Arrays

Java Basics kategorisinin ikinci topic'i: dizilerin sabit boyutlu, bitişik bellek yapısı, çok boyutlu diziler (jagged array dahil), `Arrays` yardımcı sınıfı (sort/binarySearch/equals/fill/copyOf), array covariance ve `ArrayStoreException` tuzağı, `Arrays.asList()`'in görünüm (view) olması, ve varargs.

Başlangıç 20 dk

Son güncelleme: 21 Ekim 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/arrays

Arrays

Java Temelleri kategorisinin ikinci konusu `Array` (dizi) -- `String` gibi çok temel ama altında dilin en eski, en düşük seviyeli veri yapısını barındırır. Diziler, `ArrayList` / `HashMap` gibi tüm koleksiyonların İÇİNDE kullanılan yapı taşıdır; onları anlamak, Koleksiyonlar kategorisindeki `O(1)` erişim gibi performans iddialarının GERÇEKTE nereden geldiğini de netleştirir.

Array Nedir?

Bir array (dizi), AYNI TİPTEN sabit sayıda elemanı, bellekte ARDIŞIK (contiguous) bir blokta tutan bir veri yapısıdır. Boyutu oluşturulduğu anda belirlenir ve bir daha ASLA değişmez -- bu, `ArrayList` gibi "dinamik boyutlu" koleksiyonlardan en temel farkıdır. Java'da diziler, `int[]` / `String[]` gibi ilkel (primitive) görünseler de aslında birer NESNEdir (`Object`'ten türer) -- bu yüzden `.length` bir alan (field) olarak erişilir, `String.length()`'in aksine parantez GEREKTİRMEZ.

Neden Var?

Bir index'e göre eleman okumak/yazmak, bitişik bellek yerleşimi sayesinde donanım düzeyinde doğrudan bir adres hesabıyla yapılabilir -- bu, dizileri O(1) index erişimi için mümkün olan EN HIZLI yapı yapar. `ArrayList` gibi üst düzey koleksiyonların O(1) `get(index)` performansı iddiası da tam olarak buradan gelir: `ArrayList` içeride bir array SARAR (wrap eder) ve büyüme gerektiğinde daha büyük yeni bir array'e kopyalar. Dizileri doğrudan anlamak, bu üst düzey koleksiyonların neden hızlı ya da yavaş olduğunu da açıklar.

Tarihçe

Diziler Java'nın 1.0 sürümünden (1996) beri dilin çekirdek bir parçasıdır -- `String` gibi en eski yapılardan biri. `Arrays` yardımcı sınıfı (statik `sort()` / `binarySearch()` / `equals()` metotlarıyla) Java 1.2 (1998) ile Collections Framework'le birlikte geldi. Varargs (`Type... args` , bir dizi parametresinin çağrı tarafında rahat kullanılmasını sağlayan sözdizimi) Java 5 (2004) ile eklendi. Java 8 (2014), `Arrays.stream()` ile dizileri doğrudan Stream API'ye bağladı (bkz. "Stream API Temelleri" dersi).

Temel Kullanım: Oluşturma, Erişim, Varsayılan Değerler

Bir dizi `new Type[boyut]` ile ya da bir LİTERAL (`{1, 2, 3}`) ile oluşturulabilir. İlkel tipli bir dizinin (`int[]` gibi) başlatılmamış elemanları varsayılan olarak `0` / `false` gibi bir değer alır; referans tipli bir dizinin (`String[]` gibi) elemanları ise `null` olur. Sınırların dışına çıkmak (`array[10]` , dizi boyutu 5 iken) SESSİZCE yanlış bir şey döndürmez -- gerçek bir `ArrayIndexOutOfBoundsException` fırlatır.

```
import java.util.Arrays;

public class ArrayBasicsExample {
    public static void main(String[] args) {
        // A fixed-size array of 5 ints -- size is decided at creation and can
        // NEVER change afterward.
        int[] numbers = new int[5];
        System.out.println("Freshly created int[5]: " + Arrays.toString(numbers));
        System.out.println("(uninitialized elements default to 0 for numeric types)");

        // An array LITERAL -- size and content given at once.
        String[] fruits = {"apple", "banana", "cherry"};
        System.out.println("Array literal: " + Arrays.toString(fruits));
        System.out.println("(uninitialized elements of a reference-type array default to
null)");

        // Index-based access -- O(1), reading or writing by position.
        numbers[0] = 10;
        numbers[1] = 20;
        numbers[4] = 50;
        System.out.println("After setting a few indices: " + Arrays.toString(numbers));
        System.out.println("numbers[1]: " + numbers[1]);

        // .length is a FIELD, not a method (unlike String.length() or List.size()).
        System.out.println("fruits.length: " + fruits.length);

        // Arrays are OBJECTS in Java -- printing one directly (without Arrays.
        // toString()) does NOT show its contents, just a type + hashcode string.
        System.out.println("Printing the array directly (not useful!): " + numbers);

        // Going out of bounds throws a real runtime exception, it does NOT
        // silently return null/0 or wrap around.
        try {
            int oops = numbers[10];
            System.out.println("unreachable: " + oops);
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Caught: " + e.getClass().getSimpleName() + " -- " + e.
getMessage());
        }
    }
}
```

> Bir diziyi doğrudan `System.out.println(array)` ile yazdırmak, içeriğini GÖSTERMEZ -- `[I@7ea987ac` gibi bir "tip@hashcode" metni verir, çünkü `Object.toString()`'in varsayılan davranışı budur. İçeriği görmek için her zaman `Arrays.toString()` (tek boyutlu) ya da `Arrays.deepToString()` (çok boyutlu) kullanılmalı.

Çok Boyutlu Diziler

Java'da bir "2 boyutlu dizi" aslında bir DİZİLER DİZİSİDİR (array of arrays) -- her "satır" kendi bağımsız bir dizi nesnesidir. Bu yüzden satırların uzunluğu FARKLI olabilir (buna "jagged array" -- düzensiz/pürüzlü dizi denir); dikdörtgen bir grid de yalnızca özel bir durumdur (tüm satırlar aynı uzunlukta).

```
import java.util.Arrays;

public class MultiDimensionalArrayExample {
    public static void main(String[] args) {
        // A "2D array" in Java is really an array OF arrays -- here, a
        // rectangular 3x3 grid.
        int[][] grid = {
                {1, 2, 3},
                {4, 5, 6},
                {7, 8, 9}
        };
        System.out.println("grid[1][2] (row 1, column 2): " + grid[1][2]);

        // Arrays.toString() does NOT recurse into nested arrays -- it just
        // prints each row's default "type@hashcode" string. Arrays.deepToString()
        // is the one that actually recurses.
        System.out.println("Arrays.toString(grid) (WRONG tool for nested arrays): " +
Arrays.toString(grid));
        System.out.println("Arrays.deepToString(grid) (correct tool): " + Arrays.
deepToString(grid));

        // Because each "row" is its own independent array object, rows don't
        // have to be the same length -- this is a "jagged" array.
        int[][] jagged = new int[3][];
        jagged[0] = new int[]{1};
        jagged[1] = new int[]{1, 2, 3};
        jagged[2] = new int[]{1, 2};
        System.out.println("Jagged array: " + Arrays.deepToString(jagged));
        for (int i = 0; i < jagged.length; i++) {
            System.out.println("  row " + i + " length: " + jagged[i].length);
        }

        // A 3D array works the same way, one more level of nesting.
        int[][][] cube = new int[2][2][2];
        cube[1][1][1] = 42;
        System.out.println("cube[1][1][1]: " + cube[1][1][1]);
        System.out.println("cube[0][0][0] (never set, still default 0): " + cube[0][0][0]);
    }
}
```

> `Arrays.toString()`, iç içe (nested) dizilerde YANLIŞ araçtır -- her satırı yine `[I@...` gibi bir hashcode metniyle gösterir, İÇERİĞE inmez. Çok boyutlu bir diziyi düzgün yazdırmak için `Arrays.deepToString()` kullanılmalı.

Arrays Yardımcı Sınıfı

`Arrays`, `Collections`'a benzer şekilde (bkz. "Queues & Collections Utility" dersi), diziler üzerinde çalışan hazır statik metotlar sunan bir yardımcı sınıftır: `sort()` (yerinde sıralar), `binarySearch()` (SIRALI bir dizide O(log n) arama), `equals()` (İÇERİK karşılaştırması -- `==`'nin aksine), `fill()` (tüm elemanları aynı değere ayarlar), ve `copyOf()` / `copyOfRange()` (yeni bir dizi olarak kopyalar).

```
import java.util.Arrays;

public class ArraysUtilityExample {
    public static void main(String[] args) {
        int[] numbers = {5, 3, 8, 1, 9, 2};
        System.out.println("Original: " + Arrays.toString(numbers));

        // sort() sorts IN PLACE -- it doesn't return a new array.
        Arrays.sort(numbers);
        System.out.println("After Arrays.sort(): " + Arrays.toString(numbers));

        // binarySearch() requires a SORTED array -- O(log n) lookup.
        System.out.println("Arrays.binarySearch(numbers, 8): index " + Arrays.binarySearch
(numbers, 8));

        // equals() compares CONTENT (element by element) -- this is the array
        // equivalent of the == vs equals() String lesson: == on two arrays
        // compares references, Arrays.equals() compares values.
        int[] copy = Arrays.copyOf(numbers, numbers.length);
        System.out.println("numbers == copy (reference): " + (numbers == copy));
        System.out.println("Arrays.equals(numbers, copy) (content): " + Arrays.equals
(numbers, copy));

        // fill() sets every element to the same value.
        int[] filled = new int[4];
        Arrays.fill(filled, 7);
        System.out.println("Arrays.fill(new int[4], 7): " + Arrays.toString(filled));

        // copyOf() with a length LONGER than the original pads with default
        // values (0 for int); SHORTER truncates.
        int[] longer = Arrays.copyOf(numbers, 8);
        int[] shorter = Arrays.copyOf(numbers, 3);
        System.out.println("copyOf(numbers, 8) (padded with 0): " + Arrays.toString
(longer));
        System.out.println("copyOf(numbers, 3) (truncated): " + Arrays.toString(shorter));

        // copyOfRange() extracts a sub-array (end index is EXCLUSIVE, just like
        // String.substring()).
        int[] range = Arrays.copyOfRange(numbers, 1, 4);
        System.out.println("copyOfRange(numbers, 1, 4): " + Arrays.toString(range));
    }
}
```

Beklenen Çıktı

```
Original: [5, 3, 8, 1, 9, 2]
After Arrays.sort(): [1, 2, 3, 5, 8, 9]
Arrays.binarySearch(numbers, 8): index 4
numbers == copy (reference): false
Arrays.equals(numbers, copy) (content): true
Arrays.fill(new int[4], 7): [7, 7, 7, 7]
copyOf(numbers, 8) (padded with 0): [1, 2, 3, 5, 8, 9, 0, 0]
copyOf(numbers, 3) (truncated): [1, 2, 3]
copyOfRange(numbers, 1, 4): [2, 3, 5]
```

İki diziyi `==` ile karşılaştırmak REFERANS'ı karşılaştırır (aynı bellek adresi mi), İÇERİĞİ değil -- tıpkı `String`'deki `==` vs `equals()` tuzağı gibi (bkz. "String" dersi). İki dizinin AYNI elemanlara sahip olup olmadığını kontrol etmek için `Arrays.equals()` kullanılmalı.

Array Covariance: Derleyicinin Gözden Kaçırdığı Bir Tuzak

Java dizileri KOVARYANTTIR (covariant): `Integer`, `Number`'ı genişlettiği için bir `Integer[]`, bir `Number[]` değişkenine atanabilir. Ama bu, bir tuzak kapısı açar: derleyici, `Number[]` üzerinden bir `Double` YAZMAYA izin verir (çünkü `Double` de bir `Number`'dır) -- ama dizinin GERÇEK çalışma zamanı tipi hâlâ `Integer[]`'dir, bu yüzden bu yazma işlemi derleme zamanında DEĞİL, ÇALIŞMA ZAMANINDA (`ArrayStoreException` ile) patlar.

```
public class ArrayCovarianceExample {
    public static void main(String[] args) {
        // Java arrays are COVARIANT: since Integer extends Number, an
        // Integer[] can be assigned to a Number[] variable.
        Integer[] integers = {1, 2, 3};
        Number[] numbers = integers; // legal -- Integer[] IS-A Number[]
        System.out.println("numbers[0] via the Number[] view: " + numbers[0]);

        // The DANGER: the compiler allows storing a Double into `numbers`
        // (since Double is also a Number), but the array's REAL runtime type is
        // still Integer[] -- so this fails, not at compile time, but at RUNTIME.
        try {
            numbers[1] = 3.14; // compiles fine (Double IS-A Number)...
            System.out.println("unreachable");
        } catch (ArrayStoreException e) {
            System.out.println("Caught: " + e.getClass().getSimpleName() + " -- " + e.
getMessage());
        }

        // This is exactly the kind of bug that array covariance can hide until
        // runtime -- generics (List<T>) deliberately do NOT allow this: a
        // List<Integer> cannot be assigned to a List<Number> variable at all,
        // so the equivalent mistake is caught at COMPILE time instead.
        System.out.println();
        System.out.println("Arrays: covariant, unsafe writes fail at RUNTIME
(ArrayStoreException).");
        System.out.println("Generics (List<T>): invariant, the equivalent mistake fails at
COMPILE time.");
    }
}
```

> Generic'ler (`List<T>`) BİLEREK bu tuzağı önler: bir `List<Integer>` , bir `List<Number>` değişkenine ATANAMAZ (invariant'tır) -- yani array covariance'ın sakladığı bu tarz bir hata, generic koleksiyonlarda ÇALIŞMA ZAMANI yerine DERLEME ZAMANINDA yakalanır. Bu, "neden `List<Number> list = new ArrayList<Integer>();` derlenmiyor?" sorusunun cevabıdır.

Arrays vs Collections: Arrays.asList() ve Dönüşümler

`Arrays.asList()` , verilen diziyi KOPYALAMAZ -- orijinal diziyi SABİT BOYUTLU bir `List` GÖRÜNÜMÜYLE (view) sarar. Bu görünüm üzerinden yazmak orijinal diziyi de değiştirir (ve tersi de geçerlidir); sabit boyutlu olduğu için `add()` / `remove()` desteklenmez (`UnsupportedOperationException` fırlatır), yalnızca `set()` (mevcut bir index'i değiştirmek) çalışır. Gerçekten bağımsız, yeniden boyutlanabilir bir liste için bu görünümü `new ArrayList<>(...)` içine SARMAK gerekir.

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

public class ArraysVsCollectionsExample {
    public static void main(String[] args) {
        String[] fruitsArray = {"apple", "banana", "cherry"};

        // Arrays.asList() does NOT copy -- it wraps the ORIGINAL array in a
        // fixed-size List VIEW. Writing through the list writes through to the
        // array, and vice versa.
        List<String> fruitsView = Arrays.asList(fruitsArray);
        System.out.println("View before: " + fruitsView);
        fruitsArray[0] = "avocado";
        System.out.println("View after modifying the array directly: " + fruitsView);

        // Because the view is backed by a FIXED-SIZE array, add()/remove() are
        // NOT supported -- only set() (replacing an existing index) is.
        try {
            fruitsView.add("date");
            System.out.println("unreachable");
        } catch (UnsupportedOperationException e) {
            System.out.println("Caught: " + e.getClass().getSimpleName()
                    + " -- Arrays.asList() does not support add()/remove()");
        }

        // To get a REAL, independent, resizable list, wrap the view in a new
        // ArrayList.
        List<String> realList = new ArrayList<>(Arrays.asList(fruitsArray));
        realList.add("date");
        System.out.println("Independent ArrayList (add() works): " + realList);

        // The reverse conversion: List -> array, with toArray(new String[0]).
        String[] backToArray = realList.toArray(new String[0]);
        System.out.println("List.toArray(new String[0]): " + Arrays.toString(backToArray));

        System.out.println();
        System.out.println("Array: fixed size, can hold primitives, index access is the
fastest option.");
        System.out.println("List (e.g. ArrayList): resizable, only reference types, richer
API.");
    }
}
```

Beklenen Çıktı

```
View before: [apple, banana, cherry]
View after modifying the array directly: [avocado, banana, cherry]
Caught: UnsupportedOperationException -- Arrays.asList() does not support add()/remove()
Independent ArrayList (add() works): [avocado, banana, cherry, date]
List.toArray(new String[0]): [avocado, banana, cherry, date]

Array: fixed size, can hold primitives, index access is the fastest option.
List (e.g. ArrayList): resizable, only reference types, richer API.
```

Varargs: Diziyi Rahat Bir Çağrı Sözdizimiyle Kullanmak

Varargs (`Type... isim`), bir metodun ÇAĞIRAN tarafın sıfır, bir ya da birçok argüman geçmesine izin vermesini sağlayan bir sözdizimidir -- metodun İÇİNDE, bu parametre basitçe normal bir dizidir. Varargs parametresi bir metot imzasında yalnızca SON parametre olabilir.

```
public class VarargsExample {
    public static void main(String[] args) {
        // A varargs parameter (Type... name) lets the CALLER pass zero, one, or
        // many arguments -- inside the method, it is simply an array.
        System.out.println("sum(): " + sum());
        System.out.println("sum(1): " + sum(1));
        System.out.println("sum(1, 2, 3, 4): " + sum(1, 2, 3, 4));

        // Passing an actual array works exactly the same way -- varargs IS an
        // array parameter, just with convenient call-site syntax.
        int[] values = {10, 20, 30};
        System.out.println("sum(values) (passing an int[] directly): " + sum(values));

        // A varargs parameter must be the LAST parameter in the method's
        // signature -- printLabeled() below shows a normal parameter followed by
        // varargs.
        printLabeled("Scores", 90, 85, 77);
        printLabeled("Empty case");

        // System.out.printf() and String.format() themselves use varargs
        // (Object... args) -- that's how they accept any number of placeholders.
        System.out.printf("printf is varargs too: %s scored %d%n", "Alice", 95);
    }

    private static int sum(int... numbers) {
        int total = 0;
        for (int n : numbers) {
            total += n;
        }
        return total;
    }

    private static void printLabeled(String label, int... values) {
        System.out.print(label + ": ");
        if (values.length == 0) {
            System.out.println("(no values)");
            return;
        }
        for (int v : values) {
            System.out.print(v + " ");
        }
        System.out.println();
    }
}
```

Beklenen Çıktı

```
sum(): 0
sum(1): 1
sum(1, 2, 3, 4): 10
sum(values) (passing an int[] directly): 60
Scores: 90 85 77
Empty case: (no values)
printf is varargs too: Alice scored 95
```

`System.out.printf()` ve `String.format()`'in kendisi de varargs kullanır (`Object... args`) -- bu sayede `%s` / `%d` gibi istediğiniz sayıda yer tutucuyu tek bir metotla karşılayabilirler (bkz. "String" dersi).

Best Practices

- **Bir dizinin içeriğini yazdırmak için her zaman `Arrays.toString()` (tek boyutlu) ya da `Arrays.deepToString()` (çok boyutlu) kullanın**, doğrudan `System.out.println(array)` değil -- bu yalnızca anlamsız bir "tip@hashcode" metni verir.
- **İki dizinin içeriğini karşılaştırmak için `Arrays.equals()` kullanın**, `==` değil -- `==` yalnızca referans karşılaştırır, tıpkı `String`'deki gibi.
- **Boyutu programın çalışması sırasında değişecek bir koleksiyona ihtiyacınız varsa `ArrayList` gibi bir koleksiyon kullanın**, dizi değil -- diziler oluşturulduktan sonra yeniden boyutlandırılamaz.
- **`Arrays.asList()`'in bir GÖRÜNÜM olduğunu, kopya olmadığını unutmayın** -- bağımsız, yeniden boyutlanabilir bir liste istiyorsanız `new ArrayList<>(Arrays.asList(...))` ile sarın.

Yaygın Hatalar

- **Bir diziyi doğrudan yazdırıp `[I@7ea987ac` gibi anlamsız bir çıktı almak.** `Arrays.toString()` / `Arrays.deepToString()` kullanılması gerekiyordu.
- **İki diziyi `==` ile karşılaştırıp içerik aynı olsa bile `false` almak.** İçerik karşılaştırması için `Arrays.equals()` gerekir.
- **`Arrays.asList()`'in döndürdüğü listeye `add()` / `remove()` çağırıp `UnsupportedOperationException` almak.** Bu görünüm sabit boyutludur -- gerçek bir `ArrayList`'e ihtiyaç varsa açıkça sarmalanmalı.
- **Array covariance'ın (bir `Integer[]`'in bir `Number[]` değişkenine atanabilmesinin) güvenli olduğunu varsaymak.** Bu, derleme zamanında yakalanmayan bir `ArrayStoreException` riski taşır -- generic koleksiyonlar bu riski taşımaz.

Özet, Cheat Sheet ve Terimler Sözlüğü

Bir array (dizi), aynı tipten sabit sayıda elemanı bitişik bellekte tutan, O(1) index erişimi sağlayan temel bir veri yapısıdır -- `ArrayList` gibi üst düzey koleksiyonların içinde kullanılan yapı taşıdır. `Arrays` yardımcı sınıfı `sort()` / `binarySearch()` / `equals()` / `fill()` / `copyOf()` gibi statik metotlar sunar. Diziler kovaryanttır ve bu yüzden bazı hatalar (generic koleksiyonların aksine) yalnızca çalışma zamanında yakalanır; `Arrays.asList()` orijinal diziyi saran sabit boyutlu bir görünümdür, kopya değildir.

Hızlı referans:

```
int[] numbers = new int[5];                 // sabit boyutlu, varsayılan 0'larla
String[] fruits = {"apple", "banana"};         // literal ile oluşturma
Arrays.toString(numbers);                        // içeriği düzgün yazdırmak için
Arrays.sort(numbers);                              // yerinde sıralama
Arrays.equals(a, b);                                 // İÇERİK karşılaştırması (== değil)
Arrays.copyOf(numbers, 10);                            // yeni, daha büyük bir kopya
List<String> view = Arrays.asList(fruits);                // SABİT BOYUTLU görünüm, kopya
değil
List<String> real = new ArrayList<>(Arrays.asList(fruits)); // bağımsız, resizable liste
```

Terimler Sözlüğü

Array (Dizi) — Aynı tipten sabit sayıda elemanı, bellekte ardışık bir blokta tutan temel bir veri yapısı.

Jagged Array — Satırları (alt dizileri) farklı uzunluklarda olabilen çok boyutlu bir dizi.

Array Covariance — Bir alt tip dizisinin (`Integer[]`) bir üst tip dizi değişkenine (`Number[]`) atanabilmesi; potansiyel olarak çalışma zamanı hatasına (`ArrayStoreException`) yol açabilir.

Varargs — Bir metodun çağrı tarafında sıfır ya da birçok argüman geçirilmesine izin veren, içeride basit bir diziye dönüşen sözdizimi (`Type... isim`).

Arrays — Diziler üzerinde çalışan hazır statik metotlar (`sort` , `equals` , `fill` , `copyOf` vb.) sunan yardımcı sınıf.