

Queue : – linked list
Priority queue

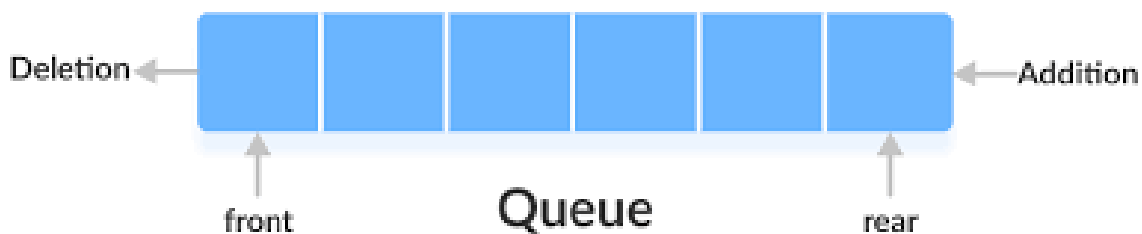
Deque -array deque
 -linked list

Thread safe :

blocking queue - array blocking queue
 Linked blocking queue
 Priority blocking queue
 Synchronous queue
 Delay queue

Conncorrent linked queue
Concurrent linked deque

FIRST IN FIRST OUT



Queue Implementation Using Java Collections

Java provides several built-in implementations of the Queue interface in its Collections Framework. Here's how you can use these implementations and also how to implement your own queue using a linked list approach similar to the Java Collections style.

Java's Built-in Queue Implementations

Java offers several Queue implementations in the `java.util` package:

1. **LinkedList** - Implements both List and Queue interfaces
2. **ArrayDeque** - Resizable-array implementation
3. **PriorityQueue** - Priority-based queue

Example using Java's LinkedList as Queue

java

Copy

Download

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

public class JavaCollectionsQueueExample {
    public static void main(String[] args) {
        // Create a queue using LinkedList
        Queue<Integer> queue = new LinkedList<>();

        // Enqueue elements
        queue.add(10);
        queue.offer(20); // alternative to add()
        queue.add(30);

        // Check queue status
        System.out.println("Queue: " + queue);
        System.out.println("Queue size: " + queue.size());
    }
}
```

```

System.out.println("Is queue empty? " + queue.isEmpty());
System.out.println("Front element: " + queue.peek());

// Dequeue elements
System.out.println("Dequeued: " + queue.remove()); //exception
System.out.println("Dequeued: " + queue.poll()); // alternative to
// remove() print null if no element

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

```

PriorityQueue vs LinkedList in Java

Does not allow null elements ,order elements according to natural order ,internal working **min heap** not sorted queue

```

package com.example.demo.Controller;
import java.util.*;
public class justChecking {
    public static void main(String[] args)
    {
        PriorityQueue<Integer> pq=new PriorityQueue<>();
        //PriorityQueue<Integer> pq=new PriorityQueue<>(Comperartor.reverseorder());
    }
}

```

```

        pq.add(15);
        pq.add(10);
        pq.add(30);
        pq.add(5);
        while (!pq.isEmpty())
        {
            System.out.println(pq.poll());
        }
    }
}

```

Output:

```

5
10
15
30

```

Key Differences Between PriorityQueue and LinkedList

Feature	PriorityQueue	LinkedList (as Queue)
Ordering	Elements ordered by natural ordering or Comparator	FIFO (insertion order)
Implementation	Typically implemented as a priority heap	Doubly-linked list
Null elements	Not allowed	Allowed
Thread safety	Not thread-safe	Not thread-safe
Performance	$O(\log n)$ for insert/remove	$O(1)$ for insert/remove at ends

Use cases

When priority ordering is needed

When FIFO behavior is
needed

PriorityQueue Methods with Examples

Here's a complete example demonstrating all the main methods of PriorityQueue:

java

Copy

Download

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

public class PriorityQueueExample {
    public static void main(String[] args) {
        // 1. Creating a PriorityQueue
        // Natural ordering (smallest first)
        PriorityQueue<Integer> pq = new PriorityQueue<>();

        // With custom comparator (largest first)
        PriorityQueue<Integer> maxHeap = new
PriorityQueue<>(Comparator.reverseOrder());

        // 2. Adding elements - add() vs offer()
        pq.add(30);    // throws exception if capacity restricted
        pq.offer(10);  // returns false if capacity restricted
        pq.add(20);
        pq.offer(50);
        pq.add(40);

        System.out.println("PriorityQueue: " + pq); // Order in storage may not
reflect priority order

        // 3. Examining elements - element() vs peek()
        System.out.println("Head using element(): " + pq.element()); // throws
NoSuchElementException if empty
```

```

        System.out.println("Head using peek(): " + pq.peek());           // returns
null if empty

        // 4. Removing elements - remove() vs poll()
        System.out.println("Removed using remove(): " + pq.remove()); // throws
NoSuchElementException if empty
        System.out.println("Removed using poll(): " + pq.poll());       // returns
null if empty

        System.out.println("Queue after removals: " + pq);

        // 5. Other utility methods
        System.out.println("Size: " + pq.size());
        System.out.println("Contains 30? " + pq.contains(30));

        // 6. Clearing the queue
        pq.clear();
        System.out.println("Is empty after clear? " + pq.isEmpty());

        // 7. Custom object example
        PriorityQueue<Student> studentQueue = new PriorityQueue<>(
Comparator.comparing(Student::getGrade).thenComparing(Student::getName)
);

        studentQueue.add(new Student("Alice", 85));
        studentQueue.add(new Student("Bob", 92));
        studentQueue.add(new Student("Charlie", 85));
        studentQueue.add(new Student("David", 78));

        System.out.println("\nStudents in priority order:");
        while (!studentQueue.isEmpty()) {
            System.out.println(studentQueue.poll());
        }
    }

    static class Student {
        private String name;
        private int grade;

        public Student(String name, int grade) {
            this.name = name;
            this.grade = grade;
        }
    }

```

```

    public String getName() { return name; }
    public int getGrade() { return grade; }

    @Override
    public String toString() {
        return name + " (" + grade + ")";
    }
}
}

```

Complete List of PriorityQueue Methods

1. Adding Elements

- `boolean add(E e)`: Inserts the element, throws exception if capacity restricted
- `boolean offer(E e)`: Inserts the element, returns false if capacity restricted

2. Removing Elements

- `E remove()`: Retrieves and removes the head, throws exception if empty
- `E poll()`: Retrieves and removes the head, returns null if empty

3. Examining Elements

- `E element()`: Retrieves but doesn't remove the head, throws exception if empty
- `E peek()`: Retrieves but doesn't remove the head, returns null if empty

4. Utility Methods

- `int size()`: Returns the number of elements
- `boolean isEmpty()`: Returns true if the queue is empty
- `boolean contains(Object o)`: Returns true if the queue contains the element
- `void clear()`: Removes all elements
- `Object[] toArray()`: Returns an array containing all elements
- `<T> T[] toArray(T[] a)`: Returns an array containing all elements (typed)
- `Iterator<E> iterator()`: Returns an iterator (note: iteration order is not guaranteed to be priority order)

5. Bulk Operations

- `boolean remove(Object o)`: Removes a single instance of the element
- `boolean removeAll(Collection<?> c)`: Removes all elements in the collection

- `boolean retainAll(Collection<?> c)`: Retains only elements in the collection
- `boolean addAll(Collection<? extends E> c)`: Adds all elements from the collection

6. Java 8+ Methods

- `boolean removeIf(Predicate<? super E> filter)`: Removes all elements matching the predicate
- `Splitter<E> splitter()`: Creates a late-binding and fail-fast splitter

Important Notes About PriorityQueue

1. The iterator returned by `iterator()` does not guarantee to traverse elements in priority order. For ordered traversal, use `poll()` in a loop.
2. PriorityQueue doesn't permit null elements.
3. The head of the queue is the least element (for natural ordering) or as determined by the Comparator.
4. Implementation is not synchronized. For thread-safe version, use:

5. [java](#)
6. [Copy](#)
7. [Download](#)

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

8. `PriorityQueue<E> threadSafePq = new PriorityQueue<>(Collections.synchronizedCollection(pq));`
9. The enqueueing and dequeuing operations (offer, poll, remove, add) run in $O(\log n)$ time.

what is the use of priority queue

The Use of Priority Queues

A priority queue is a specialized data structure that differs from a regular queue by ordering elements based on their priority rather than just following FIFO (First-In-First-Out) order. Here are the key uses and applications of priority queues:

Main Uses of Priority Queues

1. Task Scheduling

Priority queues are extensively used in operating systems for:

- CPU task scheduling (higher priority tasks get executed first)
- Load balancing in distributed systems
- Printer job scheduling (urgent print jobs get priority)

2. Graph Algorithms

Essential for efficient implementation of:

- Dijkstra's algorithm (finding shortest paths)
- Prim's algorithm (minimum spanning trees)
- A* search algorithm (pathfinding with heuristics)

3. Data Compression

Used in Huffman coding for:

- Building optimal prefix codes
- Selecting least frequent elements first during tree construction

4. Discrete Event Simulation

For managing events where:

- Events are processed based on scheduled time
- Higher priority given to earlier events

5. Bandwidth Management

In networking for:

- Quality of Service (QoS) implementations
- Prioritizing certain types of network traffic

Practical Examples

Example 1: Emergency Room Triage System

java

Copy

Download

```
PriorityQueue<Patient> emergencyRoom = new PriorityQueue<>(
    Comparator.comparingInt(Patient::getSeverity).reversed()
);

emergencyRoom.add(new Patient("John", 3)); // Mild case
emergencyRoom.add(new Patient("Sarah", 1)); // Critical
emergencyRoom.add(new Patient("Mike", 2)); // Serious

while (!emergencyRoom.isEmpty()) {
    System.out.println("Treating: " + emergencyRoom.poll());
    // Sarah (1) treated first, then Mike (2), then John (3)
}
```

Example 2: Job Scheduling System

java

Copy

Download

```
PriorityQueue<Job> scheduler = new PriorityQueue<>(
    Comparator.comparing(Job::getPriority)
                .thenComparing(Job::getSubmissionTime)
);

scheduler.add(new Job("Backup", 2, LocalDateTime.of(10, 0)));
scheduler.add(new Job("Report", 1, LocalDateTime.of(9, 30)));
scheduler.add(new Job("Update", 1, LocalDateTime.of(9, 0)));

// Jobs will execute in order: Update, Report, Backup
```

Key Characteristics

1. **Dynamic Prioritization:** Elements can be added in any order but are always removed according to priority
2. **Efficient Operations:**
 - Insertion: $O(\log n)$ time
 - Removal of highest priority: $O(\log n)$ time
 - Peek at highest priority: $O(1)$ time
3. **Implementation Choices:**
 - Typically implemented as a binary heap
 - Can also be implemented with balanced trees or other structures

When to Use Priority Queue vs Regular Queue

Use Priority Queue when:

- Processing order depends on element attributes/priority
- You need to frequently access/remove the "most important" item
- The priority of elements might change dynamically

Use Regular Queue when:

- You strictly need FIFO behavior
- The ordering is purely based on arrival time
- You don't need to evaluate elements for priority

Advanced Applications

1. **Merge K Sorted Lists:** Efficiently merge multiple sorted inputs
2. **Median Maintenance:** Track running median of a data stream
3. **Stock Order Matching:** Process buy/sell orders by price-time priority
4. **AI Algorithms:** Best-first search, beam search algorithms
5. **Discrete Optimization:** Branch and bound algorithms

Priority queues are fundamental to many efficient algorithms and system designs where prioritization is more important than simple arrival order.

Min-Heap Explained

A **min-heap** is a specialized **binary tree-based data structure** where the smallest element is always at the root, and every parent node has a value **less than or equal** to its children. It is a type of **priority queue** that allows efficient access to the minimum element.

Key Properties of a Min-Heap

- 1. **Complete Binary Tree:** All levels are fully filled except possibly the last level, which is filled from left to right.
 - 2. **Heap Property:** Every parent node has a value \leq its children.
 - 3. **Root is Minimum:** The smallest element is always at the top.
 - 4. **Efficient Operations:**
 - **Insertion** ($O(\log n)$)
 - **Extract-Min** ($O(\log n)$)
 - **Peek-Min** ($O(1)$)
-

Min-Heap vs. Max-Heap

Feature	Min-Heap	Max-Heap
Root Value	Smallest element	Largest element
Ordering	Parent \leq Children	Parent \geq Children
Use Case	Finding min, Dijkstra’s algorithm	Finding max, heap sort (descending)

Min-Heap Operations with Examples

1. Insertion (Heapify-Up)

- New elements are added at the next available position.
- The heap property is restored by swapping the new element with its parent until the parent is smaller.

Example: Insert 3 into [5, 10, 12]

1. Insert 3 at the end → [5, 10, 12, 3]
2. Compare 3 with parent 12 → swap → [5, 10, 3, 12]
3. Compare 3 with parent 5 → swap → [3, 10, 5, 12] (Heap restored)

2. Extract-Min (Heapify-Down)

- The root (minimum) is removed.
- The last element replaces the root.
- The heap property is restored by swapping with the smallest child.

Example: Extract min from [3, 10, 5, 12]

1. Remove 3 → [12, 10, 5] (last element 12 moves to root)
2. Compare 12 with children (10, 5) → swap with 5 → [5, 10, 12]

3. Peek-Min (Get Minimum)

- Returns the root without removing it.
- **Example:** `peek()` on [5, 10, 12] → returns 5.

Min-Heap Implementation in Java

java

Copy

Download

```
import java.util.PriorityQueue;

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

```

// By default, PriorityQueue is a min-heap
PriorityQueue<Integer> minHeap = new PriorityQueue<>();

// Insert elements
minHeap.add(10);
minHeap.add(5);
minHeap.add(15);
minHeap.add(3);

System.out.println("Min-Heap: " + minHeap); // [3, 5, 15, 10]

// Peek the minimum element
System.out.println("Minimum element: " + minHeap.peek()); // 3

// Extract the minimum element
System.out.println("Removed: " + minHeap.poll()); // 3
System.out.println("Heap after removal: " + minHeap); // [5, 10, 15]
}
}

```

Applications of Min-Heap

1. **Priority Scheduling** (OS task scheduling)
2. **Dijkstra's Algorithm** (finding shortest paths)
3. **Merge K Sorted Lists** (efficient merging)
4. **Huffman Coding** (data compression)
5. **Finding Kth Smallest Element** (using heap selection)

Time Complexity Summary

Operation	Time Complexity
Insert	$O(\log n)$

Extract-Min	$O(\log n)$
Peek-Min	$O(1)$
Heapify (build heap)	$O(n)$

Summary

- **Min-heap** ensures the smallest element is always at the top.
- **Used in priority queues, graph algorithms, and sorting.**
- **Java's `PriorityQueue` is a min-heap by default.**
- **Efficient for dynamic data where the minimum is frequently accessed.**

Would you like a custom implementation (without `PriorityQueue`) for better understanding? 🚀

Deque (Double-Ended Queue) Explained

A **Deque (Double-Ended Queue)** is a linear data structure that allows insertion and deletion of elements from **both ends (front and rear)**. Unlike a regular queue (FIFO) or stack (LIFO), a deque provides more flexibility.

Key Features of Deque

1. **Insert/Remove from Both Ends:**
 - `addFirst()`, `removeFirst()` (Front operations)
 - `addLast()`, `removeLast()` (Rear operations)
2. **Can Be Used as Both Queue and Stack:**
 - **Queue Mode:** `offerFirst()` + `pollLast()` (FIFO)

- **Stack Mode:** `push()` (`addFirst`) + `pop()` (`removeFirst`) (LIFO)
3. **Thread-Safe?** `ArrayDeque` is **not thread-safe**, but `LinkedBlockingDeque` is.

Deque Implementations in Java

1. `ArrayDeque` (Resizable array, faster for most operations)
 2. `LinkedList` (Doubly-linked list, slower but more flexible)
-

ArrayDeque Example (All Methods)

java

Copy

Download

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

public class DequeExample {
    public static void main(String[] args) {
        Deque<Integer> deque = new ArrayDeque<>();

        // 1. Insertion at Front (Stack-like)
        deque.addFirst(10); // Throws exception if full
        deque.offerFirst(20); // Returns false if full
        deque.push(30); // Same as addFirst (Stack method)
        System.out.println("After Front Insertions: " + deque); // [30, 20, 10]

        // 2. Insertion at Rear (Queue-like)
        deque.addLast(40); // Throws exception if full
        deque.offerLast(50); // Returns false if full
        System.out.println("After Rear Insertions: " + deque); // [30, 20, 10, 40,
50]

        // 3. Peek (Check without removing)
        System.out.println("Peek First: " + deque.peekFirst()); // 30
        System.out.println("Peek Last: " + deque.peekLast()); // 50
        System.out.println("Peek (Same as peekFirst): " + deque.peek()); // 30
```



```

// 4. Removal from Front (Stack-like)
System.out.println("Removed First: " + deque.removeFirst()); // 30 (throws
if empty)
System.out.println("Poll First: " + deque.pollFirst()); // 20
(return null if empty)
System.out.println("Pop (Same as removeFirst): " + deque.pop()); // 10

// 5. Removal from Rear (Queue-like)
System.out.println("Removed Last: " + deque.removeLast()); // 50
System.out.println("Poll Last: " + deque.pollLast()); // 40

// 6. Check if empty
System.out.println("Is Deque empty? " + deque.isEmpty()); // true
}
}

```

Output:

Copy

Download

```

After Front Insertions: [30, 20, 10]
After Rear Insertions: [30, 20, 10, 40, 50]
Peek First: 30
Peek Last: 50
Peek (Same as peekFirst): 30
Removed First: 30
Poll First: 20
Pop (Same as removeFirst): 10
Removed Last: 50
Poll Last: 40
Is Deque empty? true

```

Differences Between Deque, LinkedList, and Stack

Feature	Deque (ArrayDeque)	LinkedList	Stack (Legacy)
---------	--------------------	------------	----------------

Structure	Resizable array	Doubly-linked list	Extends <code>Vector</code> (array-based)
Thread-Safe?	❌ No (Use <code>LinkedBlockingDeque</code>)	❌ No	✅ Yes (but slow)
Performance	Faster for most ops (O(1))	Slower due to pointers	Slow (legacy, avoid)
Flexibility	Front + rear ops	Front + rear ops + list ops	Only LIFO (Stack)
Null Allowed?	❌ No	✅ Yes	✅ Yes
Recommended?	✅ Yes (Best for Deque)	✅ Yes (If list features needed)	❌ No (Deprecated)

If normal use use `arraydeque`

If more deletion and insertion use linked list for deque

When to Use Which?

- Use** `ArrayDeque`:
 - When you need a **fast, memory-efficient Deque** (queue + stack).
 - Example: BFS in graphs, sliding window problems.
- Use** `LinkedList`:
 - When you need **both Deque + List operations** (e.g., `get(index)`).
 - Example: LRU Cache implementation.
- Avoid** `Stack`:
 - It's **legacy** and **synchronized (slow)**. Use `ArrayDeque` instead for LIFO.

Summary

- **Deque** allows **insertion/deletion from both ends**.
- `ArrayDeque` is **faster** and **preferred** for Deque operations.
- `LinkedList` is **more flexible** but **slower**.
- `Stack` **is outdated**—use `ArrayDeque` for stack operations.

Would you like a real-world use case (e.g., **undo-redo system** using Deque)?

BLOCKING QUEUE:

Thread safe ,wait for queue to became non empty /wait for space

Simplify concurrency problems like producer -consumer

Standard queue immediately

Empty -remove (no waiting),full – add (no waiting)

BlockingQueue in Java - Complete Guide

A **BlockingQueue** is a thread-safe queue that provides **blocking operations** when:

- The queue is **full** (blocks on insertion)
- The queue is **empty** (blocks on removal)

It is part of the `java.util.concurrent` package and is widely used in

producer-consumer scenarios.

Key Features of BlockingQueue

1. **Thread-Safe:** Designed for multi-threaded environments.
2. **Blocking Operations:**
 - Blocks **producer threads** when the queue is full.
 - Blocks **consumer threads** when the queue is empty.

3. Bounded/Unbounded:

- Can have a fixed capacity (`ArrayBlockingQueue`).
- Can be unbounded (`LinkedBlockingQueue` without capacity).

4. Fairness Policy (optional):

- Ensures **FIFO** access for blocked threads.

Common Implementations

Implementation	Description
<code>ArrayBlockingQueue</code>	Fixed-size, backed by an array
<code>LinkedBlockingQueue</code>	Optional bounded, linked nodes
<code>PriorityBlockingQueue</code>	Priority-based, unbounded
<code>SynchronousQueue</code>	No capacity (direct handoff)
<code>DelayQueue</code>	Elements with delay times

BlockingQueue Methods (All Operations)

1. Insertion Methods

Method	Behavior
<code>add(E e)</code>	Throws <code>IllegalStateException</code> if full

<code>offer(E e)</code>	Returns <code>false</code> if full
<code>put(E e)</code>	Blocks until space is available
<code>offer(E e, long timeout, TimeUnit unit)</code>	Waits for <code>timeout</code> before giving up

2. Removal Methods

Method	Behavior
<code>remove()</code>	Throws <code>NoSuchElementException</code> if empty
<code>poll()</code>	Returns <code>null</code> if empty
<code>take()</code>	Blocks until an element is available
<code>poll(long timeout, TimeUnit unit)</code>	Waits for <code>timeout</code> before giving up

3. Inspection Methods

Method	Behavior
<code>element()</code>	Throws <code>NoSuchElementException</code> if empty
<code>peek()</code>	Returns <code>null</code> if empty

4. Bulk Operations

Method	Behavior
--------	----------

<code>drainTo(Collection<? super E> c)</code>	Removes all available elements into <code>c</code>
---	--

<code>drainTo(Collection<? super E> c, int maxElements)</code>	Removes at most <code>maxElements</code> into <code>c</code>
--	--

5. Capacity & Utility Methods

Method	Behavior
<code>remainingCapacity()</code>	Returns available slots
<code>size()</code>	Current number of elements
<code>isEmpty()</code>	Checks if empty
<code>contains(Object o)</code>	Checks if element exists

Example: Producer-Consumer Using `ArrayBlockingQueue`

java

Copy

Download

```
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;

public class BlockingQueueExample {
    public static void main(String[] args) {
        BlockingQueue<Integer> queue = new ArrayBlockingQueue<>(3); // Capacity =
3 here size is fixed no change in size
```

```

// Producer Thread
new Thread(() -> {
    try {
        queue.put(1); // Inserts 1 (blocks if full)
        System.out.println("Added 1");
        queue.put(2);
        System.out.println("Added 2");
        queue.put(3);
        System.out.println("Added 3");
        queue.put(4); // Blocks until space is available
        System.out.println("Added 4");
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }
}).start();

// Consumer Thread (removes after 2 seconds)
new Thread(() -> {
    try {
        Thread.sleep(2000);
        System.out.println("Removed: " + queue.take()); // Removes 1
        Thread.sleep(1000);
        System.out.println("Removed: " + queue.take()); // Removes 2
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }
}).start();
}
}

```

Output:

Copy

Download

```

Added 1
Added 2
Added 3
(After 2 sec) Removed: 1
Added 4
(After 1 more sec) Removed: 2

```

When to Use BlockingQueue?

- ✓ **Producer-Consumer Problems** (Thread-safe data exchange)
- ✓ **Thread Pool Management** (e.g., `ThreadPoolExecutor` uses a `BlockingQueue`)
- ✓ **Task Scheduling** (Background jobs with rate-limiting)
- ✓ **Buffered Data Processing** (E.g., batch processing)

Comparison: `BlockingQueue` vs `Queue`

Feature	<code>BlockingQueue</code>	<code>Queue</code>
Thread-Safety	✓ Yes	✗ No (except <code>ConcurrentLinkedQueue</code>)
Blocking Operations	✓ <code>put()</code> , <code>take()</code>	✗ No
Bounded Capacity	✓ Supported	✗ No (unless manually checked)
Use Case	Multi-threaded sync	Single-threaded / manual sync

Best Practices

1. **Prefer** `put()` **and** `take()` for true blocking behavior.
 2. **Use bounded queues** to prevent memory issues.
 3. **Avoid** `poll()` **with long timeouts** if immediate response is needed.
 4. **Consider** `PriorityBlockingQueue` if ordering matters.
-

Summary

- **BlockingQueue** is **thread-safe** and supports **blocking operations**.
- `put()` blocks if full, `take()` blocks if empty.
- **Used in multi-threading** (producer-consumer, thread pools).
- **Best implementations:** `ArrayBlockingQueue`, `LinkedBlockingQueue`.

Would you like a real-world example (e.g., **ThreadPool with BlockingQueue**)? 🚀

LinkedBlockingQueue in Java - Complete Guide

`LinkedBlockingQueue` is a thread-safe, optionally bounded blocking queue implementation in Java's `java.util.concurrent` package. It's based on linked nodes and is ideal for producer-consumer scenarios.

Key Features

- **Thread-safe** for concurrent access
- **Optional capacity bound** (unbounded by default)
- **FIFO ordering**
- **Higher throughput** than `ArrayBlockingQueue` for most use cases
- **Non-fair** access policy by default (no thread ordering guarantees)

All Methods of LinkedBlockingQueue

1. Constructors

java

Copy

Download

```
// Unbounded queue
LinkedBlockingQueue<Integer> unbounded = new LinkedBlockingQueue<>();

// Bounded queue with capacity
LinkedBlockingQueue<Integer> bounded = new LinkedBlockingQueue<>(100);

// Create from existing collection
```

```
LinkedBlockingQueue<Integer> fromCollection = new  
LinkedBlockingQueue<>(Arrays.asList(1, 2, 3));
```

2. Insertion Methods

Method	Behavior	Throws Exception?	Blocks?	Special Cases
<code>add(E e)</code>	Adds if space available	Yes (IllegalStateException)	No	Returns true if successful
<code>offer(E e)</code>	Adds if space available	No	No	Returns false if full
<code>put(E e)</code>	Adds element	No	Yes (if full)	Waits indefinitely
<code>offer(E e, long timeout, TimeUnit unit)</code>	Adds with timeout	No	Yes (temporarily)	Returns false if timeout

Example:

java

Copy

Download

```
LinkedBlockingQueue<String> queue = new LinkedBlockingQueue<>(2);
```

```
queue.add("A"); // succeeds
```

```
queue.offer("B"); // succeeds
// queue.add("C"); // throws IllegalStateException
queue.put("C"); // blocks until space available (in another thread)
```

3. Removal Methods

Method	Behavior	Throws Exception?	Blocks?	Special Cases
<code>remove()</code>	Removes head	Yes (NoSuchElementException)	No	Returns element
<code>poll()</code>	Removes head	No	No	Returns null if empty
<code>take()</code>	Removes head	No	Yes (if empty)	Waits indefinitely
<code>poll(long timeout, TimeUnit unit)</code>	Removes with timeout	No	Yes (tempor arily)	Returns null if timeout

Example:

java

Copy

Download

```
LinkedListBlockingQueue<String> queue = new LinkedListBlockingQueue<>(Arrays.asList("X",  
"Y"));
```

```
String first = queue.remove(); // "X"
```

```
String second = queue.poll(); // "Y"
// String third = queue.remove(); // throws NoSuchElementException
String maybeNull = queue.poll(); // returns null
```

4. Inspection Methods

Method	Behavior	Throws Exception?	Special Cases
<code>element()</code>	Returns head	Yes (NoSuchElementException)	Doesn't remove
<code>peek()</code>	Returns head	No	Returns null if empty

Example:

java

Copy

Download

```
LinkedBlockingQueue<Integer> queue = new LinkedBlockingQueue<>(List.of(10, 20));

int head = queue.element(); // 10 (queue unchanged)
Integer maybeHead = queue.peek(); // 10 (queue unchanged)
```

5. Bulk Operations

Method	Description
<code>drainTo(Collection<? super E> c)</code>	Removes all available elements

```
drainTo(Collection<? super E> c, int  
maxElements)
```

Removes up to max elements

Example:

java

Copy

Download

```
LinkedBlockingQueue<Integer> queue = new LinkedBlockingQueue<>(List.of(1, 2, 3,  
4));  
List<Integer> list = new ArrayList<>();
```

```
queue.drainTo(list); // list = [1, 2, 3, 4], queue is empty
```

6. Utility Methods

Method	Description
<code>size()</code>	Current element count
<code>remainingCapacity()</code>	Returns available slots
<code>contains(Object o)</code>	Checks for element
<code>toArray()</code>	Returns array copy
<code>clear()</code>	Removes all elements

Example:

java

Copy

Download

```
LinkedBlockingQueue<String> queue = new LinkedBlockingQueue<>(3);
queue.add("A");
queue.add("B");

int size = queue.size(); // 2
int remaining = queue.remainingCapacity(); // 1
boolean hasA = queue.contains("A"); // true
```

Complete Example: Producer-Consumer

java

Copy

Download

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

public class LinkedBlockingQueueDemo {
    public static void main(String[] args) throws InterruptedException {
        LinkedBlockingQueue<String> queue = new LinkedBlockingQueue<>(2);

        // Producer
        new Thread(() -> {
            try {
                queue.put("Message 1");
                queue.put("Message 2");
                System.out.println("Attempting to put Message 3...");
                queue.put("Message 3"); // Will block until space available
                System.out.println("Message 3 added");
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }).start();

        // Consumer
        new Thread(() -> {
            try {
                Thread.sleep(2000); // Simulate processing delay
                System.out.println("Received: " + queue.take());
            }
        }).start();
    }
}
```

```

        Thread.sleep(1000);
        System.out.println("Received: " + queue.take());
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }
}).start();
}
}

```

Output:

Copy

Download

Attempting to put Message 3...

Received: Message 1

Received: Message 2

Message 3 added

When to Use LinkedBlockingQueue

- **Producer-consumer patterns** with multiple threads
- **Work queues** in thread pools
- **Bounded buffers** for data pipelines
- When you need **higher throughput** than `ArrayBlockingQueue`

Performance Characteristics

- **Insertion/removal:** $O(1)$ average case
- **Concurrency:** High throughput due to separate locks for head and tail
- **Memory:** Slightly more overhead than array-based queues due to node links

Comparison with ArrayBlockingQueue

Feature	LinkedBlockingQueue	ArrayBlockingQueue
Structure	Linked nodes	Circular array

Default Capacity	Integer.MAX_VALUE	Must be specified
Locking	Two locks (put/take)	Single lock
Throughput	Generally higher	Lower under contention
Memory Use	Higher (node objects)	Lower

Use `LinkedBlockingQueue` when you need high throughput or an optionally bounded queue. Use `ArrayBlockingQueue` when you need predictable iteration or slightly lower memory overhead.

PriorityBlockingQueue in Java - Complete Guide

`PriorityBlockingQueue` is a thread-safe, unbounded blocking queue that orders elements according to their natural ordering or by a `Comparator` provided at construction time.

Key Features

- **Unbounded capacity** (grows as needed)
- **Thread-safe** for concurrent access
- **Priority-based ordering** (not FIFO)
- **Blocks on empty** (but not on full, since it's unbounded)
- **Implements** `BlockingQueue` interface

Comparison with Other Blocking Queues

Feature	PriorityBlockingQueue	ArrayBlockingQueue	LinkedBlockingQueue
Bounded/Unbounded	Unbounded	Bounded	Optionally bounded
Ordering	Priority-based	FIFO	FIFO
Blocking on Full	No (unbounded)	Yes	Yes (if bounded)
Blocking on Empty	Yes	Yes	Yes
Thread Safety	Yes	Yes	Yes
Implementation	Priority heap	Circular array	Linked nodes
Null Elements	Not allowed	Not allowed	Not allowed

All Methods with Examples

1. Constructors

java

Copy

Download

```
// Natural ordering
PriorityBlockingQueue<Integer> pq1 = new PriorityBlockingQueue<>();

// With initial capacity
PriorityBlockingQueue<Integer> pq2 = new PriorityBlockingQueue<>(10);

// With custom comparator
```

```
PriorityBlockingQueue<Integer> pq3 = new PriorityBlockingQueue<>(10,  
    Comparator.reverseOrder());
```

2. Insertion Methods

java

Copy

Download

```
PriorityBlockingQueue<Integer> pq = new PriorityBlockingQueue<>();  
  
pq.add(30);    // Throws exception if capacity restricted (but it's unbounded)  
pq.offer(10);  // Always returns true (since unbounded)  
pq.put(20);    // Same as offer() for unbounded queue  
pq.offer(50, 1, TimeUnit.SECONDS); // Timeout irrelevant for unbounded queue  
  
System.out.println(pq); // Order may not reflect priority in toString()
```

3. Removal Methods

java

Copy

Download

```
System.out.println(pq.poll());    // 10 (returns null if empty)  
System.out.println(pq.remove());  // 20 (throws exception if empty)  
System.out.println(pq.take());    // 30 (blocks until element available)  
System.out.println(pq.poll(1, TimeUnit.SECONDS)); // 50 (or null if timeout)
```

4. Inspection Methods

java

Copy

Download

```
System.out.println(pq.peek());    // Returns head without removal (null if empty)  
System.out.println(pq.element()); // Throws exception if empty
```

5. Bulk Operations

java

Copy

Download

```
ArrayList<Integer> list = new ArrayList<>();
pq.drainTo(list); // Moves all elements to list
System.out.println(list); // [10, 20, 30, 50] (in priority order)
```

Complete Example: Task Scheduling System

java

Copy

Download

```
import java.util.concurrent.PriorityBlockingQueue;
import java.util.concurrent.TimeUnit;

class Task implements Comparable<Task> {
    String name;
    int priority;

    Task(String name, int priority) {
        this.name = name;
        this.priority = priority;
    }

    @Override
    public int compareTo(Task other) {
        return Integer.compare(this.priority, other.priority);
    }

    @Override
    public String toString() {
        return name + " (Priority: " + priority + ")";
    }
}

public class PriorityBlockingQueueDemo {
    public static void main(String[] args) throws InterruptedException {
        PriorityBlockingQueue<Task> taskQueue = new PriorityBlockingQueue<>();

        // Producer Thread
```

```

new Thread(() -> {
    taskQueue.put(new Task("Email Server", 3));
    taskQueue.put(new Task("System Alarm", 1));
    taskQueue.put(new Task("User Request", 5));
    taskQueue.put(new Task("Admin Task", 2));
}).start();

// Consumer Thread
new Thread(() -> {
    try {
        while (true) {
            Task task = taskQueue.take(); // Blocks if empty
            System.out.println("Processing: " + task);
            Thread.sleep(1000); // Simulate work
        }
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }
}).start();
}
}

```

Output:

Copy

Download

```

Processing: System Alarm (Priority: 1)
Processing: Admin Task (Priority: 2)
Processing: Email Server (Priority: 3)
Processing: User Request (Priority: 5)

```

When to Use PriorityQueue

- **Task scheduling** where priority matters
- **Emergency handling systems** (high priority first)
- **Custom ordering** requirements in concurrent systems
- **Unbounded queues** where memory isn't a constraint

Performance Characteristics

- **Insertion/removal:** $O(\log n)$ time due to heap maintenance
- **Concurrency:** Single lock for all operations
- **Memory:** Grows dynamically as needed

Key Differences from Other Blocking Queues

1. **vs ArrayBlockingQueue:**
 - Priority ordering vs strict FIFO
 - Unbounded vs fixed capacity
 - Heap structure vs array structure
2. **vs LinkedBlockingQueue:**
 - Priority-based vs FIFO
 - Always unbounded vs optionally bounded
 - Slower insertions/removals ($O(\log n)$ vs $O(1)$)
3. **vs PriorityQueue:**
 - Thread-safe vs non-thread-safe
 - Blocking operations vs non-blocking
 - Identical ordering behavior

Best Practices

1. **Always provide a Comparator** if natural ordering isn't appropriate
2. **Beware of OOM errors** since it's unbounded
3. **Use proper task objects** (implement `Comparable`)
4. **Consider wrapping with SynchronizedQueue** if you need fairness

Would you like a real-world example like a **hospital emergency room system** using `PriorityBlockingQueue`? 🚑

SynchronousQueue in Java - Complete Guide

`SynchronousQueue` is a special blocking queue where each insert operation must wait for a corresponding remove operation by another thread, and vice versa. It's essentially a queue with **zero capacity**.

Key Features

- **Zero capacity:** Cannot hold any elements
- **Handoff mechanism:** Direct transfer from producer to consumer
- **Thread-safe:** Designed for concurrent use
- **Two fairness policies:**
 - **Non-fair mode** (default): No ordering guarantees
 - **Fair mode:** FIFO ordering for waiting threads

When to Use SynchronousQueue

- **Direct handoff** between threads
- **Thread pool work queues** (`Executors.newCachedThreadPool()` uses it)
- **High-performance pipelines** where buffering isn't needed
- **Load balancing** with immediate transfer

Comparison with Other Queues

Feature	SynchronousQueue	ArrayBlockingQueue	LinkedBlockingQueue
Capacity	0	Fixed	Configurable
Blocking on Insert	Always	Only when full	Only when full
Blocking on Remove	Always	Only when empty	Only when empty
Fairness Option	Yes	Yes	No
Best For	Direct handoffs	Fixed-size buffers	General-purpose

All Methods with Examples

1. Constructors

java

Copy

Download

```
// Non-fair (default)
SynchronousQueue<Integer> nonFairQueue = new SynchronousQueue<>();

// Fair mode (FIFO ordering)
SynchronousQueue<Integer> fairQueue = new SynchronousQueue<>(true);
```

2. Insertion Methods

java

Copy

Download

```
SynchronousQueue<String> queue = new SynchronousQueue<>();

// Producer Thread
new Thread(() -> {
    try {
        System.out.println("Attempting to put 'Data'");
        queue.put("Data"); // Blocks until consumer takes
        System.out.println("Successfully transferred 'Data'");
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }
}).start();

// Consumer Thread (starts after delay)
new Thread(() -> {
    try {
        Thread.sleep(2000); // Simulate processing delay
        String data = queue.take();
        System.out.println("Received: " + data);
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }
}).start();
```

Output:

Copy

Download

Attempting to put 'Data'

(2 second delay)

Received: Data

Successfully transferred 'Data'

3. Removal Methods

java

Copy

Download

```
SynchronousQueue<Integer> queue = new SynchronousQueue<>();
```

```
// Consumer Thread (starts first)
```

```
new Thread(() -> {  
    try {  
        System.out.println("Waiting to receive...");  
        int num = queue.take(); // Blocks until producer puts  
        System.out.println("Got: " + num);  
    } catch (InterruptedException e) {  
        Thread.currentThread().interrupt();  
    }  
}).start();
```

```
// Producer Thread
```

```
new Thread(() -> {  
    try {  
        Thread.sleep(1000);  
        queue.put(42); // Will be immediately received  
    } catch (InterruptedException e) {  
        Thread.currentThread().interrupt();  
    }  
}).start();
```

4. Non-Blocking Methods

java

Copy

Download

```
SynchronousQueue<String> queue = new SynchronousQueue<>();

System.out.println(queue.offer("Try")); // false (no waiting consumer)
System.out.println(queue.poll()); // null (no waiting producer)

// With timeout
System.out.println(queue.offer("Try", 1, TimeUnit.SECONDS)); // false after 1 sec
System.out.println(queue.poll(1, TimeUnit.SECONDS)); // null after 1 sec
```

5. Inspection Methods

java

Copy

Download

```
System.out.println(queue.isEmpty()); // Always true!
System.out.println(queue.size());    // Always 0!
System.out.println(queue.remainingCapacity()); // Always 0!
```

Real-World Example: Cached Thread Pool

java

Copy

Download

```
import java.util.concurrent.*;

public class CachedThreadPoolDemo {
    public static void main(String[] args) {
        // Internally uses SynchronousQueue
        ExecutorService executor = Executors.newCachedThreadPool();

        executor.submit(() -> System.out.println("Task 1"));
        executor.submit(() -> System.out.println("Task 2"));

        executor.shutdown();
    }
}
```

Performance Characteristics

- **Throughput:** Higher than buffered queues for handoff scenarios
- **Latency:** Lower because no buffering overhead
- **Memory:** Minimal (no storage overhead)

When Not to Use

- ❌ When buffering is needed between producers/consumers
- ❌ When producers and consumers may operate at different rates
- ❌ When you need to inspect queue contents

Advanced Use Case: Work Stealing

java

Copy

Download


```
// Create multiple worker threads sharing a SynchronousQueue
SynchronousQueue<Runnable> workQueue = new SynchronousQueue<>();

for (int i = 0; i < 4; i++) {
    new Thread(() -> {
        while (true) {
            try {
                Runnable task = workQueue.take();
                task.run();
            } catch (InterruptedException e) {
                break;
            }
        }
    }).start();
}

// Submit tasks
workQueue.put(() -> System.out.println("Task A"));
workQueue.put(() -> System.out.println("Task B"));
```

Key Takeaways

1. **Zero-capacity** blocking queue
2. **Direct handoff** between threads
3. **Two modes**: Fair (FIFO) and non-fair
4. **Used internally** by `Executors.newCachedThreadPool()`
5. **Higher throughput** than buffered queues for handoff scenarios

Would you like an example of implementing a **custom load balancer** using `SynchronousQueue`? 

DelayQueue in Java - Complete Guide

`DelayQueue` is a specialized blocking queue that holds elements until their specified delay has expired. It's part of the `java.util.concurrent` package and is used for scheduling tasks that should become available after a certain time.

Key Features

- **Time-based scheduling**: Elements can only be taken when their delay has expired
- **Unbounded capacity**: Can grow as needed
- **Thread-safe**: Designed for concurrent access
- **Elements must implement `Delayed` interface**
- **Automatically orders elements** by expiration time

Real-Time Use Cases

1. **Session timeout handling** (web applications)
2. **Task scheduling systems** (delayed job execution)
3. **Cache expiration** (auto-removal of stale entries)
4. **Retry mechanisms** (delayed retries for failed operations)
5. **Game development** (timed power-ups or effects)
6. **Financial systems** (time-based order expiration)

Core Methods

1. Required Interface (Delayed)

java

Copy

Download

```
public interface Delayed extends Comparable<Delayed> {  
    long getDelay(TimeUnit unit);  
}
```

2. Constructors

java

Copy

Download

```
DelayQueue<DelayedElement> queue = new DelayQueue<>();
```

3. Insertion Methods

Method	Behavior
<code>add(E e)</code>	Adds immediately (throws exception if capacity restricted - though unbounded)
<code>offer(E e)</code>	Always returns true (unbounded)
<code>put(E e)</code>	Same as offer()

4. Removal Methods

Method	Behavior
--------	----------

<code>poll()</code>	Retrieves and removes expired element (returns null if none available)
<code>take()</code>	Blocks until an element's delay expires
<code>poll(long timeout, TimeUnit unit)</code>	Waits for specified time for an element to expire

5. Inspection Methods

Method	Behavior
<code>peek()</code>	Retrieves but doesn't remove head (may be unexpired)
<code>size()</code>	Returns count of all elements (expired and unexpired)

Complete Example: Cache Expiration System

java

Copy

Download

```
import java.util.concurrent.*;

class CacheItem implements Delayed {
    private String data;
    private long expiryTime;

    public CacheItem(String data, long delay, TimeUnit unit) {
        this.data = data;
        this.expiryTime = System.currentTimeMillis() + unit.toMillis(delay);
    }
}
```

```

    }

    @Override
    public long getDelay(TimeUnit unit) {
        long remaining = expiryTime - System.currentTimeMillis();
        return unit.convert(remaining, TimeUnit.MILLISECONDS);
    }

    @Override
    public int compareTo(Delayed other) {
        return Long.compare(this.expiryTime, ((CacheItem)other).expiryTime);
    }

    public String getData() { return data; }
}

public class DelayQueueDemo {
    public static void main(String[] args) throws InterruptedException {
        DelayQueue<CacheItem> cache = new DelayQueue<>();

        // Add items with different expiration times
        cache.put(new CacheItem("A", 3, TimeUnit.SECONDS));
        cache.put(new CacheItem("B", 1, TimeUnit.SECONDS));
        cache.put(new CacheItem("C", 5, TimeUnit.SECONDS));

        System.out.println("Cache monitoring started at: " + new
java.util.Date());

        while (!cache.isEmpty()) {
            CacheItem item = cache.take(); // Blocks until next item expires
            System.out.printf("Item %s expired at %s\n",
                item.getData(), new java.util.Date());
        }
    }
}

```

Output:

Copy

Download

```

Cache monitoring started at: Wed Oct 05 14:30:00 EDT 2023
Item B expired at Wed Oct 05 14:30:01 EDT 2023
Item A expired at Wed Oct 05 14:30:03 EDT 2023

```

Advanced Example: Retry Mechanism

java

Copy

Download

```
class RetryTask implements Delayed, Runnable {
    private final Runnable task;
    private long nextRetryTime;
    private int retryCount;

    public RetryTask(Runnable task, long initialDelay, TimeUnit unit) {
        this.task = task;
        this.nextRetryTime = System.currentTimeMillis() +
unit.toMillis(initialDelay);
        this.retryCount = 0;
    }

    public void reschedule(long delay, TimeUnit unit) {
        this.nextRetryTime = System.currentTimeMillis() + unit.toMillis(delay);
        this.retryCount++;
    }

    @Override public void run() {
        try {
            task.run();
        } catch (Exception e) {
            if (retryCount < 3) {
                reschedule(5, TimeUnit.SECONDS); // Retry after 5 seconds
            }
        }
    }

    @Override public long getDelay(TimeUnit unit) {
        long remaining = nextRetryTime - System.currentTimeMillis();
        return unit.convert(remaining, TimeUnit.MILLISECONDS);
    }

    @Override public int compareTo(Delayed other) {
        return Long.compare(this.nextRetryTime, ((RetryTask)other).nextRetryTime);
    }
}
```

```

}

public class RetrySystem {
    public static void main(String[] args) {
        DelayQueue<RetryTask> retryQueue = new DelayQueue<>();

        retryQueue.add(new RetryTask(() -> {
            System.out.println("Processing task...");
            throw new RuntimeException("Simulated failure");
        }, 0, TimeUnit.SECONDS));

        new Thread(() -> {
            while (true) {
                try {
                    RetryTask task = retryQueue.take();
                    new Thread(task).start();
                    if (task.retryCount < 3) {
                        retryQueue.add(task); // Requeue for retry
                    }
                } catch (InterruptedException e) {
                    break;
                }
            }
        }).start();
    }
}

```

Performance Characteristics

- **Insertion:** $O(\log n)$ time (maintains heap structure)
- **Removal:** $O(\log n)$ time for expired elements
- **Memory:** Grows as needed (unbounded)

Comparison with Similar Classes

Feature	DelayQueue	ScheduledThreadPoolExecut or	Timer
Precision	Good	Good	Poor

Concurrency	High	High	Low
Task	Manual	Built-in	Built-in
Rescheduling			
Error Handling	Manual	Better	Poor
Use Case	Custom delay logic	General scheduling	Simple tasks

Best Practices

1. **Always implement** `getDelay()` **and** `compareTo()` **correctly**
2. **Monitor queue size** (unbounded growth can cause OOM)
3. **Consider using** `ScheduledThreadPoolExecutor` for simpler scheduling needs
4. **Handle thread interruption** properly in consumer threads
5. **Use** `poll()` **instead of** `take()` when you need timeout control

Real-World Implementations

1. **Netflix Hystrix**: Uses `DelayQueue` for circuit breaker retries
2. **Akka**: Scheduler implementation options
3. **Quartz**: Alternative scheduling framework
4. **Kafka**: Delayed operation handling

Would you like an example of implementing a **rate limiter** using `DelayQueue`? 🚦

ConcurrentLinkedQueue in Java - Complete Guide

`ConcurrentLinkedQueue` is a thread-safe, unbounded FIFO (First-In-First-Out) queue implementation in Java's `java.util.concurrent` package. It's designed for

high-performance concurrent scenarios where multiple threads need to access a shared queue.

Key Features

- **Thread-safe:** Lock-free algorithm using CAS (Compare-And-Swap)
- **Unbounded:** Grows as needed (no capacity restrictions)
- **Non-blocking:** No thread blocking during operations
- **FIFO ordering:** Strict first-in-first-out behavior
- **High throughput:** Optimized for concurrent access
- **Iterators are weakly consistent:** Reflect queue state at some point in time

When to Use ConcurrentLinkedQueue

- ✓ Multi-producer, multi-consumer scenarios
- ✓ High-concurrency environments
- ✓ When unbounded growth is acceptable
- ✓ Non-blocking operations are preferred

All Methods with Examples

1. Constructors

java

Copy

Download

```
// Empty queue
```

```
ConcurrentLinkedQueue<String> queue1 = new ConcurrentLinkedQueue<>();
```

```
// Initialize from existing collection
```

```
ConcurrentLinkedQueue<String> queue2 = new ConcurrentLinkedQueue<>(  
    Arrays.asList("A", "B", "C"));
```

2. Insertion Methods

java

Copy

Download

```
queue1.add("Task1");           // Throws exception if capacity restricted (though  
unbounded)
```

```
queue1.offer("Task2");        // Preferred method, returns boolean
```

```
// No put() method since it's non-blocking
```

3. Removal Methods

java

Copy

Download

```
String item1 = queue1.poll();  // Retrieves and removes head (returns null if  
empty)
```

```
String item2 = queue1.remove(); // Throws NoSuchElementException if empty
```

```
// No take() method since it's non-blocking
```

4. Inspection Methods

java

Copy

[Download](#)

```
String head = queue1.peek();    // Retrieves head without removal (null if empty)
String element = queue1.element(); // Throws exception if empty
```

5. Utility Methods

[java](#)

[Copy](#)

[Download](#)

```
int size = queue1.size();        // Warning: Not constant-time operation!

boolean isEmpty = queue1.isEmpty(); // More efficient than size() == 0
boolean contains = queue1.contains("Task1");
```

6. Bulk Operations

[java](#)

[Copy](#)

[Download](#)

```
Object[] array = queue1.toArray();
String[] stringArray = queue1.toArray(new String[0]);
```

Complete Example: Multi-Threaded Task Processor

[java](#)

[Copy](#)

[Download](#)

```
import java.util.concurrent.ConcurrentLinkedQueue;

public class TaskProcessor {
```

```
private static final ConcurrentLinkedQueue<Runnable> taskQueue =

    new ConcurrentLinkedQueue<>();

private static volatile boolean running = true;

public static void main(String[] args) {

    // Start worker threads

    for (int i = 0; i < 4; i++) {

        new Thread(() -> {

            while (running || !taskQueue.isEmpty()) {

                Runnable task = taskQueue.poll();

                if (task != null) {

                    task.run();

                } else {

                    Thread.yield(); // Avoid busy waiting

                }

            }

        }).start();

    }

    // Producer thread
```

```

new Thread(() -> {

    for (int i = 0; i < 10; i++) {

        final int taskId = i;

        taskQueue.offer(() -> {

            System.out.println("Processing task " + taskId +

                               " by " + Thread.currentThread().getName());

        });

        try {

            Thread.sleep(100); // Simulate work creation delay

        } catch (InterruptedException e) {

            Thread.currentThread().interrupt();

        }

    }

    running = false;

}).start();

}

```

Sample Output:

Copy

Download

Processing task 0 by Thread-0

Processing task 1 by Thread-1

Processing task 2 by Thread-2

Processing task 3 by Thread-3

Processing task 4 by Thread-0

...

Performance Characteristics

Operation	Time Complexity	Notes
<code>offer()</code>	$O(1)$	Amortized constant time
<code>poll()</code>	$O(1)$	Amortized constant time
<code>size()</code>	$O(n)$	Must traverse entire queue
<code>peek()</code>	$O(1)$	Constant time

Comparison with Other Concurrent Queues

Feature	ConcurrentLinkedQueue	LinkedBlockingQueue	ArrayBlockingQueue
	e	e	e

Bounded	No	Optional	Yes
Blocking	No	Yes	Yes
Locking	Lock-free (CAS)	ReentrantLock	ReentrantLock
Throughput	Very High	High	Medium
Ordering	FIFO	FIFO	FIFO
Best For	Highest concurrency	Balanced throughput	Fixed capacity

Important Considerations

1. **Size() is expensive:** Avoid frequent calls in performance-critical code
2. **No blocking operations:** Use `LinkedBlockingQueue` if you need `put()/take()`
3. **Bulk operations:** Not atomic (other threads may modify queue during operation)
4. **Iterators:** Weakly consistent - may or may not reflect concurrent modifications

Real-World Use Cases

1. **Event processing systems** (multiple producers, multiple consumers)
2. **Work-stealing algorithms**
3. **Message passing between threads**
4. **Task scheduling in thread pools**
5. **Low-latency trading systems**

Best Practices

1. **Prefer** `offer()/poll()` over `add()/remove()` for better control
2. **Check** `isEmpty()` instead of `size() == 0` when possible
3. **Combine with other concurrency tools** like `CountDownLatch` for complex coordination
4. **Consider** `LinkedBlockingQueue` if you need blocking operations
5. **Monitor queue growth** (unbounded queues can lead to OOM)

Would you like an example of implementing a **work-stealing algorithm** using `ConcurrentLinkedQueue`?

ConcurrentLinkedDeque in Java - Complete Guide

`ConcurrentLinkedDeque` is a thread-safe, unbounded double-ended queue (deque) implementation in Java's `java.util.concurrent` package. It supports insertion, removal, and access operations at both ends with high concurrency.

Key Features

- **Thread-safe:** Lock-free algorithm using CAS (Compare-And-Swap)
- **Unbounded:** Grows as needed (no capacity restrictions)
- **Non-blocking:** No thread blocking during operations
- **Double-ended:** Supports operations at both head and tail
- **High throughput:** Optimized for concurrent access
- **Iterators are weakly consistent:** Reflect queue state at some point in time

When to Use ConcurrentLinkedDeque

- ✓ Multi-threaded producer/consumer scenarios requiring operations at both ends
- ✓ High-concurrency environments needing non-blocking behavior
- ✓ Work-stealing algorithms
- ✓ When unbounded growth is acceptable

All Methods with Examples

1. Constructors

java

Copy

Download

```
// Empty deque
```

```
ConcurrentLinkedDeque<String> deque1 = new ConcurrentLinkedDeque<>();
```

```
// Initialize from existing collection
```

```
ConcurrentLinkedDeque<String> deque2 = new ConcurrentLinkedDeque<>(  
    Arrays.asList("A", "B", "C"));
```

2. Insertion Methods (First/Last)

java

Copy

Download

```
// Add at beginning
```

```
deque1.addFirst("Task1");    // Throws exception if capacity restricted (though
unbounded)

deque1.offerFirst("Task2");  // Preferred method, returns boolean

deque1.push("Task3");        // Same as addFirst()

// Add at end

deque1.addLast("Task4");     // Throws exception if restricted

deque1.offerLast("Task5");   // Preferred method

deque1.add("Task6");         // Same as addLast()
```

3. Removal Methods (First/Last)

java

Copy

Download

```
// Remove from beginning

String first1 = deque1.pollFirst(); // Retrieves and removes (returns null if
empty)

String first2 = deque1.removeFirst(); // Throws NoSuchElementException if empty

String first3 = deque1.pop();         // Same as removeFirst()

// Remove from end

String last1 = deque1.pollLast();     // Retrieves and removes (returns null if
empty)
```

```
String last2 = deque1.removeLast(); // Throws NoSuchElementException if empty
```

4. Inspection Methods (First/Last)

java

Copy

Download

```
// Examine first element
```

```
String head1 = deque1.peekFirst(); // Retrieves without removal (null if empty)
```

```
String head2 = deque1.getFirst(); // Throws exception if empty
```

```
String head3 = deque1.peek(); // Same as peekFirst()
```

```
String head4 = deque1.element(); // Same as getFirst()
```

```
// Examine last element
```

```
String tail1 = deque1.peekLast(); // Retrieves without removal (null if empty)
```

```
String tail2 = deque1.getLast(); // Throws exception if empty
```

5. Utility Methods

java

Copy

Download

```
int size = deque1.size(); // Warning: Not constant-time operation!
```

```
boolean isEmpty = deque1.isEmpty(); // More efficient than size() == 0
```

```
boolean contains = deque1.contains("Task1");
```

6. Bulk Operations

java

Copy

Download

```
Object[] array = deque1.toArray();
```

```
String[] stringArray = deque1.toArray(new String[0]);
```

```
deque1.removeIf(e -> e.startsWith("Task"));
```

Complete Example: Work-Stealing System

java

Copy

Download

```
import java.util.concurrent.ConcurrentLinkedDeque;
```

```
public class WorkStealingSystem {
```

```
    private static final ConcurrentLinkedDeque<Runnable> globalQueue =
```

```
        new ConcurrentLinkedDeque<>();
```

```
    private static final int WORKER_COUNT = 4;
```

```
    private static volatile boolean running = true;
```

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

```
        // Initialize worker threads
```

```
        for (int i = 0; i < WORKER_COUNT; i++) {
```

```
            final int workerId = i;
```

```

new Thread(() -> {

    ConcurrentLinkedDeque<Runnable> localQueue = new
ConcurrentLinkedDeque<>();

    // Each worker adds some initial tasks

    for (int j = 0; j < 3; j++) {

        final int taskId = workerId * 10 + j;

        localQueue.offerLast(() -> {

            System.out.printf("Worker %d executing task %d\n",

                workerId, taskId);

        });

    }

    while (running || !globalQueue.isEmpty() || !localQueue.isEmpty())
{

    Runnable task;

    // 1. Try local queue first (LIFO)

    task = localQueue.pollLast();

    if (task != null) {

        task.run();

```

```

        continue;
    }

    // 2. Try stealing from global queue (FIFO)

    task = globalQueue.pollFirst();

    if (task != null) {

        task.run();

        continue;
    }

    // 3. Try stealing from other workers (not shown for
simplicity)

    Thread.yield();

    }

    }).start();

}

// Producer thread adding to global queue

new Thread(() -> {

    for (int i = 0; i < 10; i++) {

```

```

        final int taskId = i;

        globalQueue.offerLast(() -> {

            System.out.println("Global task " + taskId);

        });

        try {

            Thread.sleep(50);

        } catch (InterruptedException e) {

            Thread.currentThread().interrupt();

        }

    }

    running = false;

}).start();

}

```

Performance Characteristics

Operation	Time Complexity	Notes
<code>addFirst()/addLast()</code>	$O(1)$	Amortized constant time
<code>pollFirst()/pollLast()</code>	$O(1)$	Amortized constant time

<code>size()</code>	$O(n)$	Must traverse entire deque
<code>peekFirst()/peekLast()</code>	$O(1)$	Constant time

Comparison with Other Concurrent Deques

Feature	<code>ConcurrentLinkedDeque</code>	<code>LinkedBlockingDeque</code>	<code>ArrayDeque</code>
Thread-safe	Yes	Yes	No
Bounded	No	Optional	No
Blocking	No	Yes	No
Locking	Lock-free (CAS)	<code>ReentrantLock</code>	None
Null Elements	No	No	Yes
Best For	Highest concurrency	Balanced throughput	Single-threaded

Real-World Use Cases

- 1. **Work-stealing algorithms** (like `ForkJoinPool`)

2. **Multi-threaded event systems**
3. **Producer/consumer with dual access points**
4. **Low-latency trading systems**
5. **Concurrent job scheduling**

Best Practices

1. **Prefer** `offerFirst()/offerLast()` over `addFirst()/addLast()`
2. **Use** `pollFirst()/pollLast()` for non-blocking retrieval
3. **Avoid frequent** `size()` **calls** - use `isEmpty()` instead
4. **Combine with other concurrent utilities** for complex coordination
5. **Monitor memory usage** (unbounded growth risk)

Key Differences from ConcurrentLinkedQueue

1. **Double-ended operations:** Supports both FIFO and LIFO patterns
2. **More method variants:** Additional first/last operations
3. **Better for work-stealing:** Can efficiently pop from both ends
4. **Same performance characteristics:** Similar $O(1)$ operations

Would you like an example of implementing a **concurrent job scheduler** using

`ConcurrentLinkedDeque`? 🕒