Lists

Collections kategorisinin ilk topic'i: `List` arayüzü, `ArrayList` ile `LinkedList`'in gerçek bir ısıtılmış ölçümle karşılaştırılan performans farkı, `List.of()` / `Collections.unmodifiableList()` / `List.copyOf()` ile immutable listeler, `Iterator` / `ListIterator` ile güvenli dolaşma, `List.sort(Comparator)` ile sıralama, ve `subList()` / `toArray()`.

Başlangıç 20 dk

Son güncelleme: 6 Ekim 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/lists

Lists

Java Temelleri kategorisinde tekil değerleri, sabit sayıda alanları ve özel davranışları nasıl modelleyeceğini gördün. Ama gerçek programların çoğu, sayısı önceden bilinmeyen, çalışma zamanında büyüyüp küçülen veri toplulukları tutar -- bir alışveriş sepetindeki ürünler, bir formdan gelen hata mesajları, bir API'den dönen kayıtlar. Bu, Koleksiyonlar kategorisinin konusu; ilk durağımız da Java'nın en çok kullanılan koleksiyon tipi: `List`.

List Nedir?

`List<E>`, `java.util.Collection` arayüzünü genişleten bir arayüzdür ve iki temel garanti sunar: elemanlar **sıralıdır** (eklenme sırası korunur) ve **indekslidir** (her elemana `get(index)` ile doğrudan erişilebilir). `Set`'in aksine, aynı değer bir `List`'te birden fazla kez bulunabilir.

`List` bir arayüz olduğu için doğrudan örneklenemez; en sık kullanılan iki implementasyonu `ArrayList` ve `LinkedList`'tir. İkisi de aynı sözleşmeyi (contract) uygular ama içeride tamamen farklı veri yapıları kullanır -- bu farkın pratikte ne anlama geldiğini birazdan gerçek bir ölçümle göreceğiz.

Neden Var?

Java dizileri (array) sabit boyutludur -- bir `int[10]` oluşturduğunda, o dizi hep 10 elemanlıktır, ne bir eksik ne bir fazla. Ama gerçek dünyada eleman sayısı neredeyse hiç önceden bilinmez: kullanıcı sepete kaç ürün ekleyecek, bir sorgu kaç satır dönecek? `List`, bu problemi çözer -- `add()` / `remove()` ile dinamik olarak büyür/küçülür, dizinin sabit boyut kısıtını ortadan kaldırır.

Tarihçe

`List` arayüzü, Java 1.2 (1998) ile gelen **Collections Framework**'ün parçası olarak tanıtıldı -- o zamana kadar Java'da yalnızca eski, senkronize (ve bu yüzden yavaş) `Vector` sınıfı vardı. `ArrayList`, `Vector`'ın senkronizasyon yükü olmayan modern karşılığı olarak aynı fazda geldi. Java 5 (2004) jenerikleri (`List<E>`) ekleyerek tip güvenliğini kazandırdı; Java 9 (2017) ise `List.of()` ile değiştirilemez (immutable) liste oluşturmayı kısayol hâline getirdi.

Temel List İşlemleri

En sık kullanılan `List` metotları: `add()` (sona ekler), `get(index)` (okur), `set(index, value)` (üzerine yazar), `remove()` (değere ya da index'e göre siler), `size()`, `contains()`, `indexOf()`. Bir `List`'i for-each döngüsüyle dolaşmak da doğal olarak çalışır, çünkü `List` `Iterable`'ı genişletir.

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

public class ListBasicsExample {
    public static void main(String[] args) {
        List<String> fruits = new ArrayList<>();
        fruits.add("apple");
        fruits.add("pear");
        fruits.add("banana");
        fruits.add("apple"); // a List allows duplicate elements

        System.out.println("List: " + fruits);
        System.out.println("Size: " + fruits.size());
        System.out.println("index 0: " + fruits.get(0));
        System.out.println("Contains 'banana'? " + fruits.contains("banana"));
        System.out.println("First index of 'apple': " + fruits.indexOf("apple"));

        fruits.set(1, "cherry"); // overwrite index 1
        System.out.println("After set(1, cherry): " + fruits);

        fruits.remove("cherry"); // remove by value
        System.out.println("After remove(cherry): " + fruits);

        fruits.remove(0); // remove by index
        System.out.println("After remove(0): " + fruits);

        for (String fruit : fruits) {
            System.out.println("for-each: " + fruit);
        }
    }
}
```

> `remove()`'un iki aşırı yüklemesi (overload) vardır ve `List<Integer>` gibi kutulanmış (boxed) sayısal tiplerde karıştırılması kolaydır: `remove(int index)` index'e göre siler, `remove(Object o)` ise değere göre siler. `list.remove(2)` bir `List<Integer>` üzerinde çağrıldığında, `2` otomatik olarak `Integer`'a kutulanmaz -- doğrudan `int` olarak yorumlanır ve index 2'deki elemanı siler, değeri 2 olan elemanı değil. Değere göre silmek istiyorsan `list.remove(Integer.valueOf(2))` yazman gerekir.

ArrayList ve LinkedList: İki Farklı Implementasyon

`ArrayList`, içeride büyüyebilen bir diziyle (dynamic array) çalışır -- `get(index)` doğrudan bellek adresine atlar, bu yüzden **O(1)**'dir. `LinkedList` ise çift yönlü bağlı bir listedir (doubly-linked list) -- her eleman bir öncekine ve bir sonrakine işaret eder; `get(index)`'in belirli bir index'e ulaşması için baştan (ya da sondan, hangisi yakınsa) o index'e kadar **tek tek ilerlemesi** gerekir, yani **O(n)**'dir.

Tersi de doğru: `ArrayList`'in başına eleman eklemek (`add(0, x)`), sonraki tüm elemanları bir sağa kaydırmayı gerektirir -- **O(n)**. `LinkedList`'in başına eklemek ise sadece birkaç referansı güncellemektir -- **O(1)**.

```
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;

public class ArrayListVsLinkedListExample {
    public static void main(String[] args) {
        int size = 20_000;
        List<Integer> arrayList = new ArrayList<>();
        List<Integer> linkedList = new LinkedList<>();
        for (int i = 0; i < size; i++) {
            arrayList.add(i);
            linkedList.add(i);
        }

        int middle = size / 2;
        int warmupRounds = 3_000;
        int timedRounds = 3_000;

        // Warm-up: run both paths a lot BEFORE measuring, so the JIT can optimize both --
        // a single, un-warmed-up measurement can be misleading (whichever path runs first
        // can look unfairly slow).
        for (int i = 0; i < warmupRounds; i++) {
            arrayList.get(middle);
            linkedList.get(middle);
        }

        long arrayListStart = System.nanoTime();
        for (int i = 0; i < timedRounds; i++) {
            arrayList.get(middle);
        }
        long arrayListNanos = System.nanoTime() - arrayListStart;
```

```
        long linkedListStart = System.nanoTime();
        for (int i = 0; i < timedRounds; i++) {
            linkedList.get(middle);
        }
        long linkedListNanos = System.nanoTime() - linkedListStart;

        System.out.println("get(middle element), " + timedRounds + " times, a " + size + "-
element list:");
        System.out.println("  ArrayList:  " + (arrayListNanos / 1_000_000) + " ms");
        System.out.println("  LinkedList: " + (linkedListNanos / 1_000_000) + " ms");

        // Second measurement: inserting at the front (add(0, ...))
        List<Integer> arrayList2 = new ArrayList<>();
        List<Integer> linkedList2 = new LinkedList<>();
        int addRounds = 20_000;

        for (int i = 0; i < 2_000; i++) {
            arrayList2.add(0, i);
            linkedList2.add(0, i);
        }
        arrayList2.clear();
        linkedList2.clear();

        long arrayListAddStart = System.nanoTime();
        for (int i = 0; i < addRounds; i++) {
            arrayList2.add(0, i);
        }
        long arrayListAddNanos = System.nanoTime() - arrayListAddStart;

        long linkedListAddStart = System.nanoTime();
        for (int i = 0; i < addRounds; i++) {
            linkedList2.add(0, i);
        }
        long linkedListAddNanos = System.nanoTime() - linkedListAddStart;

        System.out.println();
        System.out.println("add(0, element), " + addRounds + " times (inserting at the
front):");
        System.out.println("  ArrayList:  " + (arrayListAddNanos / 1_000_000) + " ms");
        System.out.println("  LinkedList: " + (linkedListAddNanos / 1_000_000) + " ms");
    }
}
```

Bu örnek, ısıtılmış (warmed-up) gerçek bir ölçümle şunu doğruluyor: 20.000 elemanlık bir listede, ortadaki elemana 3.000 kez `get()` ile erişmek `ArrayList`'te ölçülemeyecek kadar hızlı (0 ms) iken `LinkedList`'te milisaniyeler alıyor (yaklaşık 48 ms) -- çünkü her çağrı listenin yarısını baştan taramak zorunda. Buna karşılık, listenin başına 20.000 kez eleman eklemek `LinkedList`'te göz açıp kapayana kadar (yaklaşık 1 ms) biterken `ArrayList`'te belirgin şekilde daha uzun sürüyor (yaklaşık 16-17 ms) -- her ekleme, o ana kadarki tüm elemanları kaydırmak zorunda.

> Pratikte neredeyse her zaman `ArrayList` doğru seçimdir -- rastgele erişim (`get(index)`) çok daha yaygın bir işlemdir ve modern donanımda bitişik bellek (contiguous memory) erişimi CPU önbelleği (cache) sayesinde ek bir hız avantajı da sağlar. `LinkedList`'i yalnızca gerçekten listenin başına/sonuna sık sık ekleme-çıkarma yapıyorsan (örneğin bir kuyruk/queue olarak) düşün.

Immutable List'ler: List.of(), Collections.unmodifiableList(), List.copyOf()

Bazen bir listenin hiç değişmemesini garanti etmek istersin -- örneğin sabit bir yapılandırma listesi. Java üç farklı immutable liste aracı sunar ve aralarındaki fark önemlidir: `List.of(...)` sıfırdan değiştirilemez bir liste oluşturur; `Collections.unmodifiableList(list)` var olan bir listenin değiştirilemez bir **görünümünü** (view) döner -- orijinal liste hâlâ değişirse görünüm de değişir; `List.copyOf(list)` ise tamamen bağımsız, ayrı bir immutable **kopya** oluşturur.

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

public class ListOfImmutableExample {
    public static void main(String[] args) {
        List<String> immutable = List.of("red", "green", "blue");
        System.out.println("List.of(): " + immutable);

        try {
            immutable.add("yellow");
        } catch (UnsupportedOperationException e) {
            System.out.println("add() on a List.of() result: " + e.getClass().
getSimpleName());
        }

        try {
            immutable.set(0, "black");
        } catch (UnsupportedOperationException e) {
            System.out.println("set() on a List.of() result: " + e.getClass().
getSimpleName());
        }

        // Collections.unmodifiableList(): an UNMODIFIABLE "view" of an existing list
        List<String> mutable = new ArrayList<>(List.of("a", "b"));
        List<String> readOnlyView = Collections.unmodifiableList(mutable);
        try {
            readOnlyView.add("c");
        } catch (UnsupportedOperationException e) {
            System.out.println("add() on unmodifiableList(): " + e.getClass().
getSimpleName());
        }

        // But watch out: unmodifiableList() is just a VIEW, the original list can still
change
        mutable.add("c");
        System.out.println("The view changes when the original list changes: " +
readOnlyView);

        // List.copyOf(): creates a completely independent, separate immutable COPY
        List<String> independentCopy = List.copyOf(mutable);
        mutable.add("d");
        System.out.println("Original list changed: " + mutable);
        System.out.println("List.copyOf() copy was NOT affected: " + independentCopy);
    }
}
```

> `Collections.unmodifiableList()`'in döndürdüğü listenin "salt okunur" olması, orijinal listenin de değişmeyeceği anlamına gelmez -- sadece görünüm üzerinden değiştirme engellenir. Gerçekten bağımsız, değişmeyen bir kopya istiyorsan `List.copyOf()` kullanmalısın.

Iterator ve ListIterator

Bir `List`'i dolaşırken SIRASINDA elemanları silmek/eklemek istersen, doğrudan `List.remove()` çağırmak `ConcurrentModificationException` fırlatır -- çünkü for-each döngüsü arka planda bir `Iterator` kullanır ve `Iterator`, listenin "beklenmedik" şekilde değiştiğini fark eder. Doğru yol, `Iterator.remove()` metodunu kullanmaktır -- bu, iterator'ın kendi iç sayacını da günceller. `ListIterator`, `Iterator`'ın genişletilmiş hâlidir: hem ileri hem geri gidebilir (`hasPrevious()` / `previous()`) ve dolaşırken `set()` / `add()` de destekler.

```
import java.util.ArrayList;
import java.util.ConcurrentModificationException;
import java.util.Iterator;
import java.util.List;
import java.util.ListIterator;

public class IteratorExample {
    public static void main(String[] args) {
        List<Integer> numbers = new ArrayList<>(List.of(1, 2, 3, 4, 5, 6));

        // Safe removal with Iterator: use Iterator.remove() instead of calling
        // List.remove() DURING a for-each loop.
        Iterator<Integer> it = numbers.iterator();
        while (it.hasNext()) {
            int value = it.next();
            if (value % 2 == 0) {
                it.remove(); // safe -- the iterator updates its own internal bookkeeping
            }
        }
        System.out.println("Even numbers removed with Iterator.remove(): " + numbers);

        // ListIterator: unlike Iterator, it can move in BOTH directions (hasPrevious
/previous)
        // and also supports add()/set().
        List<String> letters = new ArrayList<>(List.of("a", "b", "c"));
        ListIterator<String> listIt = letters.listIterator();
        while (listIt.hasNext()) {
            String value = listIt.next();
            listIt.set(value.toUpperCase());
        }
        System.out.println("Converted to uppercase with ListIterator.set(): " + letters);

        while (listIt.hasPrevious()) {
            System.out.println("going backwards: " + listIt.previous());
        }

        // REAL ERROR: calling List.remove() directly DURING a for-each loop
        List<Integer> unsafe = new ArrayList<>(List.of(10, 20, 30, 40));
        try {
            for (Integer value : unsafe) {
                if (value == 20) {
                    unsafe.remove(value); // throws ConcurrentModificationException
                }
            }
        } catch (ConcurrentModificationException e) {
            System.out.println("List.remove() during a for-each loop: " + e.getClass().
getSimpleName());
        }
    }
}
```

Sıralama: List.sort() ve Comparator

`List.sort(Comparator)`, listeyi **yerinde** (in-place) sıralar -- yeni bir liste döndürmez, var olanı değiştirir. Argüman olarak `Comparator.naturalOrder()` (doğal sıralama), `Comparator.`

`reverseOrder()` (ters), ya da `Comparator.comparing(...)` ile bir nesnenin belirli bir alanına göre özel bir sıralama verilebilir. `Collections.sort(list)`, `List.sort()`'tan (Java 8) önceki eski yoldur -- hâlâ çalışır ama artık `List.sort()` tercih edilir.

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

public class SortingExample {
    record Person(String name, int age) {
    }

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

        // List.sort(): sorts in place; Comparator.naturalOrder() for natural ordering
        numbers.sort(Comparator.naturalOrder());
        System.out.println("Natural order: " + numbers);

        numbers.sort(Comparator.reverseOrder());
        System.out.println("Reversed order: " + numbers);

        // Collections.sort(): the old way, predating List.sort() (pre-Java 8), still works
        List<String> words = new ArrayList<>(List.of("banana", "apple", "kiwi", "pear"));
        Collections.sort(words);
        System.out.println("Collections.sort(): " + words);

        // Comparator.comparing() + thenComparing(): sorting objects by a field
        List<Person> people = new ArrayList<>(List.of(
                new Person("Alice", 30),
                new Person("Bob", 25),
                new Person("Alice", 22)
        ));

        people.sort(Comparator.comparing(Person::name).thenComparing(Person::age));
        System.out.println("By name, then age: " + people);

        people.sort(Comparator.comparingInt(Person::age).reversed());
        System.out.println("By age, descending: " + people);
    }
}
```

> `Comparator.comparing(Person::name).thenComparing(Person::age)` gibi zincirleme, "önce isme göre sırala, isimler eşitse yaşa göre sırala" anlamına gelir -- birden fazla alana göre sıralama gerektiğinde elle yazılmış bir `compareTo()`'dan çok daha okunabilirdir.

subList() ve toArray()

`subList(from, to)`, orijinal listenin `from` (dahil) ile `to` (hariç) arasındaki bir **görünümünü** (view) döner -- bağımsız bir kopya değildir. Bu görünüm üzerinde yapılan değişiklikler (ekleme, silme, `set()`) orijinal listeye de yansır. `toArray()`, bir `List`'i diziye çevirmenin üç

yolunu sunar: argümansız hâli tip bilgisini kaybeden bir `Object[]` döner, `toArray(new String[0])` ya da (Java 11+) `toArray(String[]::new)` ise doğru tipte bir dizi üretir.

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

public class SubListAndToArrayExample {
    public static void main(String[] args) {
        List<Integer> numbers = new ArrayList<>(List.of(0, 1, 2, 3, 4, 5, 6, 7, 8, 9));

        // subList(from, to): from inclusive, to exclusive -- NOT an independent copy, it's
        // a "view" of the original list.
        List<Integer> middle = numbers.subList(3, 6);
        System.out.println("subList(3, 6): " + middle);

        // Changes made through the subList also change the ORIGINAL list
        middle.set(0, 999);
        System.out.println("Original list after set(0, 999) via subList: " + numbers);

        middle.clear();
        System.out.println("Original list after clear() via subList: " + numbers);

        // toArray(): two ways to convert a List to an array
        List<String> letters = List.of("x", "y", "z");

        Object[] rawArray = letters.toArray();
        System.out.println("toArray() (Object[]): " + rawArray.length + " elements");

        String[] typedArray = letters.toArray(new String[0]);
        System.out.println("toArray(new String[0]) (String[]): " + String.join(", ",
typedArray));

        // toArray(IntFunction) -- Java 11+, a type-safe array without specifying the size
        String[] typedArray2 = letters.toArray(String[]::new);
        System.out.println("toArray(String[]::new): " + String.join(", ", typedArray2));
    }
}
```

> `subList()`'in bir kopya değil bir görünüm olması sık karşılaşılan bir tuzaktır -- görünüm üzerinde `clear()` çağırmak, orijinal listedeki o aralığı da siler. Bağımsız bir alt-liste istiyorsan `new ArrayList<>(numbers.subList(3, 6))` ile açıkça kopyalamalısın.

Best Practices

- **Varsayılan olarak `ArrayList` kullanın**, yalnızca listenin başına/sonuna sık sık ekleme-çıkarma yapıyorsanız `LinkedList`'i (ya da daha iyisi, `ArrayDeque`'ı) düşünün.
- **Değişmeyecek bir liste için `List.of()`'u tercih edin** -- hem niyeti nettir hem de yanlışlıkla değiştirilmeyi derleme zamanı değil ama en azından ilk çalıştırmada `UnsupportedOperationException` ile yakalar.

- **Dolaşırken silme/ekleme gerekiyorsa `Iterator.remove()` / `ListIterator` kullanın**, doğrudan `List.remove()` çağırmayın.
- **Birden fazla alana göre sıralama için `Comparator.comparing(...).thenComparing(...)` zincirini kullanın** -- elle yazılmış `compareTo()`'dan daha az hataya açıktır.

Yaygın Hatalar

- **`List<Integer>` üzerinde `remove(int)` ile `remove(Object)`'i karıştırmak.** `list.remove(2)`, index 2'yi siler; değeri 2 olan elemanı silmek için `list.remove(Integer.valueOf(2))` gerekir.
- **for-each döngüsü sırasında doğrudan `List.remove()` çağırmak.** Bu, `ConcurrentModificationException` fırlatır -- `Iterator.remove()` kullanılmalı.
- **`subList()`'in bağımsız bir kopya olduğunu sanmak.** Bir görünümdür; üzerindeki değişiklikler orijinal listeye yansır.
- **Rastgele erişimin (`get(index)`) yoğun olduğu bir senaryoda `LinkedList` seçmek.** `ArrayList`'in O(1) erişimine karşı `LinkedList`'in O(n) erişimi, büyük listelerde ölçülebilir bir performans farkına yol açar.

Özet, Cheat Sheet ve Terimler Sözlüğü

`List<E>`, sıralı ve indeksli, tekrar eden elemanlara izin veren bir koleksiyon arayüzüdür. `ArrayList` rastgele erişimde (O(1)) hızlıdır, `LinkedList` ise listenin uçlarına ekleme/çıkarmada (O(1)) hızlıdır. `List.of()` / `List.copyOf()` değiştirilemez listeler oluşturur, `Collections.unmodifiableList()` ise mevcut bir listenin salt okunur bir görünümünü döner. Dolaşırken güvenli değişiklik için `Iterator` / `ListIterator`, sıralama için `List.sort(Comparator)` kullanılır.

Hızlı referans:

```
List<String> list = new ArrayList<>();      // dinamik array tabanlı, varsayılan seçim
List<String> linked = new LinkedList<>();    // uçlara ekleme/çıkarma ağırlıklıysa
List<String> immutable = List.of("a", "b");  // değiştirilemez, sıfırdan
List<String> copy = List.copyOf(list);       // değiştirilemez, bağımsız kopya
List<String> view = Collections.unmodifiableList(list); // değiştirilemez GÖRÜNÜM
list.sort(Comparator.comparing(String::length));         // yerinde sıralama
List<String> part = new ArrayList<>(list.subList(1, 3)); // bağımsız alt-liste kopyası
```

Terimler Sözlüğü

List — Sıralı ve indeksli, tekrar eden elemanlara izin veren bir `Collection` alt arayüzü.

ArrayList — `List`'in dinamik diziyle çalışan, rastgele erişimde O(1) olan implementasyonu.

LinkedList — `List`'in çift yönlü bağlı listeyle çalışan, uçlara ekleme/çıkarmada O(1) olan implementasyonu.

View (görünüm) — `subList()` / `unmodifiableList()` gibi metotların döndürdüğü, orijinal veriyle bağlantısını koruyan (bağımsız kopya olmayan) bir nesne.

ConcurrentModificationException — Bir koleksiyon `Iterator` ile dolaşılırken, o iterator'ın dışından değiştirildiğinde fırlatılan istisna.