Queues & Collections Utility

The final topic in the Collections category: the `Queue` / `Deque` interfaces (offer()/poll()/peek() vs add()/remove()/element()), why `ArrayDeque` is preferred over `LinkedList` / `java.util.Stack` as both a queue and a stack, a real discovery that `PriorityQueue` is heap-based and its toString() is NOT sorted, and the `Collections` utility class's static methods like sort/shuffle/max/binarySearch.

Beginner 20 min

Last updated: August 17, 2026
Read online: https://www.learnforgex.com/en/topics/queues-collections-utility

---

Queues & Collections Utility

In the last stop of the Collections category, we bring together two different but complementary topics: `Queue` / `Deque` (collections designed to process elements in a specific order -- FIFO, LIFO, or by priority) and the `Collections` utility class (ready-made static methods that work on any collection). Both are short, independent topics, so they're combined into a single topic here -- the same reasoning applied in the "Primitive & Parallel Streams" lesson.

What Are Queue and Deque?

`Queue<E>` is an interface designed to process elements in a specific order -- its most common use is FIFO (first-in-first-out), just like a waiting line. `Deque<E>` ("double-ended queue") extends `Queue` and allows adding/removing from BOTH ends -- which means it can be used both as a queue (FIFO) and as a stack (LIFO -- last-in-first-out). Its most common implementations are `ArrayDeque` (a circular array, the fastest) and `LinkedList` (which implements `List`, `Deque`, and `Queue` all at once).

Why Does It Exist?

A task queue, a message queue, an "undo" history, breadth-first search (BFS) in a graph -- all of these rely on the idea of "process elements in a specific order". `List` could theoretically do something similar (`add(0, x)` or `remove(0)`), but those operations are O(n) on an `ArrayList` (see the "Lists" lesson) -- `Queue` / `Deque` implementations are designed to do these operations in O(1).

History

The `Queue` interface arrived with Java 5 (2004) -- it wasn't part of the Collections Framework's first release (1998). `Deque` and `ArrayDeque` were added in Java 6 (2006); `ArrayDeque`'s official javadoc explicitly states that it is usually faster than both the `Stack` class and `LinkedList` (when used as a deque) and should be preferred. `PriorityQueue` also arrived with Java 5, as a priority-queue (heap) implementation.

Queue Basics: Two Parallel Method Families

Every `Queue` operation has TWO parallel methods: one throws an EXCEPTION on failure (`add()` , `remove()` , `element()`), the other returns a special value (`offer()` , `poll()` , `peek()` -- `false` / `null` / `null` respectively). The general rule: the `offer()` / `poll()` / `peek()` family is preferred, because it handles a normal condition like "the queue is empty" with a checkable return value instead of throwing an exception.

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

public class QueueBasicsExample {
    public static void main(String[] args) {
        Queue<String> queue = new LinkedList<>();
        queue.offer("first");
        queue.offer("second");
        queue.offer("third");

        System.out.println("Queue: " + queue);
        System.out.println("peek() (look at the head, don't remove): " + queue.peek());
        System.out.println("poll() (remove and return the head): " + queue.poll());
        System.out.println("Queue after poll(): " + queue);

        // Queue has TWO parallel method families: one throws on failure, one returns
        // a special value (null or false). Draining the queue with poll() shows the
        // "special value" family:
        while (queue.poll() != null) {
            // draining
        }
        System.out.println("poll() on an empty queue: " + queue.poll()); // null, no
exception
        System.out.println("peek() on an empty queue: " + queue.peek()); // null, no
exception

        // The "throwing" family: add()/remove()/element() throw instead of returning
        // null/false on failure.
        try {
            queue.remove(); // empty queue -- throws
        } catch (NoSuchElementException e) {
            System.out.println("remove() on an empty queue: " + e.getClass().
getSimpleName());
        }
        try {
            queue.element(); // empty queue -- throws
        } catch (NoSuchElementException e) {
            System.out.println("element() on an empty queue: " + e.getClass().
getSimpleName());
        }
    }
}
```

> Calling `remove()` / `element()` on an empty queue throws `NoSuchElementException` -- a textbook example of using an exception for a normal, expected condition like "is the queue empty". `poll()` / `peek()` returning `null` is usually more readable and less costly (throwing/catching an exception is expensive).

Deque: Access From Both Ends

`Deque` provides access to both ends with `addFirst()` / `addLast()`, `removeFirst()` / `removeLast()`, `peekFirst()` / `peekLast()` (and their `offer` / `poll`-prefixed, non-throwing counterparts).

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

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

        // A Deque (double-ended queue) can insert and remove at BOTH ends.
        deque.addFirst("b");
        deque.addFirst("a"); // now the front
        deque.addLast("c");
        deque.addLast("d"); // now the back

        System.out.println("Deque: " + deque);
        System.out.println("peekFirst(): " + deque.peekFirst());
        System.out.println("peekLast(): " + deque.peekLast());

        System.out.println("removeFirst(): " + deque.removeFirst());
        System.out.println("removeLast(): " + deque.removeLast());
        System.out.println("Deque after removing both ends: " + deque);

        // Just like Queue, Deque also has an "offer" family that returns a boolean
        // instead of throwing (offerFirst/offerLast, pollFirst/pollLast).
        deque.offerFirst("x");
        deque.offerLast("y");
        System.out.println("After offerFirst(x)/offerLast(y): " + deque);
    }
}
```

Using ArrayDeque as a Stack

`Deque` can also be used as a STACK (LIFO -- last-in-first-out) via `push()` / `pop()`. The javadoc of Java's own `java.util.Stack` class OFFICIALLY recommends using `Deque` (specifically `ArrayDeque`) instead of this legacy class -- because `Stack` extends `Vector`, which means it inherits unnecessary synchronization overhead and index-based methods that don't fit the concept of a stack (like `insertElementAt()`).

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

public class ArrayDequeAsStackExample {
    public static void main(String[] args) {
        // The official recommendation (per the java.util.Stack javadoc itself) is to
        // use Deque as a stack via push()/pop(), NOT the legacy Stack class.
        Deque<Integer> stack = new ArrayDeque<>();
        stack.push(1);
        stack.push(2);
        stack.push(3); // last pushed...

        System.out.println("Stack (as Deque): " + stack);
        System.out.println("peek() (top of stack): " + stack.peek());
        System.out.println("pop(): " + stack.pop()); // ...is first popped: LIFO
        System.out.println("Stack after pop(): " + stack);

        // The old java.util.Stack class still works and gives the same LIFO
        // behavior, but it extends Vector, which means it inherits synchronized
        // methods (unnecessary overhead in single-threaded code) and index-based
        // methods that don't make sense for a stack (like insertElementAt()).
        Stack<Integer> legacyStack = new Stack<>();
        legacyStack.push(10);
        legacyStack.push(20);
        System.out.println("Legacy Stack: " + legacyStack + ", pop(): " + legacyStack.
pop());
    }
}
```

Performance: ArrayDeque vs. LinkedList

`ArrayDeque` and `LinkedList` are both theoretically O(1) for the same `Deque` operations -- but the constant factors differ: `LinkedList` allocates a separate node object for every element, while `ArrayDeque` uses a circular array and avoids that overhead.

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

public class ArrayDequeVsLinkedListPerformanceExample {
    public static void main(String[] args) {
        int rounds = 5_000_000;

        Deque<Integer> arrayDeque = new ArrayDeque<>();
        Deque<Integer> linkedList = new LinkedList<>();

        // Warm-up -- run both paths a lot before measuring.
        for (int i = 0; i < 500_000; i++) {
            arrayDeque.offer(i);
            arrayDeque.poll();
            linkedList.offer(i);
            linkedList.poll();
        }

        long arrayDequeStart = System.nanoTime();
        for (int i = 0; i < rounds; i++) {
            arrayDeque.offer(i);
            arrayDeque.poll();
        }
        long arrayDequeNanos = System.nanoTime() - arrayDequeStart;

        long linkedListStart = System.nanoTime();
        for (int i = 0; i < rounds; i++) {
            linkedList.offer(i);
            linkedList.poll();
        }
        long linkedListNanos = System.nanoTime() - linkedListStart;

        System.out.println("offer()+poll() pairs, " + rounds + " times:");
        System.out.println("  ArrayDeque: " + (arrayDequeNanos / 1_000_000) + " ms");
        System.out.println("  LinkedList: " + (linkedListNanos / 1_000_000) + " ms");
        System.out.println("(both are O(1) for these operations -- ArrayDeque wins mainly
on constant");
        System.out.println(" factors: no per-element node objects, better memory
locality)");
    }
}
```

Real measurement: across 5 million `offer()` + `poll()` pairs, `ArrayDeque` came out noticeably faster than `LinkedList` in most runs (for example ~40 ms vs. ~55-60 ms), but the margin wasn't identical from run to run -- in some runs the two were much closer. This is consistent with `LinkedList` allocating a separate object per element, which creates variable pressure on the garbage collector. Still, `ArrayDeque` was never measured slower than `LinkedList` in any run.

PriorityQueue: By Priority, Not By Order

`PriorityQueue` processes elements NOT in insertion order, but by natural ordering (or by a given `Comparator`) -- the smallest element (or the "highest priority" one per the `Comparator`) always comes out first via `peek()` / `poll()`. But watch out: this is ONLY true for `peek()` / `poll()` -- printing a `PriorityQueue` directly or iterating it with an `Iterator` does NOT show the elements in sorted order.

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

public class PriorityQueueExample {
    public static void main(String[] args) {
        Queue<Integer> pq = new PriorityQueue<>();
        for (int n : new int[]{50, 10, 40, 20, 30}) {
            pq.offer(n);
        }

        // SURPRISE: a PriorityQueue's toString()/iterator does NOT print elements in
        // sorted order -- it only guarantees that the HEAD (peek()) is the smallest.
        // The rest of the internal heap array can be in any order.
        System.out.println("PriorityQueue printed directly (NOT necessarily sorted!): " +
pq);
        System.out.println("peek() (always the smallest): " + pq.peek());

        // The only way to actually get elements out in sorted order is to poll()
        // repeatedly.
        System.out.print("Polling one by one (this IS sorted): ");
        Queue<Integer> copy = new PriorityQueue<>(pq);
        while (!copy.isEmpty()) {
            System.out.print(copy.poll() + " ");
        }
        System.out.println();

        // A custom Comparator reverses the priority -- now the LARGEST is the head.
        Queue<Integer> maxHeap = new PriorityQueue<>(Comparator.reverseOrder());
        maxHeap.offer(50);
        maxHeap.offer(10);
        maxHeap.offer(40);
        System.out.println("Max-heap peek() (largest is now the head): " + maxHeap.peek());
    }
}
```

> `PriorityQueue`'s `toString()` (or iterating it directly with an `Iterator`) can give the WRONG impression that elements appear sorted -- the real output in the example above proves this: `[10, 20, 40, 50, 30]`, NOT sorted. `PriorityQueue` uses a heap internally -- only the root (the first element of the array) is guaranteed to be the smallest; there's no ordering guarantee for the rest. The only way to actually get elements out in sorted order is to call `poll()` repeatedly.

The Collections Utility Class

`Collections`, similar to `Collectors` (see the "Collectors" lesson), is a utility class that offers ready-made static methods working on any `Collection`/`List`: `sort()`, `reverse()`, `shuffle()`, `max()`/`min()`, `frequency()` (counts how many times a value occurs), `binarySearch()` (O(log n) search on a SORTED list), and small, special-purpose immutable-collection factory methods like `emptyList()`/`singletonList()`/`nCopies()`.

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.Random;

public class CollectionsUtilityExample {
    public static void main(String[] args) {
        List<Integer> numbers = new ArrayList<>(List.of(5, 3, 8, 1, 9, 2));

        Collections.sort(numbers);
        System.out.println("Collections.sort(): " + numbers);

        Collections.reverse(numbers);
        System.out.println("Collections.reverse(): " + numbers);

        System.out.println("Collections.max(): " + Collections.max(numbers));
        System.out.println("Collections.min(): " + Collections.min(numbers));

        List<String> letters = List.of("a", "b", "a", "c", "a", "b");
        System.out.println("Collections.frequency(letters, \"a\"): " + Collections.frequency
(letters, "a"));

        // binarySearch() requires a SORTED list -- O(log n) instead of a linear scan.
        Collections.sort(numbers);
        System.out.println("Sorted for binarySearch(): " + numbers);
        System.out.println("Collections.binarySearch(numbers, 8): index " + Collections.
binarySearch(numbers, 8));

        // shuffle() with a seeded Random -- deterministic here only so the example's
        // output is reproducible; in real code you'd normally omit the seed.
        List<Integer> toShuffle = new ArrayList<>(List.of(1, 2, 3, 4, 5));
        Collections.shuffle(toShuffle, new Random(42));
        System.out.println("Collections.shuffle() (seeded for reproducibility): " +
toShuffle);

        // Factory methods for small/special-purpose immutable collections.
        System.out.println("Collections.emptyList(): " + Collections.emptyList());
        System.out.println("Collections.singletonList(\"x\"): " + Collections.singletonList
("x"));
        System.out.println("Collections.nCopies(4, \"z\"): " + Collections.nCopies(4, "z"));
    }
}
```

> For `Collections.binarySearch()` to work correctly, the list MUST be sorted beforehand -- calling it on an unsorted list doesn't throw an exception but returns a WRONG result (a silent bug). If you're not sure, call `Collections.sort()` first.

Best Practices

- **Use `ArrayDeque` with `push()` / `pop()` instead of `java.util.Stack` for a stack** -- this is Java's own official recommendation.
- **Prefer `ArrayDeque` by default over `LinkedList` as a queue/deque** -- it's almost always at least as fast, usually faster, and uses less memory (it doesn't allocate a separate node object per element).
- **Prefer the `offer()` / `poll()` / `peek()` family for normal conditions like "the queue is empty"**, not `add()` / `remove()` / `element()` -- throwing/catching an exception is expensive for normal control flow.
- **Don't rely on printing a `PriorityQueue` directly or iterating it with an `Iterator`** -- call `poll()` repeatedly if you want sorted output.

Common Mistakes

- **Calling `remove()` / `element()` on an empty queue and getting `NoSuchElementException`.** A normal "is it empty" check should use the `offer()` / `poll()` / `peek()` family instead.
- **Assuming `PriorityQueue`'s `toString()` or `Iterator` is sorted.** Only `peek()` / `poll()` guarantee ordering.
- **Calling `Collections.binarySearch()` on an unsorted list.** It doesn't throw an exception, but it returns a wrong result -- `Collections.sort()` must always be called first.
- **Unnecessarily using `java.util.Stack` or `Vector` in a scenario that needs frequent adds/removes.** These are legacy, synchronized classes -- modern code should use `ArrayDeque` / `ArrayList`.

Summary, Cheat Sheet, and Glossary

`Queue` processes elements in a specific order (usually FIFO); `Deque` provides access from both ends, so it can be used as both a queue and a stack. `ArrayDeque` is the modern implementation preferred over both `LinkedList` (as a Deque) and `java.util.Stack` (as a stack). `PriorityQueue` processes elements by priority, but only `poll()` / `peek()` guarantee ordering. The `Collections` utility class offers ready-made static methods that work on any list/collection.

Quick reference:

```
Queue<String> queue = new ArrayDeque<>();     // FIFO -- offer()/poll()/peek()
Deque<String> stack = new ArrayDeque<>();      // LIFO -- push()/pop()/peek()
Queue<Integer> pq = new PriorityQueue<>();      // by priority -- poll() is sorted,
toString() is NOT
Collections.sort(list);                            // sort in place
Collections.max(list); Collections.min(list);         // largest/smallest
Collections.frequency(list, value);                     // count occurrences
Collections.binarySearch(sortedList, value);              // O(log n) search (sort FIRST!)
```

Glossary

Queue — A collection interface that processes elements in a specific order (usually FIFO).

Deque — A `Queue` subinterface that allows adding/removing from both ends, usable as both a queue and a stack.

ArrayDeque — A circular-array-backed `Deque` implementation, preferred over `LinkedList` and `java.util.Stack`.

PriorityQueue — A heap-based `Queue` implementation that processes elements by priority (natural ordering or a `Comparator`), not insertion order.

Collections — A utility class offering ready-made static methods (`sort`, `reverse`, `max`, `binarySearch`, etc.) that work on any collection.