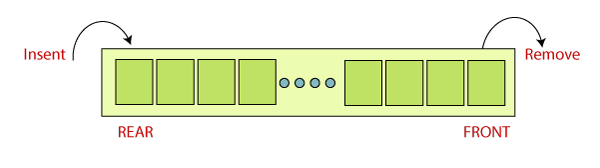
Java Queue

A queue is another kind of linear data structure that is used to store elements just like any other data structure but in a particular manner. In simple words, we can say that the queue is a type of data structure in the Java programming language that stores elements of the same kind. The components in a queue are stored in a FIFO (First In, First Out) behavior. There are two ends in the queue collection, i.e., front & rear. Queue has two ends that is front and rear.

The following figure perfectly describes the FIFO (First In, First Out) property of the Java queue.



As explained in the preceding image, we can see that the queue is a linear data structure with two terminals, i.e., start (front) and end (rear). Components are added inside the queue from the rear end of the queue and the components are extracted from the front end of the queue.

The Queue is an interface in the [Java](https://www.javatpoint.com/java-tutorial) that belongs to [Java.util package](https://www.javatpoint.com/java-util-package). It also extends the [Collection interface](https://www.javatpoint.com/java-collection).

Play Video

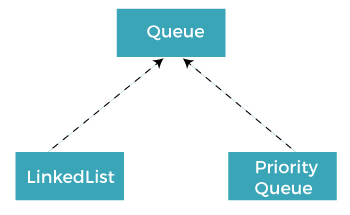
The generic representation of the Java Queue interface is shown below:

1. **public** **interface** Queue<T> **extends** Collection<T>

As we have discussed above that the Queue is an interface, therefore we can also say that the queue cannot be instantiated because interfaces cannot be instantiated. If a user wants to implement the functionality of the Queue interface in Java, then it is mandatory to have some solid classes that implement the Queue interface.

In Java programming language, there are two different classes which are used to implement the Queue interface. These classes are:

* [LinkedList](https://www.javatpoint.com/java-linkedlist)
* [PriorityQueue](https://www.javatpoint.com/java-priorityqueue)



Characteristics of the Java Queue

The Java Queue can be considered as one of the most important data structures in the programming world. Java Queue is attractive because of its properties. The significant properties of the Java Queue data structure are given as follows:

* Java Queue obeys the FIFO (First In, First Out) manner. It indicates that elements are entered in the queue at the end and eliminated from the front.
* The Java Queue interface gives all the rules and processes of the Collection interface like inclusion, deletion, etc.
* There are two different classes that are used to implement the Queue interface. These classes are LinkedList and PriorityQueue.
* Other than these two, there is a class that is, Array Blocking Queue that is used to implement the Queue interface.
* There are two types of queues, Unbounded queues and Bounded queues. The Queues that are a part of the java.util package are known as the Unbounded queues and bounded queues are the queues that are present in java.util.concurrent package.
* The Deque or (double-ended queue) is also a type of queue that carries the inclusion and deletion of elements from both ends.
* The deque is also considered thread-safe.
* Blocking Queues are also one of the types of queues that are also thread-safe. The Blocking Queues are used to implement the producer-consumer queries.
* Blocking Queues do not support null elements. In Blocking queues, if any work similar to null values is tried, then the NullPointerException is also thrown.

Implementation of Queue

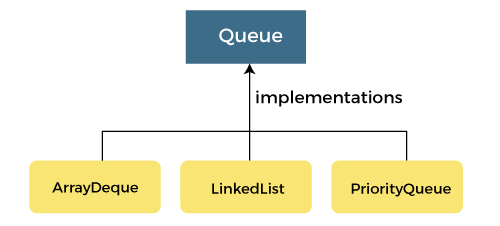
Classes used in implementation of Queue

The classes that are used to implement the functionalities of the queue are given as follows:

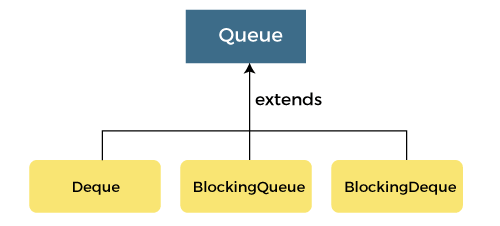
* [ArrayDeque](https://www.programiz.com/java-programming/arraydeque)
* [LinkedList](https://www.programiz.com/java-programming/linkedlist)
* [PriorityQueue](https://www.programiz.com/java-programming/priorityqueue)

Interfaces used in implementation of Queue

The Java interfaces are also used in the implementation of the Java queue. The interfaces that are used to implement the functionalities of the queue are given as follows:



* Deque
* Blocking Queue
* Blocking Deque



Java Queue Class Methods

In the Java queue, there are many methods that are used very commonly. The Queue interface promotes different methods like insert, delete, peek, etc. Some of the operations of the Java queue raise an exception whereas some of these operations return a particular value when the program is completed.

Note - In Java SE 8, there are no changes made in the Java queue collection. These methods which are defined under are further prepared in the succeeding versions of the Java programming language. For example, Java SE 9.

Different methods of the Java Queue are defined below:

|  |  |  |
| --- | --- | --- |
| **Method** | **Method Prototype** | **Description** |
| add | boolean add(E e) | Adds element e to the queue at the end (tail) of the queue without violating the restrictions on the capacity. Returns true if success or IllegalStateException if the capacity is exhausted. |
| peek | E peek() | Returns the head (front) of the queue without removing it. |
| element | E element() | Performs the same operation as the peek () method. Throws NoSuchElementException when the queue is empty. |
| remove | E remove() | Removes the head of the queue and returns it. Throws NoSuchElementException if queue is empty. |
| poll | E poll() | Removes the head of the queue and returns it. If the queue is empty, it returns null. |
| Offer | boolean offer(E e) | Insert the new element e into the queue without violating capacity restrictions. |
| size | int size() | Returns the size or number of elements in the queue. |

Java Queue Array Implementation

Queue implementation is not as straightforward as a stack implementation.

To implement queue using Arrays, we first declare an array that holds n number of elements.

Then we define the following operations to be performed in this queue.

**1) Enqueue:** An operation to insert an element in the queue is Enqueue (function queue Enqueue in the program). For inserting an element at the rear end, we need first to check if the queue is full. If it is full, then we cannot insert the element. If rear < n, then we insert the element in the queue.

**2) Dequeue:** The operation to delete an element from the queue is Dequeue (function queue Dequeue in the program). First, we check whether the queue is empty. For dequeue operation to work, there has to be at least one element in the queue.

**3) Front:** This method returns the front of the queue.

**4) Display:** This method traverses the queue and displays the elements of the queue.

Java Queue Program

The following Java program demonstrates the implementation of Queue.

**QueueArrayImplementation.java**

1. **class** Queue {
3. **private** **static** **int** front, rear, capacity;
4. **private** **static** **int** queue[];
6. Queue(**int** size) {
7. front = rear = 0;
8. capacity = size;
9. queue = **new** **int**[capacity];
10. }
12. // insert an element into the queue
13. **static** **void** queueEnqueue(**int** item)  {
14. // check if the queue is full
15. **if** (capacity == rear) {
16. System.out.printf("\nQueue is full\n");
17. **return**;
18. }
20. // insert element at the rear
21. **else** {
22. queue[rear] = item;
23. rear++;
24. }
25. **return**;
26. }
28. //remove an element from the queue
29. **static** **void** queueDequeue()  {
30. // check if queue is empty
31. **if** (front == rear) {
32. System.out.printf("\nQueue is empty\n");
33. **return**;
34. }
36. // shift elements to the right by one place uptil rear
37. **else** {
38. **for** (**int** i = 0; i < rear - 1; i++) {
39. queue[i] = queue[i + 1];
40. }

43. // set queue[rear] to 0
44. **if** (rear < capacity)
45. queue[rear] = 0;
47. // decrement rear
48. rear--;
49. }
50. **return**;
51. }
53. // print queue elements
54. **static** **void** queueDisplay()
55. {
56. **int** i;
57. **if** (front == rear) {
58. System.out.printf("Queue is Empty\n");
59. **return**;
60. }
62. // traverse front to rear and print elements
63. **for** (i = front; i < rear; i++) {
64. System.out.printf(" %d , ", queue[i]);
65. }
66. **return**;
67. }
69. // print front of queue
70. **static** **void** queueFront()
71. {
72. **if** (front == rear) {
73. System.out.printf("Queue is Empty\n");
74. **return**;
75. }
76. System.out.printf("\nFront Element of the queue: %d", queue[front]);
77. **return**;
78. }
79. }
81. **public** **class** QueueArrayImplementation {
82. **public** **static** **void** main(String[] args) {
83. // Create a queue of capacity 4
84. Queue q = **new** Queue(4);
86. System.out.println("Initial Queue:");
87. // print Queue elements
88. q.queueDisplay();
90. // inserting elements in the queue
91. q.queueEnqueue(10);
92. q.queueEnqueue(30);
93. q.queueEnqueue(50);
94. q.queueEnqueue(70);
96. // print Queue elements
97. System.out.println("Queue after Enqueue Operation:");
98. q.queueDisplay();
100. // print front of the queue
101. q.queueFront();
103. // insert element in the queue
104. q.queueEnqueue(90);
106. // print Queue elements
107. q.queueDisplay();
109. q.queueDequeue();
110. q.queueDequeue();
111. System.out.printf("\nQueue after two dequeue operations:");
113. // print Queue elements
114. q.queueDisplay();
116. // print front of the queue
117. q.queueFront();
118. }
119. }

**Output:**

Initial Queue:

Queue is Empty

Queue after Enqueue Operation:

10 , 30 , 50 , 70 ,

Front Element of the queue: 10

Queue is full

10 , 30 , 50 , 70 ,

Queue after two dequeue operations: 50 , 70 ,

Front Element of the queue: 50

Java Queue Linked List Implementation

As we have implemented the Queue data structure using Arrays in the above program, we can also implement the Queue using Linked List.

We will implement the same methods enqueue, dequeue, front, and display in this program. The difference is that we will be using the Linked List data structure instead of Array.

The below program demonstrates the Linked List implementation of Queue in Java.

**QueueLLImplementation.java**

1. **class** LinkedListQueue
2. {
3. **private** Node front, rear;
4. **private** **int** queueSize; // queue size
6. //linked list node
7. **private** **class** Node {
8. **int** data;
9. Node next;
10. }
11. //default constructor - initially front & rear are null; size=0; queue is empty
12. **public** LinkedListQueue()
13. {
14. front = **null**;
15. rear = **null**;
16. queueSize = 0;
17. }
18. //check if the queue is empty
19. **public** **boolean** isEmpty()
20. {
21. **return** (queueSize == 0);
22. }
23. //Remove item from the front of the queue.
24. **public** **int** dequeue()
25. {
26. **int** data = front.data;
27. front = front.next;
28. **if** (isEmpty())
29. {
30. rear = **null**;
31. }
32. queueSize--;
33. System.out.println("Element " + data+ " removed from the queue");
34. **return** data;
35. }
36. //Add data at the rear of the queue.
37. **public** **void** enqueue(**int** data)
38. {
39. Node oldRear = rear;
40. rear = **new** Node();
41. rear.data = data;
42. rear.next = **null**;
43. **if** (isEmpty())
44. {
45. front = rear;
46. }
47. **else**  {
48. oldRear.next = rear;
49. }
50. queueSize++;
51. System.out.println("Element " + data+ " added to the queue");
52. }
53. //print front and rear of the queue
54. **public** **void** print\_frontRear() {
55. System.out.println("Front of the queue:" + front.data
56. + " Rear of the queue:" + rear.data);
57. }
58. }
59. **class** QueueLLImplementation{
60. **public** **static** **void** main(String a[]){
61. LinkedListQueue queue = **new** LinkedListQueue();
62. queue.enqueue(6);
63. queue.enqueue(3);
64. queue.print\_frontRear();
65. queue.enqueue(12);
66. queue.enqueue(24);
67. queue.dequeue();
68. queue.dequeue();
69. queue.enqueue(9);
70. queue.print\_frontRear();
71. }
72. }

**Output:**

Element 6 added to the queue

Element 3 added to the queue

Front of the queue:6 Rear of the queue:3

Element 12 added to the queue

Element 24 added to the queue

Element 6 removed from the queue

Element 3 removed from the queue

Element 9 added to the queue

Front of the queue:12 Rear of the queue:9