Queues & Collections Utility

Collections kategorisinin son topic'i: `Queue` / `Deque` arayüzleri (offer()/poll()/peek() vs add()/remove()/element()), `ArrayDeque`'ın hem kuyruk hem stack olarak `LinkedList` / `java.util.Stack`'a tercih edilmesi, `PriorityQueue`'nun heap tabanlı ve toString()'inin sıralı OLMADIĞI gerçek bir keşif, ve `Collections` yardımcı sınıfının sort/shuffle/max/binarySearch gibi statik metotları.

Başlangıç 20 dk

Son güncelleme: 6 Ekim 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/queues-collections-utility

---

Queues & Collections Utility

Koleksiyonlar kategorisinin son durağında iki farklı ama birbirini tamamlayan konuyu bir araya getiriyoruz: `Queue` / `Deque` (elemanları belirli bir sırada -- FIFO, LIFO, ya da önceliğe göre -- işlemek için tasarlanmış koleksiyonlar) ve `Collections` yardımcı sınıfı (herhangi bir koleksiyon üzerinde çalışan hazır statik metotlar). İkisi de kısa, bağımsız konular olduğu için tek bir topic'te birleştirildi -- tıpkı "Primitive & Parallel Streams" dersinde uygulanan aynı gerekçeyle.

Queue ve Deque Nedir?

`Queue<E>`, elemanları belirli bir sırayla işlemek için tasarlanmış bir arayüzdür -- en yaygın kullanımı FIFO'dur (first-in-first-out, ilk giren ilk çıkar), tıpkı bir bekleme sırası gibi. `Deque<E>` ("double-ended queue", "dek" diye okunur), `Queue`'yu genişletir ve HER İKİ uçtan da ekleme/çıkarma yapılabilmesini sağlar -- bu sayede hem kuyruk (FIFO) hem de yığın (LIFO -- last-in-first-out) olarak kullanılabilir. En yaygın implementasyonları `ArrayDeque` (dairesel bir dizi, en hızlı) ve `LinkedList`'tir (`List`, `Deque` ve `Queue`'nun hepsini birden implement eder).

Neden Var?

Bir görev kuyruğu, bir mesaj sırası, "geri al" (undo) geçmişi, bir grafikte genişlik-öncelikli arama (BFS) -- bunların hepsi "elemanları belirli bir sırayla işle" fikrine dayanır. `List` ile de teorik olarak benzer bir şey yapılabilir (`add(0, x)` ya da `remove(0)`), ama bu işlemler `ArrayList`'te O(n)'dir (bkz. "Lists" dersi) -- `Queue` / `Deque` implementasyonları bu işlemleri O(1)'de yapacak şekilde tasarlanmıştır.

Tarihçe

`Queue` arayüzü Java 5 (2004) ile geldi -- Collections Framework'ün ilk sürümünde (1998) yoktu. `Deque` ve `ArrayDeque` Java 6 (2006) ile eklendi; `ArrayDeque`'ın resmi javadoc'u, hem `Stack` sınıfına hem de `Deque` olmadığında `LinkedList`'e göre daha hızlı olduğunu ve tercih edilmesi gerektiğini açıkça belirtir. `PriorityQueue` de Java 5 ile geldi, bir öncelik kuyruğu (heap) implementasyonu olarak.

Queue Temelleri: İki Paralel Metot Ailesi

`Queue`'nun her işlemi için İKİ paralel metot vardır: biri başarısızlıkta İSTİSNA fırlatır (`add()`, `remove()`, `element()`), diğeri özel bir değer döner (`offer()`, `poll()`, `peek()` -- sırasıyla `false` / `null` / `null`). Genel kural: `offer()` / `poll()` / `peek()` ailesi tercih edilir, çünkü "kuyruk boş" gibi normal bir durumu istisna fırlatarak değil, kontrol edilebilir bir dönüş değeriyle ele alır.

```
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());
        }
    }
}
```

> Boş bir kuyrukta `remove()` / `element()` çağırmak `NoSuchElementException` fırlatır -- bu, "kuyruk boş mu" gibi normal, beklenen bir durum için istisna kullanmanın tipik bir örneğidir. `poll()` / `peek()` 'in `null` dönmesi genellikle daha okunabilir ve daha az maliyetlidir (istisna fırlatmak/yakalamak pahalıdır).

Deque: Her İki Uçtan da Erişim

`Deque`, `addFirst()`/`addLast()`, `removeFirst()`/`removeLast()`, `peekFirst()`/`peekLast()` (ve bunların `offer`/`poll` ile başlayan, istisna fırlatmayan karşılıkları) ile her iki uca da erişim sağlar.

```
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);
    }
}
```

ArrayDeque'ı Stack Olarak Kullanmak

`Deque`, `push()`/`pop()` metotlarıyla bir YIĞIN (stack, LIFO -- last-in-first-out) gibi de kullanılabilir. Java'nın kendi `java.util.Stack` sınıfının javadoc'u, bu eski sınıf yerine `Deque`'ın (özellikle `ArrayDeque`'ın) kullanılmasını RESMİ OLARAK önerir -- çünkü `Stack`, `Vector`'ı genişletir ve bu yüzden gereksiz senkronizasyon yükü ile yığın kavramına uymayan index tabanlı metotlar (`insertElementAt()` gibi) miras alır.

```
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());
    }
}
```

ArrayDeque ile LinkedList Arasında Performans

`ArrayDeque` ve `LinkedList`, `Deque` olarak aynı işlemler için ikisi de teorik olarak O(1)'dir -- ama sabit faktörler (constant factors) farklıdır: `LinkedList`, her eleman için ayrı bir bağlantı nesnesi (node) tahsis eder, `ArrayDeque` ise dairesel bir dizi kullanarak bu ek yükten kaçınır.

```
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)");
    }
}
```

Gerçek ölçüm: 5 milyon `offer()` + `poll()` çiftinde `ArrayDeque` çoğu çalıştırmada `LinkedList` 'ten belirgin şekilde daha hızlı çıktı (örneğin ~40 ms'ye karşı ~55-60 ms), ama fark her çalıştırmada aynı oranda değildi -- bazı çalıştırmalarda ikisi birbirine çok yaklaştı. Bu, `LinkedList` 'in her eleman için ayrı bir nesne tahsis etmesinin garbage collector üzerinde değişken bir baskı yaratmasıyla tutarlı. Sonuç olarak `ArrayDeque` hiçbir çalıştırmada `LinkedList` 'ten daha yavaş ölçülmedi.

PriorityQueue: Sırayla Değil, Önceliğe Göre

`PriorityQueue`, elemanları eklenme sırasına göre DEĞİL, doğal sıralamaya (ya da verilen bir `Comparator`'a) göre işler -- her zaman en küçük (ya da `Comparator`'a göre "en öncelikli") eleman `peek()` / `poll()` ile önce çıkar. Ama dikkat: bu YALNIZCA `peek()` / `poll()` için geçerlidir -- `PriorityQueue`'yu doğrudan yazdırmak ya da `Iterator` ile dolaşmak, elemanları SIRALI göstermez.

```
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`'nun `toString()`'i (ya da doğrudan `Iterator` ile dolaşmak), elemanların sıralı görüneceği YANLIŞ izlenimini verebilir -- yukarıdaki örnekte gerçek çıktı bunu kanıtlıyor: `[10, 20, 40, 50, 30]`, sıralı DEĞİL. `PriorityQueue` içeride bir heap (yığın ağacı) kullanır -- yalnızca kökün (dizinin ilk elemanı) her zaman en küçük olduğu garanti edilir, geri kalanı için hiçbir sıra garantisi yoktur. Elemanları gerçekten sıralı almak için tek yol tekrar tekrar `poll()` çağırmaktır.

Collections Yardımcı Sınıfı

`Collections`, `Collectors`'a benzer şekilde (bkz. "Collectors" dersi), herhangi bir `Collection`/`List` üzerinde çalışan hazır statik metotlar sunan bir yardımcı sınıftır: `sort()`, `reverse()`, `shuffle()`, `max()`/`min()`, `frequency()` (bir değerin kaç kez geçtiğini sayar), `binarySearch()` (SIRALI bir listede O(log n) arama), ve `emptyList()`/`singletonList()`/`nCopies()` gibi küçük, özel amaçlı immutable koleksiyon üreten fabrika metotları.

```
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"));
    }
}
```

> `Collections.binarySearch()`'ün doğru çalışması için listenin ÖNCEDEN sıralanmış olması şarttır -- sıralanmamış bir listede çağırmak istisna fırlatmaz ama YANLIŞ bir sonuç döner (sessiz bir hata). Emin değilsen önce `Collections.sort()` çağır.

Best Practices

- **Yığın (stack) için `java.util.Stack` yerine `ArrayDeque`'ı `push()`/`pop()` ile kullanın** -- bu, Java'nın kendi resmi önerisidir.
- **Kuyruk/deque olarak `LinkedList` yerine varsayılan olarak `ArrayDeque`'ı tercih edin** -- neredeyse her zaman en az o kadar hızlı, genellikle daha hızlı, ve daha az bellek kullanır (her eleman için ayrı node nesnesi tahsis etmez).
- **"Kuyruk boş" gibi normal durumlar için `offer()`/`poll()`/`peek()` ailesini tercih edin**, `add()`/`remove()`/`element()` değil -- istisna fırlatmak/yakalamak normal akış kontrolü için pahalıdır.
- **`PriorityQueue`'yu doğrudan yazdırmaya ya da `Iterator` ile dolaşmaya güvenmeyin** -- sıralı çıktı istiyorsanız tekrar tekrar `poll()` çağırın.

Yaygın Hatalar

- **Boş bir kuyrukta `remove()`/`element()` çağırıp `NoSuchElementException` almak.** Normal bir "boş mu" kontrolü için `offer()`/`poll()`/`peek()` ailesi kullanılmalı.
- **`PriorityQueue`'nun `toString()`'inin ya da `Iterator`'ının sıralı olduğunu varsaymak.** Yalnızca `peek()`/`poll()` sıralama garantisi verir.
- **`Collections.binarySearch()`'ü sıralanmamış bir listede çağırmak.** İstisna fırlatmaz ama yanlış bir sonuç döner -- önce mutlaka `Collections.sort()` çağrılmalı.
- **Sık ekleme/çıkarma gereken bir senaryoda gereksiz yere `java.util.Stack` ya da `Vector` kullanmak.** Bunlar eski, senkronize sınıflardır -- modern kod `ArrayDeque`/`ArrayList` kullanmalı.

Özet, Cheat Sheet ve Terimler Sözlüğü

`Queue`, elemanları belirli bir sırayla (genellikle FIFO) işler; `Deque`, her iki uçtan da erişim sağlayarak hem kuyruk hem yığın olarak kullanılabilir. `ArrayDeque`, hem `LinkedList`'e (Deque olarak) hem `java.util.Stack`'e (yığın olarak) tercih edilen modern implementasyondur. `PriorityQueue`, elemanları önceliğe göre işler ama yalnızca `poll()`/`peek()` sıralama garantisi verir. `Collections` yardımcı sınıfı, herhangi bir liste/koleksiyon üzerinde çalışan hazır statik metotlar sunar.

Hızlı referans:

```
Queue<String> queue = new ArrayDeque<>();     // FIFO -- offer()/poll()/peek()
Deque<String> stack = new ArrayDeque<>();      // LIFO -- push()/pop()/peek()
Queue<Integer> pq = new PriorityQueue<>();      // önceliğe göre -- poll() sıralıdır,
toString() DEĞİL
Collections.sort(list);                            // yerinde sıralama
Collections.max(list); Collections.min(list);         // en büyük/en küçük
Collections.frequency(list, value);                     // kaç kez geçtiğini say
Collections.binarySearch(sortedList, value);              // O(log n) arama (ÖNCE sırala!)
```

Terimler Sözlüğü

Queue — Elemanları belirli bir sırayla (genellikle FIFO) işleyen bir koleksiyon arayüzü.

Deque — Her iki uçtan da ekleme/çıkarma yapılabilen, hem kuyruk hem yığın olarak kullanılabilen `Queue` alt arayüzü.

ArrayDeque — `Deque`'ın dairesel diziyle çalışan, `LinkedList`'e ve `java.util.Stack`'e tercih edilen implementasyonu.

PriorityQueue — Elemanları eklenme sırasına göre değil, önceliğe (doğal sıralama ya da `Comparator`) göre işleyen, heap tabanlı bir `Queue` implementasyonu.

Collections — Herhangi bir koleksiyon üzerinde çalışan hazır statik metotlar (`sort`, `reverse`, `max`, `binarySearch` vb.) sunan yardımcı sınıf.