kotlin]
```

Here, the remove() method takes the index number as the parameter. And, removes the element specified by the index number.

Other Methods

Methods	Description
`contains()`	checks if the LinkedList contains the element
`indexOf()`	returns the index of the first occurrence of the element
`lastIndexOf()`	returns the index of the last occurrence of the element
`clear()`	removes all the elements of the LinkedList
`iterator()`	returns an iterator to iterate over LinkedList

LinkedList as Deque and Queue

Since the LinkedList class also implements the Queue and the Deque interface, it can implement methods of these interfaces as well. Here are some of the commonly used methods:

Methods	Descriptions
`addFirst()`	adds the specified element at the beginning of the linked list
`addLast()`	adds the specified element at the end of the linked list
`getFirst()`	returns the first element
`getLast()`	returns the last element
`removeFirst()`	removes the first element
`removeLast()`	removes the last element
`peek()`	returns the first element (head) of the linked list
`poll()`	returns and removes the first element from the linked list
`offer()`	adds the specified element at the end of the linked list

Example: Java LinkedList as Queue

```
import java.util.LinkedList;
import java.util.Queue;

class Main {
  public static void main(String[] args) {
    Queue<String> languages = new LinkedList<>();

    // add elements
    languages.add("Python");
    languages.add("Java");
    languages.add("C");
    System.out.println("LinkedList: " + languages);

    // access the first element
    String str1 = languages.peek();
    System.out.println("Accessed Element: " + str1);

    // access and remove the first element
    String str2 = languages.poll();
    System.out.println("Removed Element: " + str2);
    System.out.println("LinkedList after poll(): " + languages);

    // add element at the end
    languages.offer("Swift");
    System.out.println("LinkedList after offer(): " + languages);
  }
```

```
}
```

Output

```
LinkedList: [Python, Java, C]
Accessed Element: Python
Removed Element: Python
LinkedList after poll(): [Java, C]
LinkedList after offer(): [Java, C, Swift]
```

Example: LinkedList as Deque

```
import java.util.LinkedList;
import java.util.Deque;

class Main {
  public static void main(String[] args){
    Deque<String> animals = new LinkedList<>();

    // add element at the beginning
    animals.add("Cow");
    System.out.println("LinkedList: " + animals);

    animals.addFirst("Dog");
    System.out.println("LinkedList after addFirst(): " + animals);

    // add elements at the end
    animals.addLast("Zebra");
    System.out.println("LinkedList after addLast(): " + animals);

    // remove the first element
    animals.removeFirst();
    System.out.println("LinkedList after removeFirst(): " + animals);

    // remove the last element
    animals.removeLast();
    System.out.println("LinkedList after removeLast(): " + animals);
  }
}
```

Output

```
LinkedList: [Cow]
LinkedList after addFirst(): [Dog, Cow]
LinkedList after addLast(): [Dog, Cow, Zebra]
LinkedList after removeFirst(): [Cow, Zebra]
LinkedList after removeLast(): [Cow]
```

Iterating through LinkedList

We can use the Java for-each loop to iterate through LinkedList. For example,

```
import java.util.LinkedList;

class Main {
    public static void main(String[] args) {
        // Creating a linked list
        LinkedList<String> animals = new LinkedList<>();
        animals.add("Cow");
        animals.add("Cat");
        animals.add("Dog");
        System.out.println("LinkedList: " + animals);

        // Using forEach loop
        System.out.println("Accessing linked list elements:");
        for(String animal: animals) {
            System.out.print(animal);
            System.out.print(", ");
        }
    }
}
```

Output

```
LinkedList: [Cow, Cat, Dog]
Accessing linked list elements:
Cow, Cat, Dog,
```

LinkedList Vs. ArrayList

Both the Java ArrayList and LinkedList implements the List interface of the Collections framework. However, there exists some difference between them.

LinkedList	ArrayList
Implements `List`, `Queue`, and `Deque` interfaces.	Implements `List` interface.
Stores 3 values (**previous address**, **data,** and **next address**) in a single position.	Stores a single value in a single position.
Provides the doubly-linked list implementation.	Provides a resizable array implementation.

Whenever an element is added, `prev` and `next` address are changed.	Whenever an element is added, all elements after that position are shifted.
To access an element, we need to iterate from the beginning to the element.	Can randomly access elements using indexes.

Note: We can also create a LinkedList using interfaces in Java. For example,

```
// create linkedlist using List
List<String> animals1 = new LinkedList<>();

// creating linkedlist using Queue
Queue<String> animals2 = new LinkedList<>();

// creating linkedlist using Deque
Deque<String> animals3 = new LinkedList<>();
```

Here, if the LinkedList is created using one interface, then we cannot use methods provided by other interfaces. That is, *animals1* cannot use methods specific to Queue and Deque interfaces.

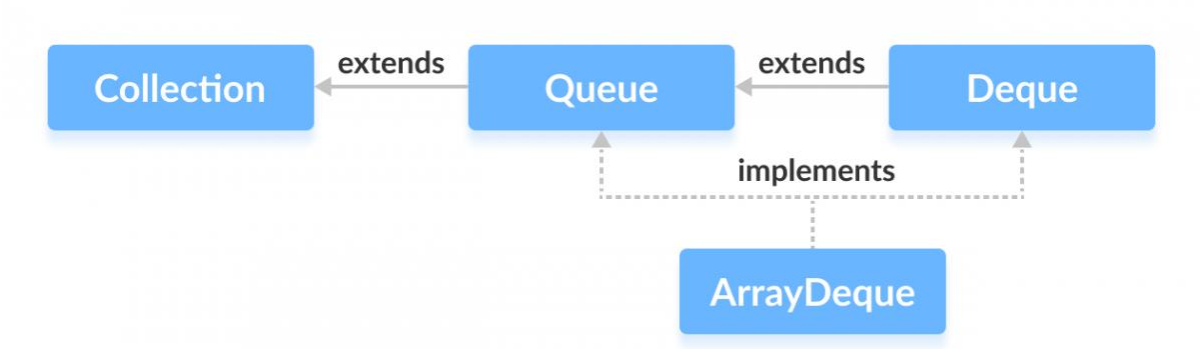
Java ArrayDeque

In Java, we can use the ArrayDeque class to implement queue and deque data structures using arrays.

Interfaces implemented by ArrayDeque

The ArrayDeque class implements these two interfaces:

- Java Queue Interface
- Java Deque Interface



Creating ArrayDeque

In order to create an array deque, we must import the `java.util.ArrayDeque` package.

Here is how we can create an array deque in Java:

```
ArrayDeque<Type> animal = new ArrayDeque<>();
```

Here, *Type* indicates the type of the array deque. For example,

```
// Creating String type ArrayDeque
ArrayDeque<String> animals = new ArrayDeque<>();

// Creating Integer type ArrayDeque
ArrayDeque<Integer> age = new ArrayDeque<>();
```

Methods of ArrayDeque

The `ArrayDeque` class provides implementations for all the methods present in `Queue` and `Deque` interface.

Insert Elements to Deque

1. Add elements using add(), addFirst() and addLast()

- **`add()`** - inserts the specified element at the end of the array deque
- **`addFirst()`** - inserts the specified element at the beginning of the array deque
- **`addLast()`** - inserts the specified at the end of the array deque (equivalent to `add()`)

Note: If the array deque is full, all these methods add(), addFirst() and addLast() throws IllegalStateException.

For example,

```
import java.util.ArrayDeque;

class Main {
    public static void main(String[] args) {
        ArrayDeque<String> animals= new ArrayDeque<>();

        // Using add()
        animals.add("Dog");

        // Using addFirst()
        animals.addFirst("Cat");

        // Using addLast()
        animals.addLast("Horse");
        System.out.println("ArrayDeque: " + animals);
    }
}
```

Output

```
ArrayDeque: [Cat, Dog, Horse]
```

2. Insert elements using offer(), offerFirst() and offerLast()

- **`offer()`** - inserts the specified element at the end of the array deque
- **`offerFirst()`** - inserts the specified element at the beginning of the array deque
- **`offerLast()`** - inserts the specified element at the end of the array deque

Note: `offer()`, `offerFirst()` and `offerLast()` returns `true` if the element is successfully inserted; if the array deque is full, these methods return `false`.

For example,

```
import java.util.ArrayDeque;

class Main {
    public static void main(String[] args) {
        ArrayDeque<String> animals= new ArrayDeque<>();
        // Using offer()
        animals.offer("Dog");
```

```
        // Using offerFirst()
        animals.offerFirst("Cat");

        // Using offerLast()
        animals.offerLast("Horse");
        System.out.println("ArrayDeque: " + animals);
    }
}
```

Output

```
ArrayDeque: [Cat, Dog, Horse]
```

Note: If the array deque is full

- the `add()` method will throw an exception
- the `offer()` method returns `false`

Access ArrayDeque Elements

1. Access elements using getFirst() and getLast()

- **getFirst()** - returns the first element of the array deque
- **getLast()** - returns the last element of the array deque

Note: If the array deque is empty, getFirst() and getLast() throws NoSuchElementException.

For example,

```
import java.util.ArrayDeque;

class Main {
    public static void main(String[] args) {
        ArrayDeque<String> animals= new ArrayDeque<>();
        animals.add("Dog");
        animals.add("Cat");
        animals.add("Horse");
        System.out.println("ArrayDeque: " + animals);

        // Get the first element
        String firstElement = animals.getFirst();
        System.out.println("First Element: " + firstElement);

        // Get the last element
        String lastElement = animals.getLast();
        System.out.println("Last Element: " + lastElement);
```

```
    }
}
```

Output

```
ArrayDeque: [Dog, Cat, Horse]
First Element: Dog
Last Element: Horse
```

2. Access elements using peek(), peekFirst() and peekLast() method

- **`peek()`** - returns the first element of the array deque
- **`peekFirst()`** - returns the first element of the array deque (equivalent to `peek()`)
- **`peekLast()`** - returns the last element of the array deque

For example,

```
import java.util.ArrayDeque;

class Main {
    public static void main(String[] args) {
        ArrayDeque<String> animals= new ArrayDeque<>();
        animals.add("Dog");
        animals.add("Cat");
        animals.add("Horse");
        System.out.println("ArrayDeque: " + animals);

        // Using peek()
        String element = animals.peek();
        System.out.println("Head Element: " + element);

        // Using peekFirst()
        String firstElement = animals.peekFirst();
        System.out.println("First Element: " + firstElement);

        // Using peekLast
        String lastElement = animals.peekLast();
        System.out.println("Last Element: " + lastElement);
    }
}
```

Output

```
ArrayDeque: [Dog, Cat, Horse]
Head Element: Dog
First Element: Dog
Last Element: Horse
```

Note: If the array deque is empty, `peek()`, `peekFirst()` and `getLast()` throws `NoSuchElementException`.

Remove ArrayDeque Elements

1. Remove elements using the remove(), removeFirst(), removeLast() method

- **`remove()`** - returns and removes an element from the first element of the array deque
- `remove(element)` - returns and removes the specified element from the head of the array deque
- **`removeFirst()`** - returns and removes the first element from the array deque (equivalent to `remove()`)
- **`removeLast()`** - returns and removes the last element from the array deque

Note: If the array deque is empty, remove(), removeFirst() and removeLast() method throws an exception. Also, remove(element) throws an exception if the element is not found.

For example,

```
import java.util.ArrayDeque;

class Main {
    public static void main(String[] args) {
        ArrayDeque<String> animals= new ArrayDeque<>();
        animals.add("Dog");
        animals.add("Cat");
        animals.add("Cow");
        animals.add("Horse");
        System.out.println("ArrayDeque: " + animals);

        // Using remove()
        String element = animals.remove();
        System.out.println("Removed Element: " + element);

        System.out.println("New ArrayDeque: " + animals);

        // Using removeFirst()
        String firstElement = animals.removeFirst();
        System.out.println("Removed First Element: " + firstElement);

        // Using removeLast()
        String lastElement = animals.removeLast();
        System.out.println("Removed Last Element: " + lastElement);
    }
}
```

Output

```
ArrayDeque: [Dog, Cat, Cow, Horse]
Removed Element: Dog
New ArrayDeque: [Cat, Cow, Horse]
Removed First Element: Cat
Removed Last Element: Horse
```

2. Remove elements using the poll(), pollFirst() and pollLast() method

- **`poll()`** - returns and removes the first element of the array deque
- **`pollFirst()`** - returns and removes the first element of the array deque (equivalent to `poll()`)
- **`pollLast()`** - returns and removes the last element of the array deque

Note: If the array deque is empty, `poll()`, `pollFirst()` and `pollLast()` returns `null` if the element is not found.

For example,

```
import java.util.ArrayDeque;

class Main {
    public static void main(String[] args) {
        ArrayDeque<String> animals= new ArrayDeque<>();
        animals.add("Dog");
        animals.add("Cat");
        animals.add("Cow");
        animals.add("Horse");
        System.out.println("ArrayDeque: " + animals);

        // Using poll()
        String element = animals.poll();
        System.out.println("Removed Element: " + element);
        System.out.println("New ArrayDeque: " + animals);

        // Using pollFirst()
        String firstElement = animals.pollFirst();
        System.out.println("Removed First Element: " + firstElement);

        // Using pollLast()
        String lastElement = animals.pollLast();
        System.out.println("Removed Last Element: " + lastElement);
    }
}
```

Output

```
ArrayDeque: [Dog, Cat, Cow, Horse]
Removed Element: Dog
New ArrayDeque: [Cat, Cow, Horse]
Removed First Element: Cat
Removed Last Element: Horse
```

3. Remove Element: using the clear() method

To remove all the elements from the array deque, we use the `clear()` method. For example,

```
import java.util.ArrayDeque;

class Main {
    public static void main(String[] args) {
        ArrayDeque<String> animals= new ArrayDeque<>();
        animals.add("Dog");
        animals.add("Cat");
        animals.add("Horse");
        System.out.println("ArrayDeque: " + animals);

        // Using clear()
        animals.clear();

        System.out.println("New ArrayDeque: " + animals);
    }
}
```

Output

```
ArrayDeque: [Dog, Cat, Horse]
New ArrayDeque: []
```

Iterating the ArrayDeque

- **`iterator()`** - returns an iterator that can be used to iterate over the array deque
- **`descendingIterator()`** - returns an iterator that can be used to iterate over the array deque in reverse order

In order to use these methods, we must import the java.util.Iterator package. For example,

```
import java.util.ArrayDeque;
import java.util.Iterator;
```

```
class Main {
    public static void main(String[] args) {
        ArrayDeque<String> animals= new ArrayDeque<>();
        animals.add("Dog");
        animals.add("Cat");
        animals.add("Horse");

        System.out.print("ArrayDeque: ");

        // Using iterator()
        Iterator<String> iterate = animals.iterator();
        while(iterate.hasNext()) {
            System.out.print(iterate.next());
            System.out.print(", ");
        }

        System.out.print("\nArrayDeque in reverse order: ");
        // Using descendingIterator()
        Iterator<String> desIterate = animals.descendingIterator();
        while(desIterate.hasNext()) {
            System.out.print(desIterate.next());
            System.out.print(", ");
        }
    }
}
```

Output

```
ArrayDeque: [Dog, Cat, Horse]
ArrayDeque in reverse order: [Horse, Cat, Dog]
```

Other Methods

Methods	Descriptions
`element()`	Returns an element from the head of the array deque.
`contains(element)`	Searches the array deque for the specified element. If the element is found, it returns `true`, if not it returns `false`.
`size()`	Returns the length of the array deque.
`toArray()`	Converts array deque to array and returns it.

`clone()`	Creates a copy of the array deque and returns it.

ArrayDeque as a Stack

To implement a **LIFO (Last-In-First-Out)** stacks in Java, it is recommended to use a deque over the Stack class. The ArrayDeque class is likely to be faster than the Stack class.

ArrayDeque provides the following methods that can be used for implementing a stack.

- **`push()`** - adds an element to the top of the stack
- **`peek()`** - returns an element from the top of the stack
- **`pop()`** - returns and removes an element from the top of the stack

For example,

```
import java.util.ArrayDeque;

class Main {
    public static void main(String[] args) {
        ArrayDeque<String> stack = new ArrayDeque<>();

        // Add elements to stack
        stack.push("Dog");
        stack.push("Cat");
        stack.push("Horse");
        System.out.println("Stack: " + stack);

        // Access element from top of stack
        String element = stack.peek();
        System.out.println("Accessed Element: " + element);

        // Remove elements from top of stack
        String remElement = stack.pop();
        System.out.println("Removed element: " + remElement);
    }
}
```

Output

```
Stack: [Horse, Cat, Dog]
Accessed Element: Horse
Removed Element: Horse
```

ArrayDeque Vs. LinkedList Class

Both ArrayDeque and Java LinkedList implements the Deque interface. However, there exist some differences between them.

- `LinkedList` supports `null` elements, whereas `ArrayDeque` doesn't.
- Each node in a linked list includes links to other nodes. That's why `LinkedList` requires more storage than `ArrayDeque`.
- If you are implementing the queue or the deque data structure, an `ArrayDeque` is likely to faster than a `LinkedList`.

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.

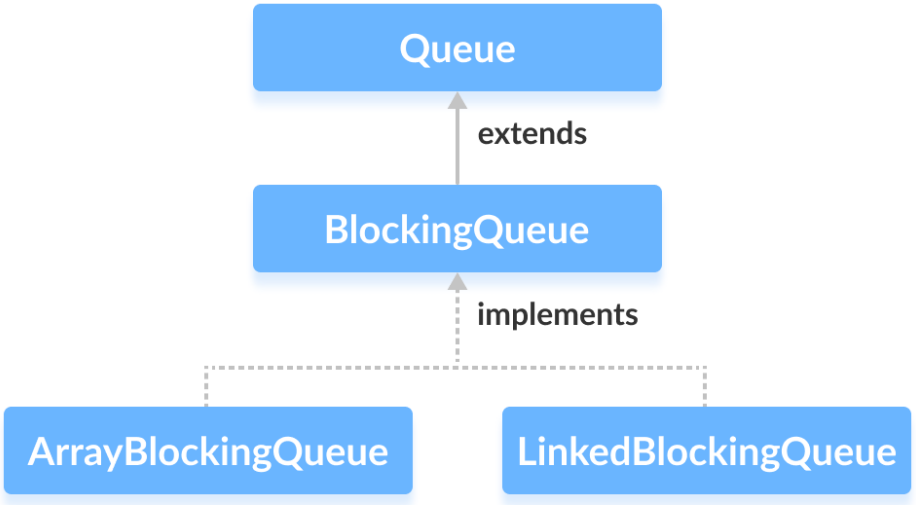
For example, if we want to delete an element from an empty queue, then the blocking queue allows the delete operation to wait until the queue contains some elements to be deleted.

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
- 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
```

Why BlockingQueue?

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

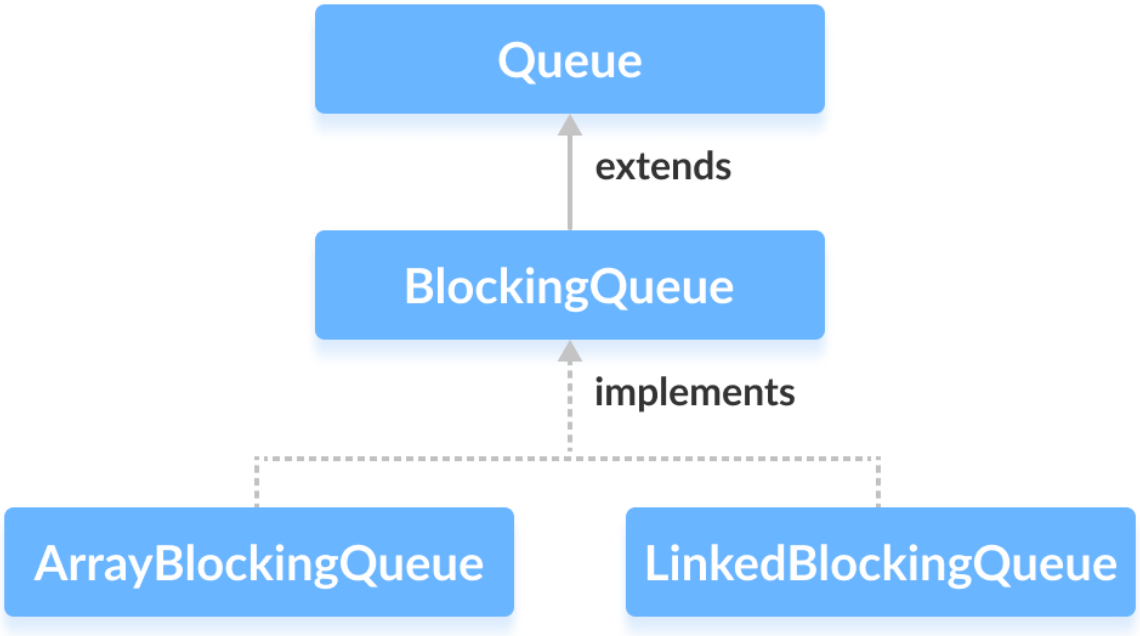
Suppose one thread is inserting elements to the queue and another thread is removing elements from the queue.

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

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.

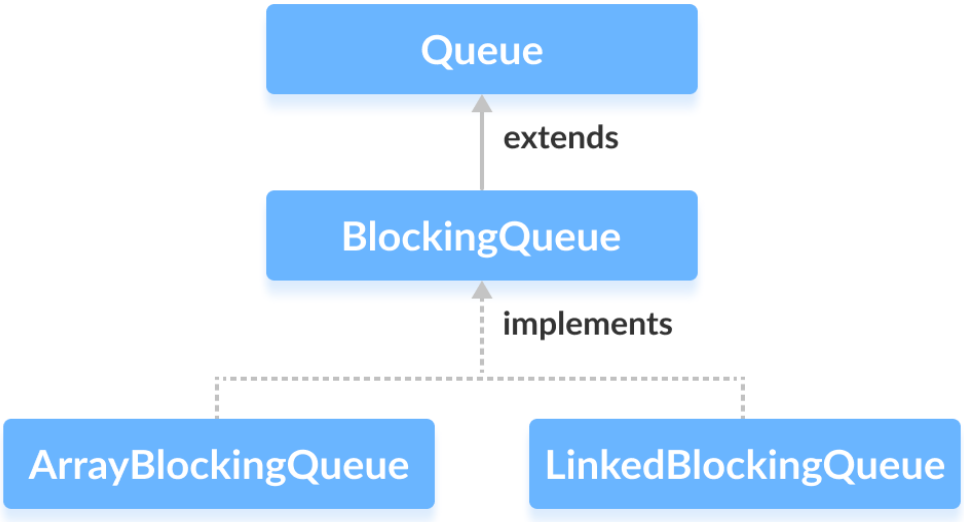
Suppose, one thread is inserting elements to the queue and another thread is removing elements from the queue.

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.

Java LinkedBlockingQueue

The LinkedBlockingQueue class of the Java Collections framework provides the blocking queue implementation using a linked list.

It implements the Java BlockingQueue interface.



Creating LinkedBlockingQueue

In order to create a linked blocking queue, we must import the `java.util.concurrent.LinkedBlockingQueue` package.

Here is how we can create a linked blocking queue in Java:

1. Without the initial capacity

```
LinkedBlockingQueue<Type> animal = new LinkedBlockingQueue<>();
```

Here the default initial capacity will be 2^{31}-1.

2. With the initial capacity

```
LinkedBlockingQueue<Type> animal = new LinkedBlockingQueue<>(int
capacity);
```

Here,

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

For example,

```
// Creating String type LinkedBlockingQueue with size 5
LinkedBlockingQueue<String> animals = new LinkedBlockingQueue<>(5);

// Creating Integer type LinkedBlockingQueue with size 5
LinkedBlockingQueue<Integer> age = new LinkedBlockingQueue<>(5);
```

Note: It is not compulsory to provide the size of the linked list.

Methods of LinkedBlockingQueue

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

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

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

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

Insert Elements

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

For example,

```
import java.util.concurrent.LinkedBlockingQueue;

class Main {
    public static void main(String[] args) {
```

```
        LinkedBlockingQueue<String> animals = new
LinkedBlockingQueue<>(5);

        // Using add()
        animals.add("Dog");
        animals.add("Cat");

        // Using offer()
        animals.offer("Horse");
        System.out.println("LinkedBlockingQueue: " + animals);
    }
}
```

Output

```
LinkedBlockingQueue: [Dog, Cat, Horse]
```

Access Elements

- **`peek()`** - Returns an element from the front of the linked blocking queue. It returns `null` if the queue is empty.
- **`iterator()`** - Returns an iterator object to sequentially access an element from the linked 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.LinkedBlockingQueue;
import java.util.Iterator;

class Main {
    public static void main(String[] args) {
        LinkedBlockingQueue<String> animals = new
LinkedBlockingQueue<>(5);

        // Add elements
        animals.add("Dog");
        animals.add("Cat");
        animals.add("Horse");
        System.out.println("LinkedBlockingQueue: " + animals);

        // Using peek()
        String element = animals.peek();
        System.out.println("Accessed Element: " + element);

        // Using iterator()
        Iterator<String> iterate = animals.iterator();
        System.out.print("LinkedBlockingQueue Elements: ");
```

```
        while(iterate.hasNext()) {
            System.out.print(iterate.next());
            System.out.print(", ");
        }
    }
}
```

Output

```
LinkedBlockingQueue: [Dog, Cat, Horse]
Accessed Element: Dog
LinkedBlockingQueue Elements: Dog, Cat, Horse,
```

Remove Elements

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

For example,

```
import java.util.concurrent.LinkedBlockingQueue;

class Main {
    public static void main(String[] args) {
        LinkedBlockingQueue<String> animals = new
LinkedBlockingQueue<>(5);

        animals.add("Dog");
        animals.add("Cat");
        animals.add("Horse");
        System.out.println("LinkedBlockingQueue " + 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 LinkedBlockingQueue " + animals);
    }
```

```
}
```

Output

```
LinkedBlockingQueue: [Dog, Cat, Horse]
Removed Elements:
Using remove(): Dog
Using poll(): Cat
Updated LinkedBlockingQueue: []
```

put() and take() Methods

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 insert the specified element to the end of a linked blocking queue, we use the put() method.

If the linked blocking queue is full, it waits until there is space in the linked blocking queue to insert the element.

For example,

```
import java.util.concurrent.LinkedBlockingQueue;

class Main {
    public static void main(String[] args) {
        LinkedBlockingQueue<String> animals = new
LinkedBlockingQueue<>(5);

       try {
        // Add elements to animals
           animals.put("Dog");
           animals.put("Cat");
           System.out.println("LinkedBlockingQueue: " + animals);
        }
        catch(Exception e) {
            System.out.println(e);
        }
    }
}
```

Output

```
LinkedBlockingQueue: [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 linked blocking queue, we can use the take() method.

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

For example,

```
import java.util.concurrent.LinkedBlockingQueue;

class Main {
    public static void main(String[] args) {
        LinkedBlockingQueue<String> animals = new
LinkedBlockingQueue<>(5);

       try {
           //Add elements to animals
           animals.put("Dog");
           animals.put("Cat");
           System.out.println("LinkedBlockingQueue: " + animals);

           // Remove an element
           String element = animals.take();
           System.out.println("Removed Element: " + element);
           System.out.println("New LinkedBlockingQueue: " + animals);
        }
        catch(Exception e) {
            System.out.println(e);
        }
    }
}
```

Output

```
LinkedBlockingQueue: [Dog, Cat]
Removed Element: Dog
New LinkedBlockingQueue: [Cat]
```

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 linked 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 linked blocking queue.
`toArray()`	Converts linked blocking queue to an array and return the array.
`toString()`	Converts the linked blocking queue to string

Why use LinkedBlockingQueue?

The LinkedBlockingQueue uses linked lists 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 linked blocking queue can make the second thread waits until the first thread completes its operations.