**Queue Interface In Java**

The **Queue Interface** is present in [java.util](https://www.geeksforgeeks.org/java-util-package-java/) package and extends the [Collection interface](https://www.geeksforgeeks.org/collections-in-java-2/). It stores and processes the data in FIFO(First In First Out) order. It is an ordered list of objects limited to inserting elements at the end of the list and deleting elements from the start of the list.

* **No Null Elements**: Most implementations like PriorityQueue do not allow null elements.
* **Implementation Classes**: LinkedList , PriorityQueue, [ArrayDeque](https://www.geeksforgeeks.org/arraydeque-in-java/" \t "_blank), [ConcurrentLinkedQueue](https://www.geeksforgeeks.org/concurrentlinkedqueue-in-java-with-examples/" \t "_blank) (for thread-safe operations).
* **Use Cases**: Commonly used for Task scheduling, Message passing, and Buffer management in applications.
* **Iteration**: Supports iterating through elements. The order of iteration depends on the implementation.

**Example:**

// Java Program Implementing Queue Interface

import java.util.LinkedList;

import java.util.Queue;

public class QueueCreation {

public static void main(String args[])

{

// Create a Queue of Integers using LinkedList

Queue<Integer> q = new LinkedList<>();

// Displaying the Queue

System.out.println("Queue elements: " + q);

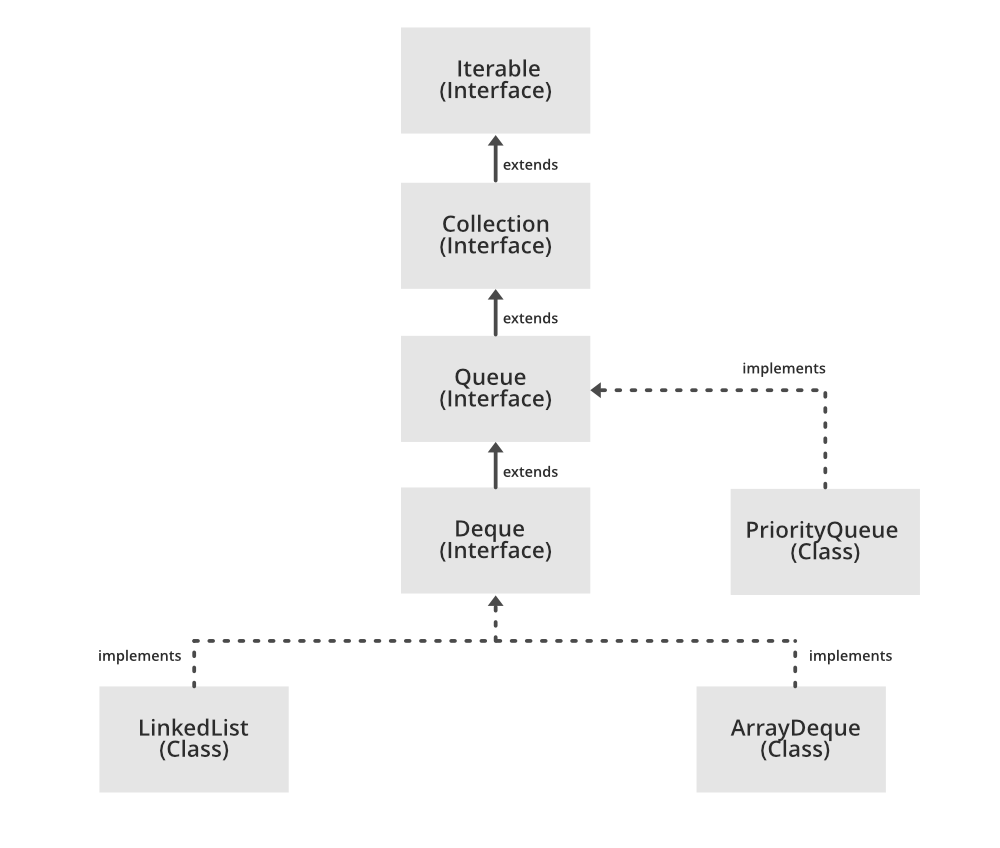
}

}

**Output**

Queue elements: []

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/) and [LinkedList](https://www.geeksforgeeks.org/linked-list-in-java/) in Java. Note that neither of these implementations is thread-safe. [PriorityBlockingQueue](https://www.geeksforgeeks.org/priorityblockingqueue-class-in-java/) is one alternative implementation if the thread-safe implementation is needed.



**Declaration of Java Queue Interface**

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> ();*

In Java, the Queue interface is a subtype of the Collection interface and represents a collection of elements in a specific order. It follows the first-in, first-out (FIFO) principle, which means that the elements are retrieved in the order in which they were added to the queue.

The Queue interface provides several methods for adding, removing, and inspecting elements in the queue. Here are some of the most commonly used methods:

* **add(element)**: Adds an element to the rear of the queue. If the queue is full, it throws an exception.
* **offer(element)**: Adds an element to the rear of the queue. If the queue is full, it returns false.
* **remove()**: Removes and returns the element at the front of the queue. If the queue is empty, it throws an exception.
* **poll()**: Removes and returns the element at the front of the queue. If the queue is empty, it returns null.
* **element()**: Returns the element at the front of the queue without removing it. If the queue is empty, it throws an exception.
* **peek()**: Returns the element at the front of the queue without removing it. If the queue is empty, it returns null.

The Queue interface is implemented by several classes in Java, including LinkedList, ArrayDeque, and PriorityQueue. Each of these classes provides different implementations of the queue interface, with different performance characteristics and features.

Overall, the Queue interface is a useful tool for managing collections of elements in a specific order, and is widely used in many different applications and industries.

**Example:**

import java.util.LinkedList;

import java.util.Queue;

public class QueueExample {

public static void main(String[] args) {

Queue<String> queue = new LinkedList<>();

// add elements to the queue

queue.add("apple");

queue.add("banana");

queue.add("cherry");

// print the queue

System.out.println("Queue: " + queue);

// remove the element at the front of the queue

String front = queue.remove();

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

// print the updated queue

System.out.println("Queue after removal: " + queue);

// add another element to the queue

queue.add("date");

// peek at the element at the front of the queue

String peeked = queue.peek();

System.out.println("Peeked element: " + peeked);

// print the updated queue

System.out.println("Queue after peek: " + queue);

}

}

**Output**

Queue: [apple, banana, cherry]

Removed element: apple

Queue after removal: [banana, cherry]

Peeked element: banana

Queue after peek: [banana, cherry, date]

**Example:**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<>();

// Adds elements {0, 1, 2, 3, 4} to

// the queue

for (int i = 0; i < 5; i++)

q.add(i);

// Display contents of the queue.

System.out.println("Elements of queue "

+ q);

// To remove the head of queue.

int removedele = q.remove();

System.out.println("removed element-"

+ removedele);

​

System.out.println(q);

// To view the head of queue

int head = q.peek();

System.out.println("head of queue-"

+ head);

// Rest all methods of collection

// interface like size and contains

// can be used with this

// implementation.

int size = q.size();

System.out.println("Size of queue-"

+ size);

}

}

**Output**

Elements of queue [0, 1, 2, 3, 4]

removed element-0

[1, 2, 3, 4]

head of queue-1

Size of queue-4

**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.

**Example:**

// 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.

**Example:**

// 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.

**Example:**

// 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/) 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 that implement the Queue Interface:**

**1. [PriorityQueue](https://www.geeksforgeeks.org/priority-queue-class-in-java/" \t "_blank):**

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 the 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.

**Example:**

// 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.

**Example:**

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/)**

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/) 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/). Let’s see how to create a queue object using this class.

**Example:**

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.poll());

// 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(int index, element)](https://www.geeksforgeeks.org/list-addint-index-e-element-method-in-java/) | This method is used to add an element at a particular index in the queue. When a single parameter is passed, it simply adds the element at the end of the queue. |
| [addAll(int index, Collection collection)](https://www.geeksforgeeks.org/list-addall-method-in-java-with-examples/) | This method is used to add all the elements in the given collection to the queue. When a single parameter is passed, it adds all the elements of the given collection at the end of the queue. |
| [size()](https://www.geeksforgeeks.org/list-size-method-in-java-with-examples/) | This method is used to return the size of the queue. |
| [clear()](https://www.geeksforgeeks.org/list-clear-method-in-java-with-examples/) | This method is used to remove all the elements in the queue. However, the reference of the queue created is still stored. |
| [remove()](https://www.geeksforgeeks.org/queue-remove-method-in-java/#:~:text=The%20remove()%20method%20of,when%20the%20Queue%20is%20empty.&text=Returns%3A%20This%20method%20returns%20the%20head%20of%20the%20Queue.) | This method is used to remove the element from the front of the queue. |
| [remove(int index)](https://www.geeksforgeeks.org/list-removeint-index-method-in-java-with-examples/) | This method removes an element from the specified index. It shifts subsequent elements(if any) to left and decreases their indexes by 1. |
| remove(element) | This method is used to remove and return the first occurrence of the given element in the queue. |
| [get(int index)](https://www.geeksforgeeks.org/list-get-method-in-java-with-examples/) | This method returns elements at the specified index. |
| [set(int index, element)](https://www.geeksforgeeks.org/arraylist-set-method-in-java-with-examples/) | This method replaces elements at a given index with the new element. This function returns the element which was just replaced by a new element. |
| [indexOf(element)](https://www.geeksforgeeks.org/list-indexof-method-in-java-with-examples/) | This method returns the first occurrence of the given element or *-1* if the element is not present in the queue. |
| [lastIndexOf(element)](https://www.geeksforgeeks.org/list-lastindexof-method-in-java-with-examples/) | This method returns the last occurrence of the given element or *-1* if the element is not present in the queue. |
| equals(element) | This method is used to compare the equality of the given element with the elements of the queue. |
| hashCode() | This method is used to return the hashcode value of the given queue. |
| isEmpty() | This method is used to check if the queue is empty or not. It returns true if the queue is empty, else false. |
| contains(element) | This method is used to check if the queue contains the given element or not. It returns true if the queue contains the element. |
| [containsAll(Collection collection)](https://www.geeksforgeeks.org/list-containsall-method-in-java-with-examples/) | This method is used to check if the queue contains all the collection of elements. |
| sort(Comparator comp) | This method is used to sort the elements of the queue on the basis of the given [comparator](https://www.geeksforgeeks.org/comparator-interface-java/). |
| boolean add(object) | This method is used to insert the specified element into a queue and return true upon success. |
| boolean offer(object) | This method is used to insert the specified element into the queue. |
| Object poll() | This method is used to retrieve and removes the head of the queue, or returns null if the queue is empty. |
| Object element() | This method is used to retrieves, but does not remove, the head of queue. |
| Object peek() | This method is used to retrieves, but does not remove, the head of this queue, or returns null if this queue is empty. |

**Advantages of using the Queue Interface in Java**

* **Order preservation**: The Queue interface provides a way to store and retrieve elements in a specific order, following the first-in, first-out (FIFO) principle.
* **Flexibility**: The Queue interface is a subtype of the Collection interface, which means that it can be used with many different data structures and algorithms, depending on the requirements of the application.
* **Thread**–**safety**: Some implementations of the Queue interface, such as the java.util.concurrent.ConcurrentLinkedQueue class, are thread-safe, which means that they can be accessed by multiple threads simultaneously without causing conflicts.
* **Performance**: The Queue interface provides efficient implementations for adding, removing, and inspecting elements, making it a useful tool for managing collections of elements in performance-critical applications.

**Disadvantages of using the Queue Interface in Java**

* **Limited functionality:** The Queue interface is designed specifically for managing collections of elements in a specific order, which means that it may not be suitable for more complex data structures or algorithms.
* **Size restrictions:** Some implementations of the Queue interface, such as the ArrayDeque class, have a fixed size, which means that they cannot grow beyond a certain number of elements.
* **Memory usage:**Depending on the implementation, the Queue interface may require more memory than other data structures, especially if it needs to store additional information about the order of the elements.
* **Complexity**: The Queue interface can be difficult to use and understand for novice programmers, especially if they are not familiar with the principles of data structures and algorithms.