Arrays

The second topic in the Java Basics category: arrays' fixed-size, contiguous-memory structure, multi-dimensional arrays (including jagged arrays), the `Arrays` utility class (sort/binarySearch/equals/fill/copyOf), array covariance and the `ArrayStoreException` trap, `Arrays.asList()` being a view, and varargs.

Beginner 20 min

Last updated: October 21, 2026
Read online: https://www.learnforgex.com/en/topics/arrays

Arrays

`Array` is the second topic in the Java Basics category -- as fundamental as `String`, but underneath it holds the language's oldest, lowest-level data structure. Arrays are the building block used INSIDE every collection like `ArrayList` / `HashMap`; understanding them also clarifies where performance claims like O(1) access in the Collections category actually come from.

What Is an Array?

An array is a data structure that holds a fixed number of elements of the SAME type in a CONTIGUOUS block of memory. Its size is decided at creation time and can NEVER change afterward -- this is its most fundamental difference from "dynamically sized" collections like `ArrayList`. In Java, arrays like `int[]` / `String[]` look primitive, but they are actually OBJECTS (they derive from `Object`) -- which is why `.length` is accessed as a field, and does NOT require parentheses, unlike `String.length()`.

Why Does It Exist?

Reading/writing an element by index can be done with a direct address calculation at the hardware level, thanks to contiguous memory layout -- this makes arrays the FASTEST possible structure for O(1) index access. The O(1) `get(index)` performance claim of higher-level collections like `ArrayList` comes from exactly this: `ArrayList` wraps an array internally and copies to a new, larger array when it needs to grow. Understanding arrays directly also explains why these higher-level collections are fast or slow.

History

Arrays have been a core part of Java since version 1.0 (1996) -- one of the oldest structures, alongside `String`. The `Arrays` utility class (with its static `sort()` / `binarySearch()` / `equals()` methods) arrived with the Collections Framework in Java 1.2 (1998). Varargs (`Type... args`, syntax that makes an array parameter convenient to use at the call site) was added in Java 5 (2004). Java 8 (2014) connected arrays directly to the Stream API with `Arrays.stream()` (see the "Stream API Fundamentals" lesson).

Basic Usage: Creation, Access, Default Values

An array can be created with `new Type[size]` or with a LITERAL (`{1, 2, 3}`). An uninitialized element of a primitive-type array (like `int[]`) defaults to a value like `0` / `false` ; elements of a reference-type array (like `String[]`) default to `null` . Going out of bounds (`array[10]` when the array's size is 5) does NOT silently return something wrong -- it throws a real `ArrayIndexOutOfBoundsException` .

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

> Printing an array directly with `System.out.println(array)` does NOT show its contents -- it gives a "type@hashcode" string like `[I@7ea987ac`, because that's `Object.toString()`'s default behavior. Always use `Arrays.toString()` (one-dimensional) or `Arrays.deepToString()` (multi-dimensional) to see the actual content.

Multi-Dimensional Arrays

A "2D array" in Java is really an ARRAY OF ARRAYS -- each "row" is its own independent array object. This means rows can have DIFFERENT lengths (called a "jagged array"); a rectangular grid is just a special case where all rows happen to be the same length.

```
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()` is the WRONG tool for nested arrays -- it shows each row as another hashcode string like `[I@...`, without descending into the CONTENT. Use `Arrays.deepToString()` to properly print a multi-dimensional array.

The Arrays Utility Class

`Arrays`, similar to `Collections` (see the "Queues & Collections Utility" lesson), is a utility class offering ready-made static methods that work on arrays: `sort()` (sorts in place), `binarySearch()` (O(log n) search on a SORTED array), `equals()` (CONTENT comparison -- unlike `==`), `fill()` (sets every element to the same value), and `copyOf()` / `copyOfRange()` (copy into a new array).

```
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));
    }
}
```

Expected Output

```
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]
```

> Comparing two arrays with `==` compares REFERENCE (is it the same memory address), not CONTENT -- exactly the same trap as `==` vs `equals()` on `String` (see the "String" lesson). Use `Arrays.equals()` to check whether two arrays hold the same elements.

Array Covariance: A Trap the Compiler Misses

Java arrays are COVARIANT: since `Integer` extends `Number`, an `Integer[]` can be assigned to a `Number[]` variable. But this opens a trap door: the compiler allows WRITING a `Double` through that `Number[]` reference (since `Double` is also a `Number`) -- but the array's REAL runtime type is still `Integer[]`, so this write fails not at compile time, but at RUNTIME, with an `ArrayStoreException`.

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

> Generics (`List<T>`) DELIBERATELY avoid this trap: a `List<Integer>` CANNOT be assigned to a `List<Number>` variable at all (they're invariant) -- so the kind of mistake array covariance hides is caught at COMPILE time instead of runtime with generic collections. This is the answer to "why doesn't `List<Number> list = new ArrayList<Integer>();` compile?"

Arrays vs Collections: Arrays.asList() and Conversions

`Arrays.asList()` does NOT copy the given array -- it wraps the original array in a FIXED-SIZE `List` VIEW. Writing through this view also changes the original array (and vice versa); since it's fixed-size, `add()` / `remove()` aren't supported (they throw `UnsupportedOperationException`), only `set()` (replacing an existing index) works. For a truly independent, resizable list, this view needs to be WRAPPED in `new ArrayList<>(...)`.

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

Expected Output

```
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: Using an Array with Convenient Call Syntax

Varargs (`Type... name`) is syntax that lets the CALLING side pass zero, one, or many arguments to a method -- INSIDE the method, that parameter is simply a regular array. A varargs parameter can only be the LAST parameter in a method signature.

```
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();
    }
}
```

Expected Output

```
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()` and `String.format()` themselves use varargs (`Object... args`) -- that's how they can accept as many `%s` / `%d` placeholders as you need with a single method (see the "String" lesson).

Best Practices

- **Always use `Arrays.toString()` (one-dimensional) or `Arrays.deepToString()` (multi-dimensional) to print an array's contents**, not `System.out.println(array)` directly -- that just gives a meaningless "type@hashcode" string.
- **Use `Arrays.equals()` to compare two arrays' contents**, not `==` -- `==` only compares reference, just like with `String`.
- **Use a collection like `ArrayList` if you need a collection whose size changes while the program runs**, not an array -- arrays can't be resized after creation.
- **Remember that `Arrays.asList()` is a VIEW, not a copy** -- if you need an independent, resizable list, wrap it with `new ArrayList<>(Arrays.asList(...))`.

Common Mistakes

- **Printing an array directly and getting a meaningless output like `[I@7ea987ac`.** `Arrays.toString()` / `Arrays.deepToString()` should have been used instead.
- **Comparing two arrays with `==` and getting `false` even though the content is identical.** `Arrays.equals()` is needed for content comparison.
- **Calling `add()` / `remove()` on the list returned by `Arrays.asList()` and getting `UnsupportedOperationException`.** That view is fixed-size -- an actual `ArrayList` needs to be wrapped explicitly if a real one is needed.
- **Assuming array covariance (that an `Integer[]` can be assigned to a `Number[]` variable) is safe.** It carries a risk of `ArrayStoreException` that isn't caught at compile time -- generic collections don't carry this risk.

Summary, Cheat Sheet, and Glossary

An array is a fundamental data structure that holds a fixed number of same-type elements in contiguous memory, providing O(1) index access -- it's the building block used inside higher-level collections like `ArrayList`. The `Arrays` utility class offers static methods like `sort()` / `binarySearch()` / `equals()` / `fill()` / `copyOf()`. Arrays are covariant, so some mistakes (unlike with generic collections) are only caught at runtime; `Arrays.asList()` is a fixed-size view wrapping the original array, not a copy.

Quick reference:

```
int[] numbers = new int[5];                 // fixed size, default 0s
String[] fruits = {"apple", "banana"};         // creation via a literal
Arrays.toString(numbers);                        // to print the content correctly
Arrays.sort(numbers);                              // sort in place
Arrays.equals(a, b);                                 // CONTENT comparison (not ==)
Arrays.copyOf(numbers, 10);                            // a new, larger copy
List<String> view = Arrays.asList(fruits);                // FIXED-SIZE view, not a copy
List<String> real = new ArrayList<>(Arrays.asList(fruits)); // independent, resizable list
```

Glossary

Array — A fundamental data structure that holds a fixed number of same-type elements in a contiguous block of memory.

Jagged Array — A multi-dimensional array whose rows (sub-arrays) can have different lengths.

Array Covariance — The ability to assign a subtype array (`Integer[]`) to a supertype array variable (`Number[]`); can potentially lead to a runtime error (`ArrayStoreException`).

Varargs — Syntax that lets a method's caller pass zero or many arguments, which becomes a plain array inside the method (`Type... name`).

Arrays — A utility class offering ready-made static methods that work on arrays (`sort` , `equals` , `fill` , `copyOf` , etc.).