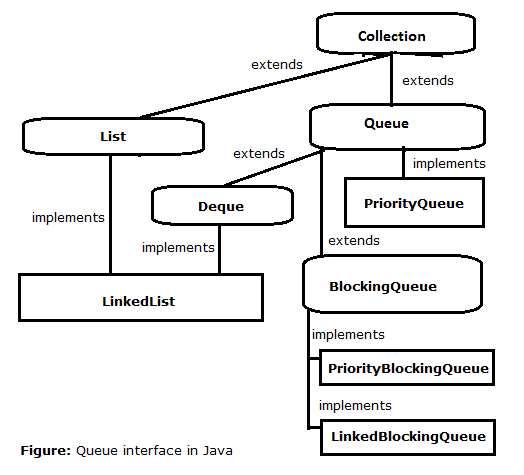
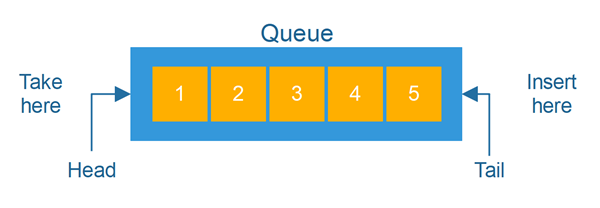
**Queues in Collection**

**The queue** is the child interface of Collection(I).



In the above image LinkedList also started implementing Deque from version 1.5v.

The *Java* *Queue* interface, java.util.Queue represents a data structure designed to have elements inserted at the end of the queue, and elements removed from the beginning of the queue. This is similar to how a queue in a supermarket works.



The Java Queue interface is a subtype of the [**Java Collection**](https://jenkov.com/tutorials/java-collections/collection.html) interface. It represents an ordered sequence of objects just like a [**Java List**](https://jenkov.com/tutorials/java-collections/list.html), but its intended use is slightly different. Because the Java Queue interface is a subtype of the Java Collection interface, all methods in the Collection interface are also available in the Queue interface.

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.

Since Queue is an interface you need to instantiate a concrete implementation of the interface in order to use it. You can choose between the following Queue implementations in the Java Collections API:

* java.util.LinkedList
* java.util.PriorityQueue

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

public interface Queue extends Collection

LinkedList is a pretty standard queue implementation. Elements in the queue are stored internally in a standard linked list data structure. This makes it fast to insert elements at the end (tail) of the list, and remove elements from the beginning (head) of the list.

PriorityQueue stores its elements internally according to their natural order (if they implement Comparable), or according to a Comparator passed to the PriorityQueue.

There are also Queue implementations in the java.util.concurrent package, but I will leave the concurrency utilities out of this tutorial.

Here are a few examples of how to create a Queue instance:

Queue queueA = new LinkedList();

Queue queueB = new PriorityQueue();

In most Queue implementations the head and tail of the queue are at opposite ends. It is possible, however, to implement the Queue interface so that the head and tail of the queue is in the same end. In that case you would have a stack.

**Generic Queue**

By default, you can put any Object into a Queue, but from Java 5, Java Generics makes it possible to limit the types of object you can insert into a Queue. Here is an example:

Queue<MyObject> queue = new PriorityQueue<MyObject>();

**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.

Queue Methods Java

The Queue declares a number of methods. As interface’s methods they should be represented in all classes that implement Queue. The most important Queue Methods, Java:

* **Boolean offer()** – inserts a new element into the queue if it is possible
* **Boolean add(E e)** – inserts a new element into the queue if it is possible. Returns true in case of success and throws an IllegalStateException if there is no space.
* **Object poll()** – retrieves and removes an element from the head of the. Returns null if the queue is empty.
* **Object remove()** – retrieves and removes an element from the head of the queue.
* **Object peek()** – retrieves, but doesn’t remove an element from the head of the queue. Returns null if the queue is empty.
* **Object element()** – retrieves, but doesn’t remove an element from the head of the queue.

**Implementation Class of Queue(I)**

A **PriorityQueue** is used when the objects are supposed to be processed based on the priority. It is known that a [Queue](https://www.geeksforgeeks.org/queue-interface-java/) 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.

**Declaration:**

public class PriorityQueue<E> extends AbstractQueue<E> implements Serializable

where E is the type of elements held in this queue

The class implements **Serializable**, **Iterable<E>**, **Collection<E>**, [Queue<E>](https://www.geeksforgeeks.org/queue-interface-java/) interfaces.

A few **important points on Priority Queue** are as follows:

* PriorityQueue doesn’t permit null.( Throws Null pointer Exception)
* Duplicates are allowed in Priority Queue.
* We can’t create a PriorityQueue of Objects that are non-comparable and for heterogeneous. If we are depending on natural sorting order then objects should be Comparable and Homogeneous.
* PriorityQueue are unbound queues.
* The head of this queue is the least element with respect to the specified ordering. If multiple elements are tied for the least value, the head is one of those elements — ties are broken arbitrarily.
* Since PriorityQueue is not thread-safe, java provides [PriorityBlockingQueue](https://www.geeksforgeeks.org/priorityblockingqueue-class-in-java/#:~:text=PriorityBlockingQueue%20is%20an%20unbounded%20blocking,and%20supplies%20blocking%20retrieval%20operations.&text=PriorityBlockingQueue%20class%20and%20its%20iterator,the%20Collection%20and%20Iterator%20interfaces.) class that implements the [BlockingQueue](https://www.geeksforgeeks.org/blockingqueue-interface-in-java/#:~:text=Methods%20in%20Blocking%20Queue%20Interface&text=Removes%20all%20available%20elements%20from,them%20to%20the%20given%20collection.&text=Removes%20at%20most%20the%20given,them%20to%20the%20given%20collection.) interface to use in a java multithreading environment.
* The queue retrieval operations poll, remove, peek, and element access the element at the head of the queue.
* It provides O(log(n)) time for add and poll methods.
* It inherits methods from **AbstractQueue**, **AbstractCollection**, **Collection,** and **Object** class.

**Constructors:**

**1. PriorityQueue():** Creates a PriorityQueue with the default initial capacity (11) that orders its elements according to their natural ordering.

*PriorityQueue<E> pq = new PriorityQueue<E>();*

**2. PriorityQueue(Collection<E> c):** Creates a PriorityQueue containing the elements in the specified collection.

*PriorityQueue<E> pq = new PriorityQueue<E>(Collection<E> c);*

**3. PriorityQueue(int initialCapacity)**: Creates a PriorityQueue with the specified initial capacity that orders its elements according to their natural ordering.

*PriorityQueue<E> pq = new PriorityQueue<E>(int initialCapacity);*

**4. PriorityQueue(int initialCapacity, Comparator<E> comparator):** Creates a PriorityQueue with the specified initial capacity that orders its elements according to the specified comparator.

*PriorityQueue<E> pq = new PriorityQueue(int initialCapacity, Comparator<E> comparator);*

**5. PriorityQueue(PriorityQueue<E> c)**: Creates a PriorityQueue containing the elements in the specified priority queue.

*PriorityQueue<E> pq = new PriorityQueue(PriorityQueue<E> c);*

**6. PriorityQueue(SortedSet<E> c)**: Creates a PriorityQueue containing the elements in the specified sorted set.

*PriorityQueue<E> pq = new PriorityQueue<E>(SortedSet<E> c);*

NOTE:  By default, the priority queue in Java is **min Priority queue with natural ordering**. To make it max, we have to use a custom comparator so that head of the queue returns the greatest element in the queue.

public class PriorityQueueDemo {

public static void main(String[] args) {

Queue<Integer> integerQueue = new PriorityQueue<Integer>();

integerQueue.add(15);

integerQueue.add(2);

integerQueue.add(1);

integerQueue.add(14);

integerQueue.add(16);

System.out.println(integerQueue);

for (Iterator iterator = integerQueue.iterator(); iterator.hasNext();) {

System.out.println(integerQueue.poll());

}

System.out.println(integerQueue);

}

}

Output:

[1, 14, 2, 15, 16]

1 2 14 15 16 // Natural sorting order.

[]

If we want to max the priority as MAX then we need to use explicit Comparator.

public class PriorityQueueDemo {

public static void main(String[] args) {

Queue<Integer> integerQueue = new PriorityQueue<Integer>((x, y) -> Integer.compare(y, x));

integerQueue.add(15);

integerQueue.add(2);

integerQueue.add(1);

integerQueue.add(14);

integerQueue.add(16);

System.out.println(integerQueue);

for (Iterator iterator = integerQueue.iterator(); iterator.hasNext();) {

System.out.println(integerQueue.poll());

}

System.out.println(integerQueue);

}

}

Output:

[16, 15, 1, 2, 14]

16 15 14 2 1

[]

### ****Methods in PriorityQueue class****

| METHOD | DESCRIPTION |
| --- | --- |
| [add(E e)](https://www.geeksforgeeks.org/priorityqueue-add-method-in-java/) | Inserts the specified element into this priority queue. |
| [clear()](https://www.geeksforgeeks.org/priorityqueue-clear-method-in-java/#:~:text=clear()%20method%20is%20used,only%20empty%20an%20existing%20PriorityQueue.) | Removes all of the elements from this priority queue. |
| [comparator()](https://www.geeksforgeeks.org/priorityqueue-comparator-method-in-java/) | Returns the comparator used to order the elements in this queue, or null if this queue is sorted according to the natural ordering of its elements. |
| [contains​(Object o)](https://www.geeksforgeeks.org/priorityqueue-contains-method-in-java/#:~:text=PriorityQueue.,any%20particular%20element%20or%20not.&text=Return%20Value%3A%20The%20method%20returns,queue%20otherwise%20it%20returns%20False.) | Returns true if this queue contains the specified element. |
| forEach​(Consumer<? super E> action) | Performs the given action for each element of the Iterable until all elements have been processed or the action throws an exception. |
| [iterator()](https://www.geeksforgeeks.org/priorityqueue-iterator-method-in-java/) | Returns an iterator over the elements in this queue. |
| [offer​(E e)](https://www.geeksforgeeks.org/priorityqueue-offer-method-in-java/) | Inserts the specified element into this priority queue. |
| [remove​(Object o)](https://www.geeksforgeeks.org/priorityqueue-remove-method-in-java/) | Removes a single instance of the specified element from this queue, if it is present. |
| removeAll​(Collection<?> c) | Removes all of this collection’s elements that are also contained in the specified collection (optional operation). |
| removeIf​(Predicate<? super E> filter) | Removes all of the elements of this collection that satisfy the given predicate. |
| retainAll​(Collection<?> c) | Retains only the elements in this collection that are contained in the specified collection (optional operation). |
| [spliterator()](https://www.geeksforgeeks.org/priorityqueue-spliterator-method-in-java/) | Creates a late-binding and fail-fast Spliterator over the elements in this queue. |
| [toArray()](https://www.geeksforgeeks.org/priorityqueue-toarray-method-in-java/#:~:text=toArray(arr%5B%5D)%20method%20in,the%20previous%20method%20without%20parameters.) | Returns an array containing all of the elements in this queue. |
| [toArray​(T[] a)](https://www.geeksforgeeks.org/priorityqueue-toarray-method-in-java/#:~:text=toArray(arr%5B%5D)%20method%20in,the%20previous%20method%20without%20parameters.) | Returns an array containing all of the elements in this queue; the runtime type of the returned array is that of the specified array. |

**Subinterfaces of Java Queue**

*Queue* interface is inherited by 4 subinterfaces – **BlockingDeque<E>, BlockingQueue<E>, Deque<E>, TransferQueue<E>**. You may divide them into 3 groups: Deques, Blocking Queues and Transfer Queues with BlockingDeque belonging to the two first. Let's take a glimpse at these groups.

**Deques**

Deque means **D**ouble-**E**nded **Q**ueue and supports addition or removal from either tail of the data as a queue (first-in-first-out/FIFO) or from the head as another popular data structure called **stack** (last-in-first-out/LIFO). **Classes that implement Deque Interface:** ArrayDeque, ConcurrentLinkedDeque, LinkedBlockingDeque, LinkedList.

**Blocking Queues**

A blocking queue is a queue that blocks a thread in two cases:

* thread is trying to get elements from an empty queue
* thread is trying to put elements in the full queue

When a thread tries to get items from an empty queue, it waits until some other thread puts the items into the queue. Similarly, when a thread tries to put elements into a full queue, it waits until some other thread takes the elements out of the queue to get free space for the elements. Sure, the concept of "full queue" implies that the queue has a limited size, which is usually specified in the constructor. Standard Blocking Queues include LinkedBlockingQueue, SynchronousQueue, and ArrayBlockingQueue. Implementing classes of **BlockingQueue** interface: ArrayBlockingQueue, DelayQueue, LinkedBlockingDeque, LinkedBlockingQueue, LinkedTransferQueue, PriorityBlockingQueue, SynchronousQueue.

**BlockingDeque** is a subinterface for BlockingQueue. BlockingDeque such as BlockingQueue is a blocking queue, but bidirectional. So it inherits the properties of the Deque interface. It is oriented to multi-threaded execution, doesn’t allow zero elements and capacity could be limited. Implementations of the BlockingDeque interface block the operation of getting elements if the queue is empty, and adding an element into the queue if it is full.

**Transfer Queues**

TransferQueue interface extends BlockingQueue interface. However unlike the implementation of BlockingQueue interface queues, where threads can be blocked if the queue is empty (reading), or if the queue is full (writing), TransferQueue interface queues block the write stream until another stream retrieves the element. Use a transfer method for this. In other words, the implementation of BlockingQueue guarantees that the element created by the Producer must be in the queue, while the implementation of TransferQueue guarantees that the Producer element is "received" by the Consumer. There is only one official Java implementation of TransferQueue interface — LinkedTransferQueue.

**Java Queue Implementations**

There are many classes that implement Queue interface:

* **AbstractQueue** according to Queue Java 8 docs, this abstract class provides basic implementations of some Queue operations. It doesn’t allow null elements. There are 3 more methods add, remove, and element based on Queue classical **offer**, **poll**, and **peek**, respectively. However they throw exceptions instead of indicating failure via false or null returns.
* **ArrayBlockingQueue** — a fixed size FIFO blocking queue backed by an array
* **ArrayDeque** — resizable array implementation of the Deque interface
* **ConcurrentLinkedDeque** — an unbounded concurrent deque based on linked nodes.
* **ConcurrentLinkedQueue** — an unbounded thread-safe queue based on linked nodes.
* **DelayQueue** — a time-based scheduling queue backed by a heap
* **LinkedBlockingDeque** — the concurrent implementation of the Deque interface.
* **LinkedBlockingQueue** — an optionally bounded FIFO blocking queue backed by linked nodes
* **LinkedList** — doubly-linked list implementation of the List and Deque interfaces. Implements all optional list operations, and permits all elements (including null)
* **LinkedTransferQueue** — an unbounded TransferQueue based on linked nodes
* **PriorityBlockingQueue** — an unbounded blocking priority queue backed by a heap
* **PriorityQueue** — a priority queue based on the heap data structure
* **SynchronousQueue** — a blocking queue where each insert operation must wait for a corresponding remove operation by another thread, and vice versa.

The most popular implementations are LinkedList, ArrayBlockingQueue and PriorityQueue. As we have already completed Priority Queue we will look into LinkedList and ArrayBlockingQueue.

**LinkedList**

Class LinkedList in Java implements List and Deque interfaces. So, it is a combination of List and Deque, a two-way queue, that supports adding and removing elements from both sides. In Java LinkedList is doubly-linked List: every element of List calls Node and contains an object and references to two neighboring objects — the previous and the next. You may say that LinkedList isn’t very effective in terms of using memory. That’s true, but this data structure can be useful in case of the insert and delete operations performance. However, it happens only if you use iterators for them (in this case it occurs in constant time). Access operations by index are performed by searching from the beginning of the end (whichever is closer) to the desired element. However, don’t forget about additional costs for storing references between elements. So, LinkedList is the most popular queue implementation in Java. It is an implementation of List and Deque as well and it allows us to create a bidirectional queue consisting of any objects including **null**. LinkedList is a collection of elements.

**LinkedList Constructors**

**LinkedList()** without parameters is used to construct an empty list.

**LinkedList(Collection<? extends E> c)** is for creating a list containing the elements of the specified collection, in order, they are returned by the collection's iterator.

**Main LinkedList Methods:**

* add(E element) Appends the specified element to the end of this list;
* add(int index, E element) Inserts the element at the specified position index;
* get(int index) Returns the element at the specified position in this list;
* remove(int index) Removes the element that is at position index;
* remove(Object o) Removes the first occurrence of ?o element from this list if it is there.
* remove() Retrieves and removes the first element of the list.
* addFirst(), addLast() add an element to the beginning/end of a list
* clear() removes all elements from the list
* contains(Object o) returns true if the list contains the o element.
* indexOf(Object o) returns the index of the first occurrence of the o element, or -1 if it isn’t in the list.
* set(int index, E element) replaces the element at index position with the element
* size()Returns the quantity of elements in the list.
* toArray() returns an array containing all list’s elements from first to the last element.
* pop() that pops an element from the stack (represented by the list)
* push(E e) that pushes an element onto the stack (represented by this list)

Example:

package com.skg.queue;

import java.util.LinkedList;

public class LinkedListTest {

public static void main(String args[]) {

LinkedList<Integer> myLinkedList = new LinkedList<Integer>();

myLinkedList.add(1);

myLinkedList.add(2);

myLinkedList.add(4);

System.out.println("three added elements: " + myLinkedList);

// put one element into the head, not to the tail:

myLinkedList.push(5);

System.out.println("The new element last in will be the first: " + myLinkedList);

// add new element at the specified position:

myLinkedList.add(4, 3);

// put one element into the head, not to the tail (same as push):

myLinkedList.addFirst(6);

System.out.println(myLinkedList);

// now remove element no 2 (it is 1):

myLinkedList.remove(2);

System.out.println(myLinkedList);

// now remove the head of the list

myLinkedList.pop();

System.out.println(myLinkedList);

// remove with the other method

myLinkedList.remove();

System.out.println(myLinkedList);

// and with one more

myLinkedList.poll();

System.out.println(myLinkedList);

}

}

Output:

three added elements: [1, 2, 4]

The new element last in will be the first: [5, 1, 2, 4]

[6, 5, 1, 2, 4, 3]

[6, 5, 2, 4, 3]

[5, 2, 4, 3]

[2, 4, 3]

[4, 3]

**ArrayBlockingQueue** supports an additional fairness policy with **fair** parameter in the constructor to order the work of waiting flows of producers (inserting elements) and consumers (extracting elements). By default, the order is not guaranteed. However if the queue is created with "fair == true", the implementation of the ArrayBlockingQueue class provides thread access in FIFO order. Equity typically reduces bandwidth, but also reduces volatility and prevents running out of resources.

**ArrayBlockingQueue Class сonstructors**

* **ArrayBlockingQueue (int capacity)** creates a queue of fixed capacity and with a default access policy.
* **ArrayBlockingQueue (int capacity, boolean fair)** creates a queue with a fixed capacity and a specified access policy.
* **ArrayBlockingQueue (int capacity, boolean fair, Collection <? extends E> c)** creates a queue with a fixed capacity specified by the access policy and includes elements in the queue.

Here we’ve got the BlockingQueueExample example. We create a queue of the ArrayBlockingQueue with a capacity of one element and a fair flag. Two threads are started. The first of them, Producer thread, queues messages from the messages array using the put method. The second one, Consumer, thread reads elements from the queue using **take** method and displays them in the console. The order of elements is the natural one for the queue.

package com.skg.queue;

import java.util.concurrent.\*;

public class ArrayBlockingQueueDemo {

private BlockingQueue<Integer> blockingQueue;

private final Integer[] myArray = { 1, 2, 3, 4, 5 };

public ArrayBlockingQueueDemo() {

blockingQueue = new ArrayBlockingQueue<Integer>(1, true);

(new Thread(new Producer())).start();

(new Thread(new Consumer())).start();

}

class Producer implements Runnable {

public void run() {

try {

int counter = 0;

for (int i = 0; i < myArray.length; i++) {

blockingQueue.put(myArray[i]);

if (counter++ < 2)

Thread.sleep(3000);

}

blockingQueue.put(-1);

} catch (InterruptedException e) {

System.err.println(e.getMessage());

}

}

}

class Consumer implements Runnable {

public void run() {

try {

Integer message = 0;

while (!((message = blockingQueue.take()).equals(-1)))

System.out.println(message);

} catch (InterruptedException e) {

System.err.println(e.getMessage());

}

}

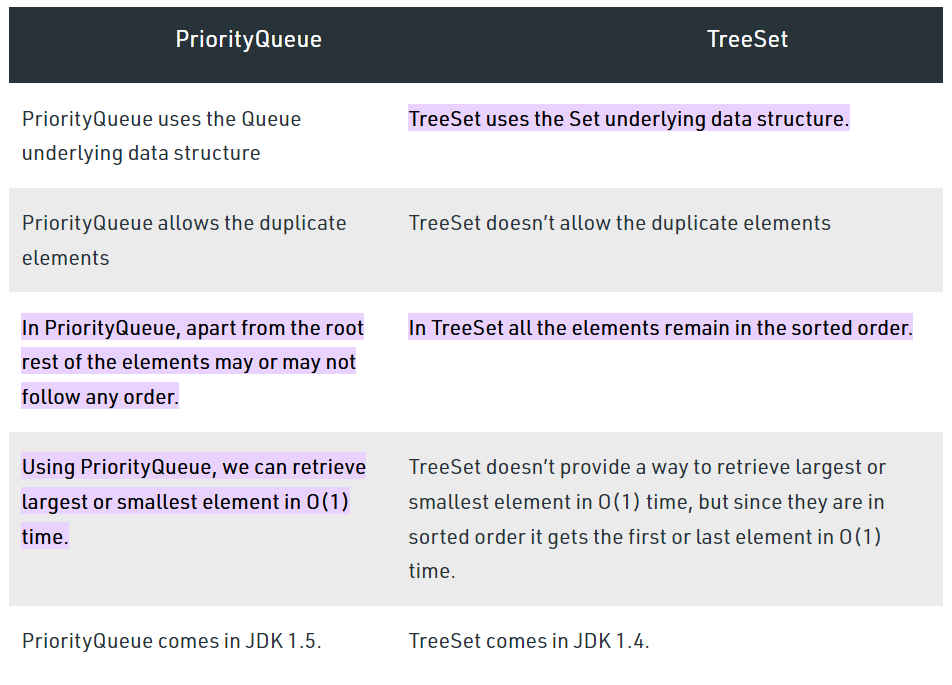
}

public static void main(String[] args) {

new ArrayBlockingQueueDemo();

}

}



**Conclusions**

* The Queue is used to insert elements at the end of the queue and removes from the beginning of the queue. It follows the FIFO concept.
* Java Queue is a part of the Collection Framework and implements Collection interface. So it supports all methods of Collection interface such as insertion, deletion and so on.
* The most frequently used implementations of Queue are LinkedList, ArrayBlockingQueue and PriorityQueue.
* The elements of the priority queue are ordered according to their natural ordering, or by a Comparator provided at queue construction time, depending on which constructor is used.
* If any null operation is performed on BlockingQueues, NullPointerException is thrown.