Module 09 — Arrays

ITP 120 — Java Programming I

Northern Virginia Community College | Randy Michak

Prerequisites: Modules 1–8 — you should be comfortable with variables, control flow, methods, and the basics of OOP before working through this module.

🎯 Learning Objectives

- Declare, create, and initialize arrays in Java
- Access and modify array elements using index notation
- Use `.length` and loops to process every element
- Implement common operations: sum, average, min, max, search, count
- Pass arrays to methods and return arrays from methods
- Use utility methods from `java.util.Arrays`
- Declare and traverse two-dimensional arrays with nested loops
- Recognize and avoid the most common array bugs

1. What Are Arrays?

Every variable you have written so far holds exactly one value. That is fine for a single score or a single name. But what if you need to store 30 quiz scores? Writing `score1`, `score2`, ... `score30` is painful and nearly impossible to loop over.

An **array** solves this. An array is a fixed-size, ordered collection of elements that are all the same type. Once you create an array with a certain size, that size never changes.

> 🧠 **Think of It Like a Row of Mailboxes**
>
> Picture a row of 5 mailboxes outside an apartment building. Each box is numbered 0–4. Every box holds exactly one piece of mail. You can reach any box instantly if you know its number — that instant access is what makes arrays efficient.

Zero-Based Indexing

Java arrays start counting at **0**, not 1. An array with 5 elements has valid indexes 0, 1, 2, 3, and 4. Index 5 does not exist — trying to use it causes a runtime crash.

```
// 5-element array conceptual layout
Index:   0     1     2     3     4
       +-----+-----+-----+-----+-----+
Value: | 10  | 20  | 30  | 40  | 50  |
       +-----+-----+-----+-----+-----+
```

> 📖 **Key Terms**
>
> - **Array** — a fixed-size, ordered collection of same-type elements stored in contiguous memory
> - **Element** — one item stored inside an array
> - **Index** — the integer position used to access a specific element (starts at 0)
> - **Zero-based indexing** — the first element is at index 0, not index 1

2. Declaring and Creating Arrays

Declaration Syntax

Declaring an array tells Java the type and name. It does *not* create the array or allocate memory yet.

```
// Syntax: type[] arrayName;
int[]    scores;
String[] names;
double[] prices;
```

Creating an Array with `new`

Creating the array reserves memory for a specific number of elements. All elements are set to their default value automatically: `0` for numeric types, `false` for boolean, `null` for objects and Strings.

```
// Declaration and creation in one line (most common pattern)
int[]    scores = new int[5];      // 5 ints, all initialized to 0
String[] names  = new String[3];   // 3 Strings, all initialized to null
double[] prices = new double[10];  // 10 doubles, all initialized to 0.0
```

Initialization Lists

When you already know the values at compile time, use curly braces. Java counts the values and sets the array size automatically.

```
int[]    scores = { 88, 92, 75, 100, 63 };        // length 5
String[] days   = { "Mon", "Tue", "Wed", "Thu", "Fri" }; // length 5
double[] temps  = { 98.6, 37.0, 212.0 };          // length 3
```

> 💡 **new vs. Initialization List**
>
> Use `new int[n]` when you know the size but not the values yet (you will fill them in with user input or calculations). Use an initialization list `{ v1, v2, … }` when you know the values at the time you write the code.

3. Accessing Array Elements

Index Notation

Use `arrayName[index]` to read or write any element.

```
int[] scores = { 88, 92, 75, 100, 63 };

// Reading elements
System.out.println(scores[0]);   // 88
System.out.println(scores[3]);   // 100

// Writing (overwriting) an element
scores[2] = 80;
System.out.println(scores[2]);   // 80  (was 75)

// A variable can serve as the index
int i = 4;
System.out.println(scores[i]);   // 63
```

ArrayIndexOutOfBoundsException

This is the most common array crash. It happens the instant you use an index below 0 or at/above the array's length.

```
int[] scores = new int[5];   // valid indexes: 0, 1, 2, 3, 4

scores[5] = 90;
// Exception in thread "main"
// java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 5
```

> ⚠️ **ArrayIndexOutOfBoundsException**
>
> For an array of length `n`, valid indexes are **0 through n − 1**. Index `n` is always out of bounds. Read the exception message — it tells you the index you tried and the array's length, which is enough to find the bug.

4. Array Length

Every array has a built-in `.length` **property** that tells you how many elements it holds. It is a field, not a method — no parentheses.

```
int[] scores = { 88, 92, 75, 100, 63 };

System.out.println(scores.length);   // 5

// scores.length()  <-- compile error: .length is a field, not a method
```

Using `.length` in Loops

Hard-coding the number of elements in a loop condition is fragile. Use `.length` so the loop still works if the array size ever changes.

```
int[] scores = { 88, 92, 75, 100, 63 };

for (int i = 0; i < scores.length; i++) {
    System.out.println(scores[i]);
}
// 88
// 92
// 75
// 100
// 63
```

> 💡 **length vs. length()**
>
> Arrays use `.length` (no parentheses — it is a field). Strings use `.length()` (with parentheses — it is a method). The distinction is annoying but important. You will get a compiler error immediately if you mix them up, so at least it fails early.

5. Processing Arrays with Loops

Standard for Loop

The standard `for` loop gives you the index at each step, so you can read *and* write elements, process every other element, or traverse in reverse.

```
int[] numbers = { 3, 7, 1, 9, 4 };

// Print each element with its index
for (int i = 0; i < numbers.length; i++) {
    System.out.println("numbers[" + i + "] = " + numbers[i]);
}
// numbers[0] = 3
// numbers[1] = 7
// numbers[2] = 1
// numbers[3] = 9
// numbers[4] = 4

// Double every element (requires writing via index)
for (int i = 0; i < numbers.length; i++) {
    numbers[i] = numbers[i] * 2;
}
```

Enhanced for-each Loop

The for-each loop is cleaner when you just need to read every element in order. A local variable takes on each element's value, one at a time.

```
int[] numbers = { 3, 7, 1, 9, 4 };

// Syntax: for (type variable : arrayName)
for (int num : numbers) {
    System.out.println(num);
}
// 3
// 7
// 1
// 9
// 4
```

💡 Which Loop to Use

- **Standard for loop** — you need the index, you need to write to elements, you need to skip elements, or you need to traverse in reverse
- **for-each** — you are reading every element in order and the index does not matter

⚠️ for-each Cannot Modify the Array

The loop variable is a *copy* of the element's value. Changing the copy does not change the array.

```
int[] nums = { 1, 2, 3 };
for (int n : nums) {
    n = n * 10;   // modifies the copy only — array unchanged
}
System.out.println(nums[0]);   // still 1
```

6. Common Array Operations

Sum and Average

```
int[] scores = { 88, 92, 75, 100, 63 };
int sum = 0;

for (int score : scores) {
    sum += score;
}

double average = (double) sum / scores.length;

System.out.println("Sum: "     + sum);      // Sum: 418
System.out.println("Average: " + average);  // Average: 83.6
```

> 💡 **Cast Before Dividing**
>
> Both `sum` and `scores.length` are integers. Integer division drops the decimal. Cast one to `double` first: `(double) sum / scores.length`.

Find Minimum and Maximum

```
int[] scores = { 88, 92, 75, 100, 63 };

// Seed with the first element, then challenge with every other
int min = scores[0];
int max = scores[0];

for (int i = 1; i < scores.length; i++) {
    if (scores[i] < min) min = scores[i];
    if (scores[i] > max) max = scores[i];
}

System.out.println("Min: " + min);   // Min: 63
System.out.println("Max: " + max);   // Max: 100
```

Search for a Value (Linear Search)

```
int[] scores = { 88, 92, 75, 100, 63 };
int target  = 75;
int foundAt = -1;   // -1 signals "not found"

for (int i = 0; i < scores.length; i++) {
    if (scores[i] == target) {
        foundAt = i;
        break;
    }
}

if (foundAt != -1) {
    System.out.println("Found " + target + " at index " + foundAt);
} else {
    System.out.println(target + " not in array");
}
// Found 75 at index 2
```

Count Occurrences

```
int[] rolls = { 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5 };
int count = 0;

for (int roll : rolls) {
    if (roll == 5) count++;
}

System.out.println("Fives rolled: " + count);   // Fives rolled: 3
```

🔧 Try It Yourself

Write a program that reads 6 integers from the user (Scanner in a loop), stores them in an array, then prints the sum, average, and the largest value. Use `.length` everywhere — never hard-code the number 6.

7. Arrays and Methods

Passing Arrays to Methods

When you pass an array to a method, Java passes the **reference** — both the caller and the method are pointing at the same object in memory. Any changes made inside the method affect the original array.

```
public static void doubleAll(int[] arr) {
    for (int i = 0; i < arr.length; i++) {
        arr[i] *= 2;
    }
}

public static void main(String[] args) {
    int[] nums = { 1, 2, 3, 4, 5 };
    doubleAll(nums);

    for (int n : nums) {
        System.out.print(n + " ");
    }
    // 2 4 6 8 10
}
```

⚠️ Arrays Are Passed by Reference

Unlike primitives (`int` , `double` , etc.), passing an array to a method does *not* give the method a copy. It gives the method access to the same array. If you need to protect the original, use `Arrays.copyOf()` to make a copy before passing it.

Returning Arrays from Methods

```
public static int[] buildSquares(int n) {
    int[] result = new int[n];
    for (int i = 0; i < n; i++) {
        result[i] = (i + 1) * (i + 1);
    }
    return result;
}

public static void main(String[] args) {
    int[] squares = buildSquares(5);
    for (int s : squares) {
        System.out.print(s + " ");
    }
    // 1 4 9 16 25
}
```

8. The Arrays Class

Java's `java.util.Arrays` class has ready-made methods for the most common array tasks. Add the import at the top of your file.

```
import java.util.Arrays;
```

Arrays.sort()

```
int[] nums = { 5, 2, 8, 1, 9, 3 };
Arrays.sort(nums);
System.out.println(Arrays.toString(nums));
// [1, 2, 3, 5, 8, 9]
```

Arrays.toString()

Prints the array in a readable `[a, b, c]` format. Without this, printing an array directly produces a useless memory address.

```
int[] nums = { 10, 20, 30 };
System.out.println(nums);                  // [I@1b6d3586  <-- useless
System.out.println(Arrays.toString(nums)); // [10, 20, 30]  <-- readable
```

Arrays.copyOf()

Creates a new array that is a copy of an existing one. Use this when you want to work with a modified version without touching the original.

```
int[] original = { 1, 2, 3, 4, 5 };

int[] copy    = Arrays.copyOf(original, original.length);  // all 5
int[] partial = Arrays.copyOf(original, 3);                // first 3

System.out.println(Arrays.toString(copy));    // [1, 2, 3, 4, 5]
System.out.println(Arrays.toString(partial)); // [1, 2, 3]
```

Arrays.fill()

Sets every element to the same value — handy for initializing to something other than zero.

```
int[] grades = new int[5];
Arrays.fill(grades, 100);
System.out.println(Arrays.toString(grades));
// [100, 100, 100, 100, 100]
```

Method	What It Does	Returns
`Arrays.sort(arr)`	Sorts elements ascending (modifies the array in-place)	`void`
`Arrays.toString(arr)`	Produces a readable string like `[1, 2, 3]`	`String`
`Arrays.copyOf(arr, len)`	Returns a new copy with the specified length	New array

Method	What It Does	Returns
`Arrays.fill(arr, val)`	Sets every element to `val`	`void`

> 🔧 **Try It Yourself**
>
> Create an unsorted array of 7 integers. Print it with `Arrays.toString()`. Sort it with `Arrays.sort()` and print again. Then copy it with `Arrays.copyOf()`, fill the copy with `-1` using `Arrays.fill()`, and print the copy to confirm.

9. Two-Dimensional Arrays

A 2D array is an array of arrays — think of it as a table with rows and columns.

Declaration and Creation

```
// Syntax: type[][] name = new type[rows][cols];
int[][] grid = new int[3][4];   // 3 rows, 4 columns, all 0

// Initialization list
int[][] matrix = {
    { 1, 2, 3 },
    { 4, 5, 6 },
    { 7, 8, 9 }
};
// matrix[0][0] = 1   matrix[0][2] = 3
// matrix[1][1] = 5   matrix[2][2] = 9
```

Accessing Elements

Use two indexes: `[row][col]`. Both start at 0.

```
int[][] matrix = {
    { 1, 2, 3 },
    { 4, 5, 6 },
    { 7, 8, 9 }
};

System.out.println(matrix[0][0]);   // 1  (row 0, col 0)
System.out.println(matrix[1][2]);   // 6  (row 1, col 2)
System.out.println(matrix[2][1]);   // 8  (row 2, col 1)

matrix[1][1] = 99;
System.out.println(matrix[1][1]);   // 99
```

Nested Loops to Process a 2D Array

Use an outer loop for rows and an inner loop for columns. `matrix.length` gives the row count; `matrix[0].length` gives the column count.

```
int[][] matrix = {
    { 1, 2, 3 },
    { 4, 5, 6 },
    { 7, 8, 9 }
};

for (int row = 0; row < matrix.length; row++) {
    for (int col = 0; col < matrix[row].length; col++) {
        System.out.print(matrix[row][col] + " ");
    }
    System.out.println();   // newline after each row
}
// 1 2 3
// 4 5 6
// 7 8 9
```

Practical Example: Classroom Grade Table

```
// 3 students, 4 quiz scores each
int[][] grades = {
    { 88, 92, 75, 100 },   // Student 0
    { 70, 85, 90,  78 },   // Student 1
    { 95, 60, 88,  82 }    // Student 2
};

// Print each student's average
for (int s = 0; s < grades.length; s++) {
    int sum = 0;
    for (int q = 0; q < grades[s].length; q++) {
        sum += grades[s][q];
    }
    double avg = (double) sum / grades[s].length;
    System.out.printf("Student %d average: %.1f%n", s, avg);
}
// Student 0 average: 88.8
// Student 1 average: 80.8
// Student 2 average: 81.3
```

10. Common Mistakes

⚠️ Mistake 1: Off-by-One Errors

The last valid index is `length - 1`, not `length`. Using `<=` instead of `<` in a loop condition is the classic version of this bug.

```
int[] arr = new int[5];

// WRONG: i <= arr.length runs i = 5, which crashes
for (int i = 0; i <= arr.length; i++) { arr[i] = i; }

// CORRECT
for (int i = 0; i < arr.length; i++) { arr[i] = i; }
```

⚠️ Mistake 2: null vs. Empty Array

`null` means the variable does not point to any array at all. An empty array (`new int[0]`) exists but has zero elements. Calling `.length` on a `null` reference crashes with a `NullPointerException`.

```
int[] a = null;          // no array at all
int[] b = new int[0];    // empty array: exists, length = 0

System.out.println(b.length);   // 0  (fine)
System.out.println(a.length);   // NullPointerException  (crash)
```

⚠️ Mistake 3: Forgetting `new`

Declaring an array variable without creating the array leaves it as `null`. You must call `new` (or use an initialization list) before using the array.

```
int[] scores;        // declared but null
scores[0] = 90;      // NullPointerException

// Fix: create it first
int[] scores = new int[5];
scores[0] = 90;      // fine
```

⚠️ Mistake 4: Trying to Modify the Array in a for-each Loop

As covered in Section 5, the for-each loop variable is a copy. Assigning to it has no effect on the original array. Use a standard `for` loop when you need to write to elements.

```
int[] nums = { 1, 2, 3 };

// WRONG: does not change the array
for (int n : nums) { n = 0; }

// CORRECT
for (int i = 0; i < nums.length; i++) { nums[i] = 0; }
```

Module Summary

📋 **Module Summary**

- An array stores a fixed number of same-type elements, indexed from 0 to length−1.
- Declare with `type[] name`, create with `new type[size]` or an initialization list.
- Access elements with `name[index]`; using an out-of-range index throws `ArrayIndexOutOfBoundsException`.
- `.length` (no parentheses) gives the number of elements — use it in loop conditions.
- Standard `for` loop when you need the index or need to write; for-each when reading in order.
- Common operations: sum/average, min/max, linear search, count — all use loops.
- Arrays are passed by reference: changes inside a method affect the original array.
- `java.util.Arrays` provides `sort()`, `toString()`, `copyOf()`, and `fill()`.
- 2D arrays use `[row][col]` notation; nested loops process every element.
- Watch out for: off-by-one, null vs. empty, missing `new`, for-each modification.

Vocabulary

Term	Definition
Array	A fixed-size, ordered collection of same-type elements stored in contiguous memory
Element	A single item stored inside an array
Index	The integer position used to access an element; starts at 0

Term	Definition
Zero-based indexing	A numbering convention where the first element is at position 0
`.length`	A field on every array that holds its total number of elements
Initialization list	A set of values in curly braces used to create and populate an array at declaration
for-each loop	An enhanced loop that iterates over every element without exposing the index
Pass by reference	Passing the memory address of an object (like an array) so the method accesses the original
Linear search	Scanning an array from start to end to find a target value
`java.util.Arrays`	A Java utility class with helper methods for sorting, copying, and printing arrays
2D array	An array of arrays; accessed with two indexes `[row][col]`
`ArrayIndexOutOfBoundsException`	A runtime exception thrown when an index is below 0 or at/above the array's length
`NullPointerException`	A runtime exception thrown when you use a variable that points to `null` instead of an object

Module 09 Quiz

Choose the best answer for each question. Write the letter on your answer sheet.

1. What is the index of the *last* element in an array declared as `int[] data = new int[8];` ?

A) 7

B) 9

C) 0

D) 8

2. Which of the following correctly declares *and* creates an array of 4 doubles?

A) `double[] arr = new double[4];`

B) `double arr = new double[4];`

C) `double[] arr = double[4];`

D) `new double[4] arr;`

3. What does the following code print?

```
int[] vals = { 10, 20, 30, 40, 50 };
System.out.println(vals[2] + vals[4]);
```

A) 50

B) 70

C) 80

D) 60

4. Which exception is thrown when you access an array with an index equal to the array's length?

A) `NullPointerException`

B) `IllegalArgumentException`

C) `IndexOutOfRangeException`

D) `ArrayIndexOutOfBoundsException`

5. What does `scores.length` return for the array `int[] scores = { 5, 3, 8, 1 };` ?

A) 3

B) 4

C) 5

D) 8

6. What is the output of the following code?

```
int[] nums = { 2, 4, 6, 8 };
int total = 0;
for (int n : nums) {
    total += n;
}
System.out.println(total);
```

A) 10

B) 48

C) 20

D) 14

7. A method receives an `int[]` parameter and doubles every element. After the method returns, the caller's array:

A) Is unchanged because arrays are passed by value

B) Is doubled because arrays are passed by reference

C) Has all elements set to zero

D) Throws an exception when the method returns

8. Which `java.util.Arrays` method would you call to display an array as `[3, 7, 12]` ?

A) `Arrays.toString()`

B) `Arrays.display()`

C) `Arrays.toArray()`

D) `Arrays.print()`

9. What is the output of the following code?

```
int[][] table = {
    { 1, 2 },
    { 3, 4 },
    { 5, 6 }
};
System.out.println(table[2][0] + table[0][1]);
```

A) 8

B) 6

C) 7

D) 3

10. Which loop structure should you use when you need to *write* new values into every element of an existing array?

A) Enhanced for-each loop, because it is cleaner

B) A while loop only — for loops cannot modify arrays

C) Either loop works identically for writing

D) Standard indexed for loop, because you need the index to assign values

✅ Answer Key

Question	Answer	Explanation
1	**A**	An array stores a fixed-size collection of elements of the same data type.
2	**A**	Array indices start at 0 in Java.
3	**C**	`array.length` returns the number of elements in the array (no parentheses).
4	**D**	An `ArrayIndexOutOfBoundsException` occurs when accessing an invalid index.
5	**B**	An enhanced for loop (`for-each`) iterates over every element without using an index.
6	**C**	Arrays are passed by reference — the method can modify the original array.
7	**B**	A 2D array is an array of arrays, like a table with rows and columns.
8	**A**	`Arrays.sort()` sorts the elements in ascending order.
9	**C**	Array elements are initialized to default values (0 for int, null for objects).
10	**D**	Once created, the size of a Java array cannot be changed.