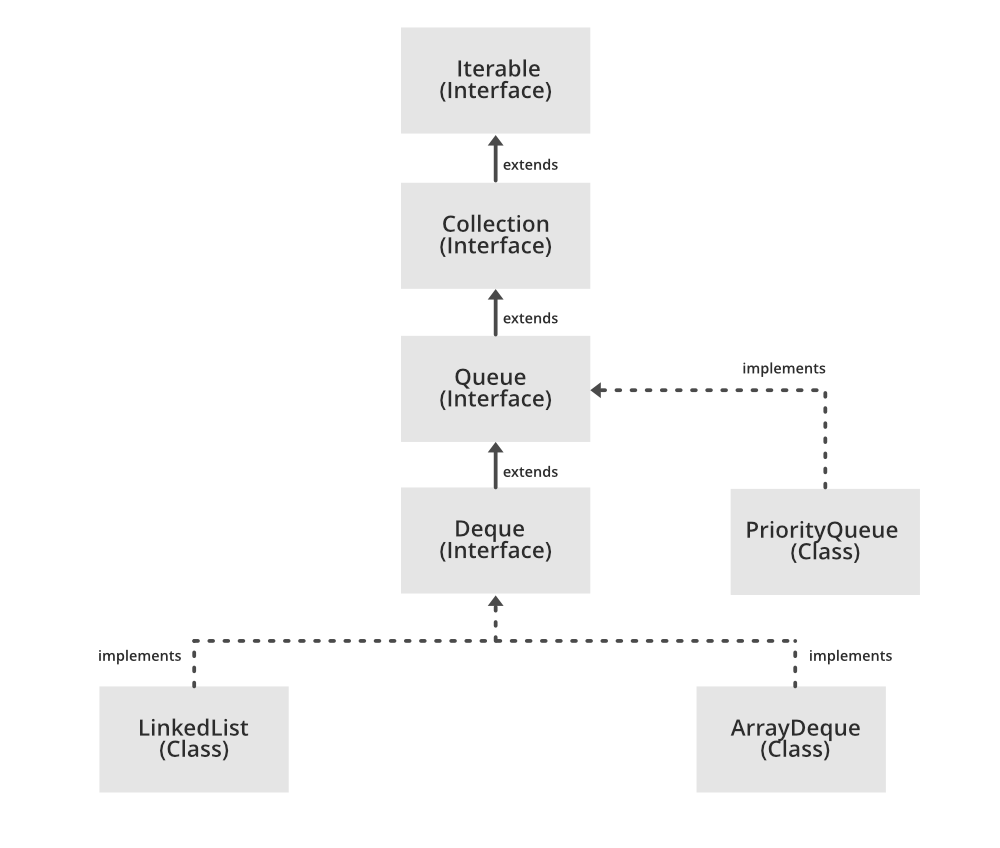
**Queue Interface In Java**

The Queue interface present in the [java.util](https://www.geeksforgeeks.org/java-util-package-java/) package and extends the [Collection interface](https://www.geeksforgeeks.org/collections-in-java-2/) is used to hold the elements about to be processed in FIFO(First In First Out) order. It is an ordered list of objects with its use limited to insert elements at the end of the list and deleting elements from the start of the list, (i.e.), it follows the FIFO or the First-In-First-Out principle.



Being an interface the queue needs a concrete class for the declaration and the most common classes are the [PriorityQueue](https://www.geeksforgeeks.org/priority-queue-class-in-java-2/" \t "_blank) and [LinkedList](https://www.geeksforgeeks.org/linked-list-in-java/) in Java.It is to be noted that both the implementations are not thread safe. *PriorityBlockingQueue* is one alternative implementation if thread safe implementation is needed.

**Declaration:** The Queue interface is declared as:

*public interface Queue extends Collection*

**Creating Queue Objects**

Since *Queue* is an [interface](https://www.geeksforgeeks.org/interfaces-in-java/), objects cannot be created of the type queue. We always need a class which extends this list in order to create an object. And also, after the introduction of [Generics](https://www.geeksforgeeks.org/generics-in-java/) in Java 1.5, it is possible to restrict the type of object that can be stored in the Queue. This type-safe queue can be defined as:

*// Obj is the type of the object to be stored in Queue  
Queue<Obj> queue = new PriorityQueue<Obj> ();*

**Example of a Queue:**

|  |
| --- |
| // Java program to demonstrate a Queue  /\*  import java.util.LinkedList;  import java.util.Queue;    public class QueueExample {    public static void main(String[] args)  {  Queue<Integer> q = new LinkedList<>();      // the queue    q.add(10);  q.add(5);  q.add(4);  q.add(15);  q.add(100);    // Display contents of the queue.  System.out.println("Elements of queue " + q);  // To view the head of queue  int head = q.peek();  System.out.println("head of queue-" + head);      // how many elements in queue  int size = q.size();  System.out.println("Size of queue-" + size);    // To remove the head of queue.  int removedele = q.remove();  System.out.println("removed element-" + removedele);    System.out.println(q);      }  }  \*/ |

**Operations on Queue Interface**

Let’s see how to perform a few frequently used operations on the queue using the [Priority Queue class](https://www.geeksforgeeks.org/priority-queue-class-in-java-2/).

**1. Adding Elements:** In order to add an element in a queue, we can use the [add() method](https://www.geeksforgeeks.org/queue-add-method-in-java/). The insertion order is not retained in the PriorityQueue. The elements are stored based on the priority order which is ascending by default.

|  |
| --- |
| // Java program to add elements  // to a Queue    import java.util.\*;    public class GFG {        public static void main(String args[])      {          Queue<String> pq = new PriorityQueue<>();            pq.add("Geeks");          pq.add("For");          pq.add("Geeks");            System.out.println(pq);      }  } |

**Output:**

[For, Geeks, Geeks]

**2. Removing Elements:** In order to remove an element from a queue, we can use the [remove() method.](https://www.geeksforgeeks.org/queue-remove-method-in-java/) If there are multiple such objects, then the first occurrence of the object is removed. Apart from that, poll() method is also used to remove the head and return it.

|  |
| --- |
| // Java program to remove elements  // from a Queue    import java.util.\*;    public class GFG {        public static void main(String args[])      {          Queue<String> pq = new PriorityQueue<>();            pq.add("Geeks");          pq.add("For");          pq.add("Geeks");            System.out.println("Initial Queue " + pq);            pq.remove("Geeks");            System.out.println("After Remove " + pq);            System.out.println("Poll Method " + pq.poll());            System.out.println("Final Queue " + pq);      }  } |

**Output:**

Initial Queue [For, Geeks, Geeks]

After Remove [For, Geeks]

Poll Method For

Final Queue [Geeks]

**3. Iterating the Queue:** There are multiple ways to iterate through the Queue. The most famous way is converting the queue to the array and traversing using the for loop. However, the queue also has an inbuilt iterator which can be used to iterate through the queue.

|  |
| --- |
| // Java program to iterate elements  // to a Queue    import java.util.\*;    public class GFG {        public static void main(String args[])      {          Queue<String> pq = new PriorityQueue<>();            pq.add("Geeks");          pq.add("For");          pq.add("Geeks");            Iterator iterator = pq.iterator();            while (iterator.hasNext()) {              System.out.print(iterator.next() + " ");          }      }  } |

**Output:**

For Geeks Geeks

**Characteristics of a Queue:** The following are the characteristics of the queue:

* The Queue is used to insert elements at the end of the queue and removes from the beginning of the queue. It follows FIFO concept.
* The Java Queue supports all methods of Collection interface including insertion, deletion etc.
* [LinkedList](https://www.geeksforgeeks.org/linked-list-in-java/), ArrayBlockingQueue and [PriorityQueue](https://www.geeksforgeeks.org/priority-queue-class-in-java-2/" \t "_blank) are the most frequently used implementations.
* If any null operation is performed on BlockingQueues, NullPointerException is thrown.
* The Queues which are available in java.util package are Unbounded Queues.
* The Queues which are available in java.util.concurrent package are the Bounded Queues.
* All Queues except the Deques supports insertion and removal at the tail and head of the queue respectively. The Deques support element insertion and removal at both ends.

**Classes which implement the Queue Interface:**

[1. PriorityQueue:](https://www.geeksforgeeks.org/priority-queue-class-in-java-2/) PriorityQueue class which is implemented in the collection framework provides us a way to process the objects based on the priority. It is known that a queue follows First-In-First-Out algorithm, but sometimes the elements of the queue are needed to be processed according to the priority, that’s when the PriorityQueue comes into play. Let’s see how to create a queue object using this class.

|  |
| --- |
| // Java program to demonstrate the  // creation of queue object using the  // PriorityQueue class    import java.util.\*;    class GfG {        public static void main(String args[])      {          // Creating empty priority queue          Queue<Integer> pQueue              = new PriorityQueue<Integer>();            // Adding items to the pQueue          // using add()          pQueue.add(10);          pQueue.add(20);          pQueue.add(15);            // Printing the top element of          // the PriorityQueue          System.out.println(pQueue.peek());            // Printing the top element and removing it          // from the PriorityQueue container          System.out.println(pQueue.poll());            // Printing the top element again          System.out.println(pQueue.peek());      }  } |

**Output:**

10

10

15

**2.**[LinkedList:](https://www.geeksforgeeks.org/linked-list-in-java/) LinkedList is a class which is implemented in the collection framework which inherently implements the [linked list data structure](https://www.geeksforgeeks.org/data-structures/linked-list/). It is a linear data structure where the elements are not stored in contiguous locations and every element is a separate object with a data part and address part. The elements are linked using pointers and addresses. Each element is known as a node. Due to the dynamicity and ease of insertions and deletions, they are preferred over the arrays or queues. Let’s see how to create a queue object using this class.

|  |
| --- |
| // Java program to demonstrate the  // creation of queue object using the  // LinkedList class    import java.util.\*;    class GfG {        public static void main(String args[])      {          // Creating empty LinkedList          Queue<Integer> ll              = new LinkedList<Integer>();            // Adding items to the ll          // using add()          ll.add(10);          ll.add(20);          ll.add(15);            // Printing the top element of          // the LinkedList          System.out.println(ll.peek());            // Printing the top element and removing it          // from the LinkedList container          System.out.println(ll.poll());            // Printing the top element again          System.out.println(ll.peek());      }  } |

**Output:**

10

10

20

**3.**[PriorityBlockingQueue:](https://www.geeksforgeeks.org/priorityblockingqueue-class-in-java/" \t "_blank) It is to be noted that both the implementations, the PriorityQueue and LinkedList are not thread-safe. PriorityBlockingQueue is one alternative implementation if thread-safe implementation is needed. PriorityBlockingQueue is an unbounded blocking queue that uses the same ordering rules as class [PriorityQueue](https://www.geeksforgeeks.org/priority-queue-class-in-java-2/" \t "_blank) and supplies blocking retrieval operations.

Since it is unbounded, adding elements may sometimes fail due to resource exhaustion resulting in [OutOfMemoryError](https://www.geeksforgeeks.org/understanding-outofmemoryerror-exception-java/" \t "_blank). Let’s see how to create a queue object using this class.

|  |
| --- |
| // Java program to demonstrate the  // creation of queue object using the  // PriorityBlockingQueue class    import java.util.concurrent.PriorityBlockingQueue;  import java.util.\*;    class GfG {      public static void main(String args[])      {          // Creating empty priority          // blocking queue          Queue<Integer> pbq         = new PriorityBlockingQueue<Integer>();            // Adding items to the pbq          // using add()          pbq.add(10);          pbq.add(20);          pbq.add(15);            // Printing the top element of          // the PriorityBlockingQueue          System.out.println(pbq.peek());            // Printing the top element and          // removing it from the          // PriorityBlockingQueue          System.out.println(pbq.ren());            // Printing the top element again          System.out.println(pbq.peek());      }  } |

**Output:**

10

10

15

**Methods of Queue Interface**

The queue interface inherits all the methods present in the [collections interface](https://www.geeksforgeeks.org/collections-in-java-2/) while implementing the following methods:

|  |  |
| --- | --- |
| Method | Description |
| [add(element)](https://www.geeksforgeeks.org/queue-add-method-in-java/) | This method is used to add elements at the tail of queue. More specifically, at the last of linked-list if it is used, or according to the priority in case of priority queue implementation. |
| [element()](https://www.geeksforgeeks.org/queue-element-method-in-java/) | This method is similar to peek(). It throws NoSuchElementException when the queue is empty. |
| [offer(element)](https://www.geeksforgeeks.org/queue-offer-method-in-java/) | This method is used to insert an element in the queue. This method is preferable to add() method since this method does not throws an exception when the capacity of the container is full since it returns false. |
| [peek()](https://www.geeksforgeeks.org/queue-peek-method-in-java/) | This method is used to view the head of queue without removing it. It returns Null if the queue is empty. |
| [poll()](https://www.geeksforgeeks.org/queue-poll-method-in-java/) | This method removes and returns the head of the queue. It returns null if the queue is empty. |
| [remove()](https://www.geeksforgeeks.org/queue-remove-method-in-java/) | This method removes and returns the head of the queue. It throws NoSuchElementException when the queue is empty. |

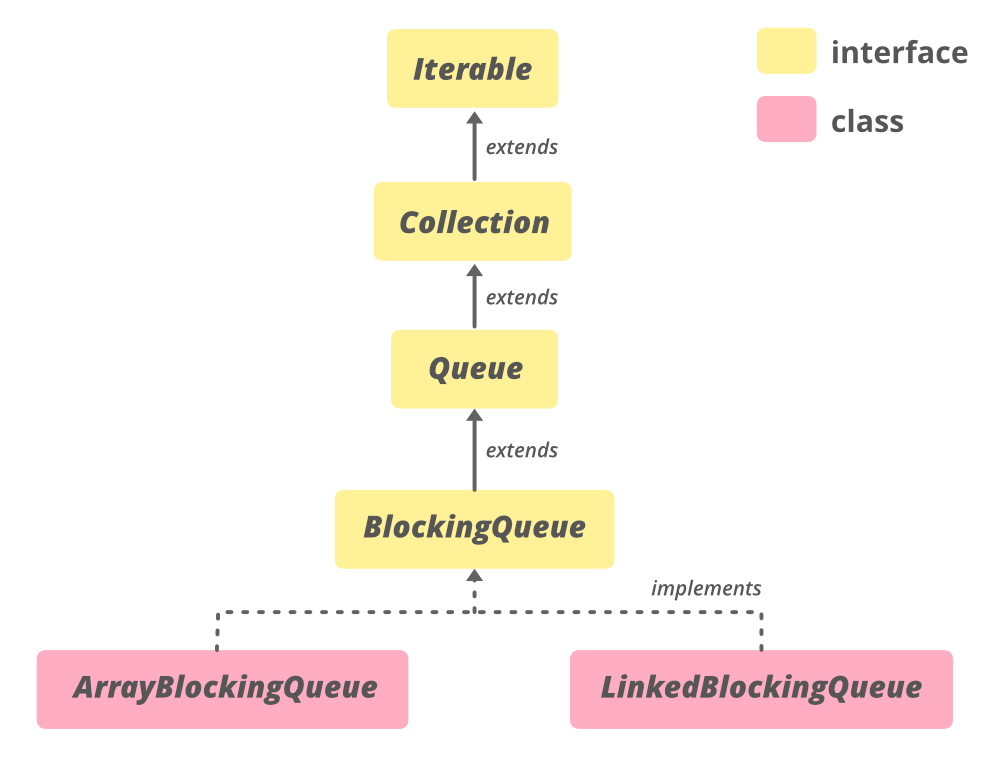
ArrayBlockingQueue Class in Java

**ArrayBlockingQueue** class is a bounded blocking queue backed by an array. By bounded, it means that the size of the Queue is fixed. Once created, the capacity cannot be changed. Attempts to put an element into a full queue will result in the operation blocking. Similarly attempts to take an element from an empty queue will also be blocked. Boundness of the ArrayBlockingQueue can be achieved initially bypassing capacity as the parameter in the constructor of ArrayBlockingQueue. This queue orders elements **FIFO (first-in-first-out)**. It means that the head of this queue is the oldest element of the elements present in this queue.

The tail of this queue is the newest element of the elements of this queue. The newly inserted elements are always inserted at the tail of the queue, and the queue retrieval operations obtain elements at the head of the queue.

This class and its iterator implement all of the optional methods of the **Collection** and **Iterator** interfaces. This class is a member of the [Java Collections Framework](https://www.geeksforgeeks.org/collections-in-java-2/).

**The Hierarchy of ArrayBlockingQueue**



This class extends [AbstractQueue<E>](https://www.geeksforgeeks.org/abstractqueue-in-java-with-examples/) and implements **Serializable**, **Iterable<E>**, **Collection<E>**, [BlockingQueue<E>](https://www.geeksforgeeks.org/blockingqueue-interface-in-java/), [Queue<E>](https://www.geeksforgeeks.org/queue-interface-java/) interfaces.

**Declaration**

*public class ArrayBlockingQueue<E> extends AbstractQueue<E> implements BlockingQueue<E>, Serializable*

Here,**E**is the type of elements stored in the collection.

**Constructors of ArrayBlockingQueue**

Here, **capacity**isthe size of the array blocking queue.

**1. ArrayBlockingQueue(int capacity):** Creates an ArrayBlockingQueue with the given (fixed) capacity and default access policy.

*ArrayBlockingQueue<E> abq = new ArrayBlockingQueue<E>(int capacity);*

**2. ArrayBlockingQueue(int capacity, boolean fair):** Creates an ArrayBlockingQueue with the given (fixed) capacity and the specified access policy. If the fair value is if true then queue accesses for threads blocked on insertion or removal, are processed in FIFO order; if false the access order is unspecified.

*ArrayBlockingQueue<E> abq = new ArrayBlockingQueue<E>(int capacity, boolean fair);*

**3. ArrayBlockingQueue(int capacity, boolean fair, Collection c):**Creates an ArrayBlockingQueue with the given (fixed) capacity, the specified access policy and initially containing the elements of the given collection, added in traversal order of the collection’s iterator. If the fair value is if true then queue accesses for threads blocked on insertion or removal, are processed in FIFO order; if false the access order is unspecified.

*ArrayBlockingQueue<E> abq = new ArrayBlockingQueue<E>(int capacity, boolean fair, Collection c);*

**Example:**

|  |
| --- |
| // Java program to demonstrate  // ArrayBlockingQueue(int initialCapacity)  // constructor    import java.util.concurrent.ArrayBlockingQueue;    public class ArrayBlockingQueueDemo {        public static void main(String[] args)      {          // define capacity of ArrayBlockingQueue          int capacity = 15;            // create object of ArrayBlockingQueue          // using ArrayBlockingQueue(int initialCapacity) constructor          ArrayBlockingQueue<Integer> abq = new ArrayBlockingQueue<Integer>(capacity);            // add  numbers          abq.add(1);          abq.add(2);          abq.add(3);            // print queue          System.out.println("ArrayBlockingQueue:" + abq);      }  } |

**Output:**

ArrayBlockingQueue:[1, 2, 3]

**Basic Operations**

**1. Adding Elements**

The [add(E e)](https://www.geeksforgeeks.org/arrayblockingqueue-add-method-in-java/) method inserts the element passed as a parameter to the method at the tail of this queue. If adding the element exceeds the capacity of the queue then the method will throw an **IllegalStateException**. This method returns true if adding of the element is successful else it will throw an IllegalStateException.

|  |
| --- |
| // Java Program to Demonstrate adding  // elements to an ArrayBlockingQueue.    import java.util.concurrent.ArrayBlockingQueue;    public class AddingElementsExample {        public static void main(String[] args)      {          // define capacity of ArrayBlockingQueue          int capacity = 15;            // create object of ArrayBlockingQueue          ArrayBlockingQueue<Integer> abq = new ArrayBlockingQueue<Integer>(capacity);            // add  numbers          abq.add(1);          abq.add(2);          abq.add(3);            // print queue          System.out.println("ArrayBlockingQueue:" + abq);      }  } |

**Output**

ArrayBlockingQueue:[1, 2, 3]

**2. Removing Elements**

The [remove(Object o)](https://www.geeksforgeeks.org/arrayblockingqueue-remove-method-in-java/) method removes a single instance of the specified element from this queue if it is present. We can say that method removes an element e such that o.equals(e) if this queue contains one or more such elements. Remove() method returns true if this queue contained the specified element which we want to remove.

|  |
| --- |
| // Java program to demonstrate removal of  // elements from an AbstractQueue    import java.util.concurrent.ArrayBlockingQueue;    public class RemovingElementsExample {        public static void main(String[] args)      {          // define capacity of ArrayBlockingQueue          int capacity = 15;            // create object of ArrayBlockingQueue          ArrayBlockingQueue<Integer> abq = new ArrayBlockingQueue<Integer>(capacity);            // add  numbers          abq.add(1);          abq.add(2);          abq.add(3);            // print queue          System.out.println("ArrayBlockingQueue:" + abq);            // remove 223          boolean response = abq.remove(2);            // print Queue          System.out.println("Removal of 2 :" + response);            // print Queue          System.out.println("queue contains " + abq);            // remove all the elements          abq.clear();            // print queue          System.out.println("ArrayBlockingQueue:" + abq);      }  } |

**Output**

ArrayBlockingQueue:[1, 2, 3]

Removal of 2 :true

queue contains [1, 3]

ArrayBlockingQueue:[]

**3. Accessing Elements**

The [peek()](https://www.geeksforgeeks.org/arrayblockingqueue-peek-method-in-java/) method provided by the **Queue** interface is used to return the head of the queue. It retrieves but does not remove, the head of this queue. If the queue is empty then this method returns null.

|  |
| --- |
| // Java program to demonstrate accessing  // elements of ArrayBlockingQueue    import java.util.concurrent.ArrayBlockingQueue;    public class AccessingElementsExample {        public static void main(String[] args)      {            // Define capacity of ArrayBlockingQueue          int capacity = 5;            // Create object of ArrayBlockingQueue          ArrayBlockingQueue<Integer> queue = new ArrayBlockingQueue<Integer>(capacity);            // Add element to ArrayBlockingQueue          queue.add(23);          queue.add(32);          queue.add(45);          queue.add(12);            // Print queue after adding numbers          System.out.println("After addding numbers queue is ");          System.out.println(queue);            // Print head of queue using peek() method          System.out.println("Head of queue " + queue.peek());      }  } |

**Output**

After addding numbers queue is

[23, 32, 45, 12]

Head of queue 23

**4. Traversing**

The [iterator()](https://www.geeksforgeeks.org/arrayblockingqueue-iterator-method-in-java/#:~:text=The%20iterator()%20method%20of,returned%20iterator%20is%20weakly%20consistent.) method of **ArrayBlockingQueue** class is used to returns an iterator of the same elements as this queue in a proper sequence. The elements returned from this method contains elements in order from first(head) to last(tail). The returned iterator is weakly consistent.

|  |
| --- |
| // Java Program to Demonstrate iterating  // over ArrayBlockingQueue.    import java.util.concurrent.ArrayBlockingQueue;  import java.util.\*;    public class TraversingExample {        public static void main(String[] args)      {          // Define capacity of ArrayBlockingQueue          int capacity = 5;            // Create object of ArrayBlockingQueue          ArrayBlockingQueue<String> queue = new ArrayBlockingQueue<String>(capacity);            // Add 5 elements to ArrayBlockingQueue          queue.offer("User");          queue.offer("Employee");          queue.offer("Manager");          queue.offer("Analyst");          queue.offer("HR");            // Print queue          System.out.println("Queue is " + queue);            // Call iterator() method and Create an iterator          Iterator iteratorValues = queue.iterator();            // Print elements of iterator          System.out.println("\nThe iterator values:");          while (iteratorValues.hasNext()) {              System.out.println(iteratorValues.next());          }      }  } |

**Output**

Queue is [User, Employee, Manager, Analyst, HR]

The iterator values:

User

Employee

Manager

Analyst

HR