Day 1 of DSA: Arrays Fundamentals

January 1, 2026 • DSA

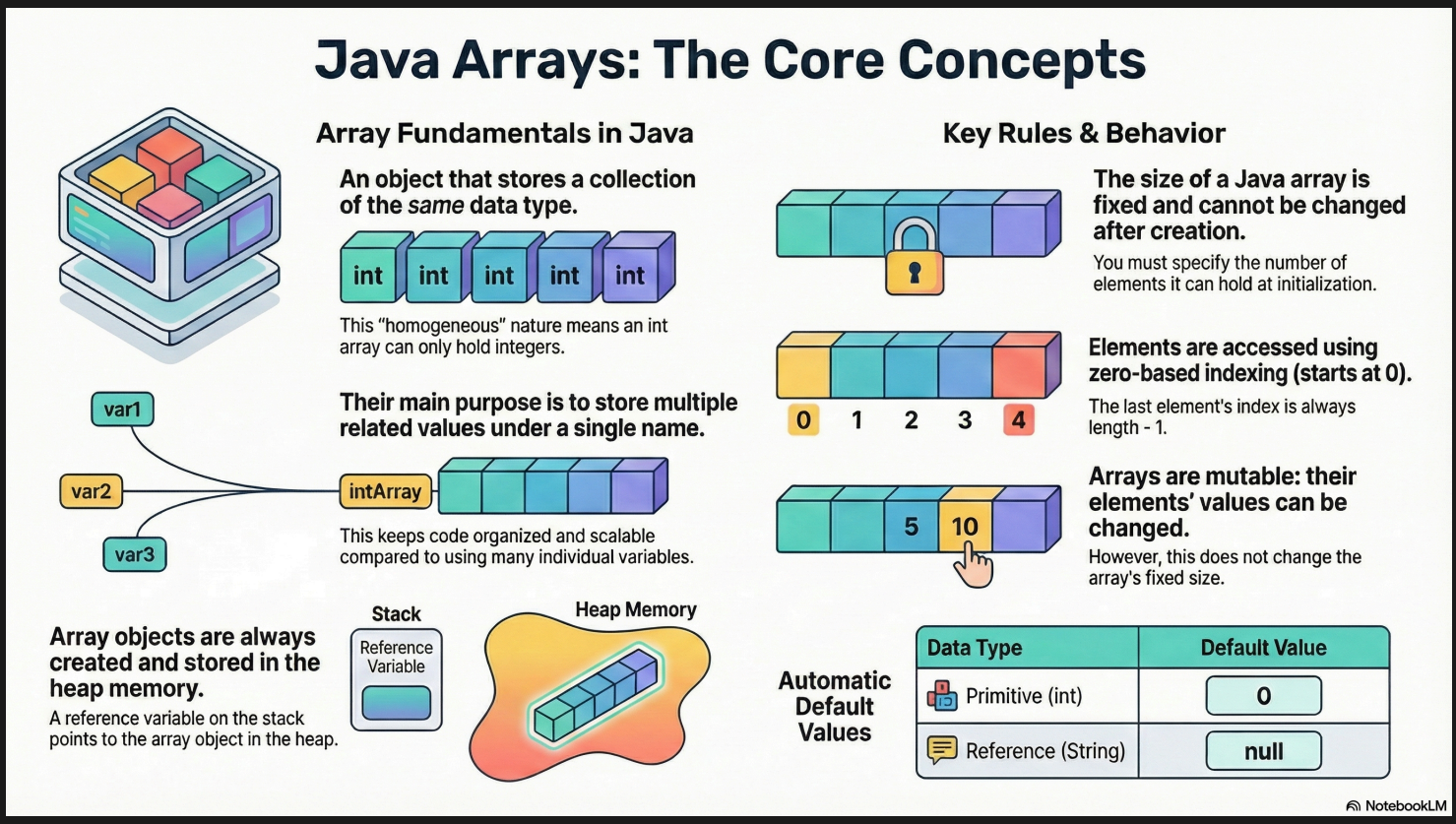
Day 1: Arrays

I chose **arrays** as the starting point for my DSA journey. Although I am not a complete beginner—I have learned Java and basic DSA earlier—I felt arrays are a good concept to revisit and strengthen before moving to more advanced topics.

However, I would not recommend absolute beginners to start directly with arrays. Beginners should first be comfortable with **basic programming concepts such as variables, data types, and loops**. Once those fundamentals are clear, arrays become much easier to understand.

Before learning DSA, it is important to choose a programming language. I have chosen **Java** as my primary language, and in this blog, I will cover DSA concepts **specifically from a Java perspective**, with brief comparisons to Python where necessary.

A quick thanks to the instructor of this YouTube tutorial—it helped me understand the concepts better.



Visual guide generated by NotebookLM—thanks for making these concepts easier to grasp!

Topics Covered in This Blog

1. Why do we need Arrays?
2. What is an Array?
3. Arrays in Java and Python
4. Syntax of an Array in Java
5. Rules of Arrays in Java
6. Declaration and Initialization
7. Example of Creating an Array
8. Indexing in Arrays
9. Length of an Array
10. Default Values in Arrays
11. String Arrays and Memory
12. Primitive Types vs Objects
13. Mutability of Arrays

Why Do We Need Arrays?

Arrays are used to store a **collection of related values under a single name**, which is usually called a **reference variable**.

For example, if we want to store the names of our friends, instead of creating multiple variables for each name, we can store all the names in a single array called `friendsNames`. This makes the code more organized, scalable, and easier to work with.

When I picture an array, I imagine a row of empty boxes on a shelf. Each box has a fixed position number, and I can drop one related item per box. That mental model makes it easier to write loops later because I know each index follows a predictable pattern.

Array as labeled boxes

Fixed-size row of slots; each slot knows its index and holds one value.

What Is an Array?

A commonly used definition of an array is:

An array is a collection of elements of the same data type stored in contiguous memory locations.

This definition is **completely accurate for low-level languages such as C and C++**.

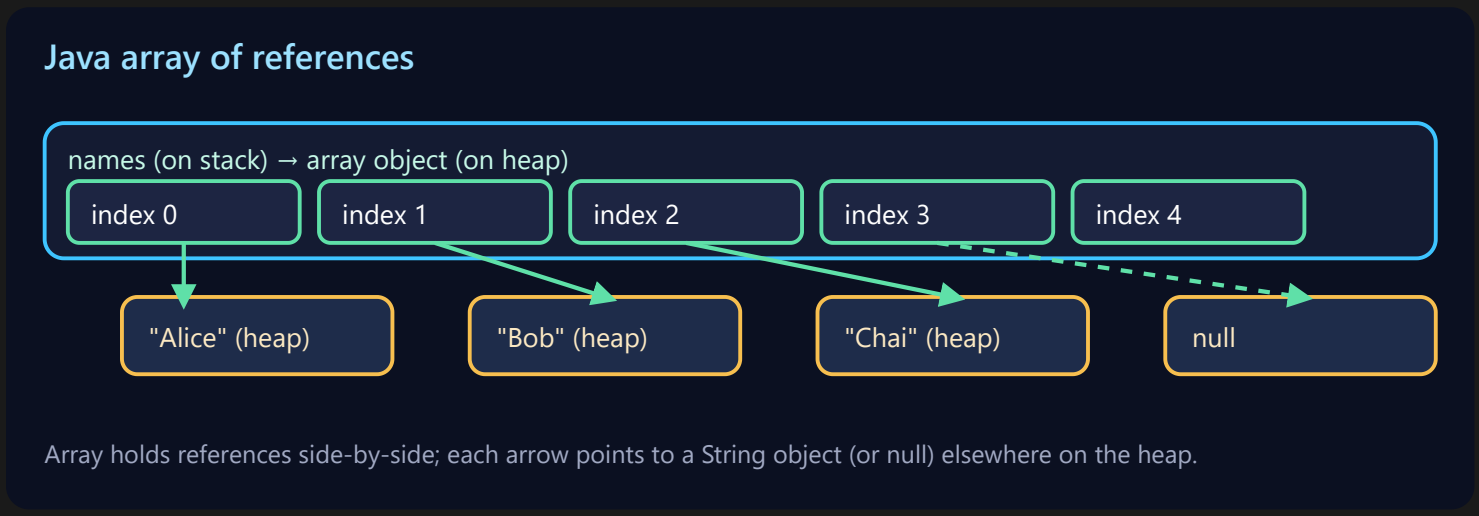
However, the internal implementation of arrays can differ across programming languages.

“Contiguous” simply means the slots are side by side in memory. The benefit is fast random access: jumping to index `i` is quick because the program can calculate the exact memory address. The trade-off is that resizing usually means creating a brand new array and copying items over.

Arrays in Java

- In Java, an array can store **only one data type**, which makes arrays **homogeneous**.
- Arrays in Java are **objects**, and they are created in the **heap memory**.
- For **primitive data types** (`int`, `double`, etc.), the values are stored **contiguously within the array object**.
- For **reference types** (`String`, `Object`, etc.), the array stores **contiguous references**, while the actual objects may be located at different places in the heap.

I like to remember that `String[] names` really holds references. Changing `names[0] = "Bob"` only swaps which `String` object that slot points to; it does not move or copy the underlying `String` objects elsewhere.



Arrays in Python

- Python does not provide traditional arrays by default.
- What is commonly used in Python is a **list**, which can store **different data types**.
- A Python list stores **references to objects**, not the raw values themselves.

Because Python lists are dynamic, appending feels effortless. The runtime quietly reallocates and copies when needed. That is friendly for beginners but also hides the cost of growing a list compared to a fixed-size Java array.

Syntax of an Array in Java

```
datatype[] arrayName = new datatype[size];
```

Explanation:

- `datatype` → specifies the type of elements the array can store
- `[]` → indicates that the variable is of array type
- `arrayName` → reference variable that points to the array object
- `new` → used to create an array object in the heap
- `datatype[size]` → specifies the data type and the number of elements the array can hold

When I first wrote this, the position of `[]` confused me. `datatype[] arrayName` and `datatype arrayName[]` both compile in Java, but I stick to the first style because it reads as "`arrayName` is an array of `datatype`." The size is fixed at creation time.

Rules of Arrays in Java

1. The reference variable name must follow the same naming conventions as normal variables.
2. All elements in an array must be of the same data type.
3. The size of an array is **fixed** once it is created and cannot be changed.

If I think I will need a changing size, I start with `ArrayList` instead. Sticking to plain arrays is great practice for learning indexing and for interview-style questions where fixed-size storage is expected.

Declaration and Initialization

```
datatype[] arr;           // declaration
arr = new datatype[size]; // initialization
```

- During declaration, a **reference variable** is created.
- During initialization, the **array object** is created in the heap.
- Declaration is checked at **compile time**.
- Memory allocation for the array happens at **runtime**.

An easy mistake: calling `arr[0]` before `arr` is initialized triggers a `NullPointerException` because the reference points to nothing. I keep reminding myself that declaration alone does not create the boxes—initialization does.

Example of Creating an Array

```
int[] nums = {1, 2, 3, 4, 5, 6};
```

This creates and initializes an integer array containing six elements.

Whenever I declare with curly braces, I read the values out loud—"one, two, three, four, five, six"—to check I did not skip a number. Printing `nums.length` also confirms the size matches what I expect.

Indexing in Arrays

Indexing is used to access elements of an array.

- Array indexing in Java starts from **0**.
- The first element is stored at index `0`.

Example:

```
int[] num = {2, 7, 8};
```

- `num[0]` → `2`
- `num[1]` → `7`
- `num[2]` → `8`

Here:

- Length of the array = `3`
- Last index = `length - 1 = 2`

Many beginners confuse the **length of an array** with the **last index**, but they are not the same.

If we try to access an index greater than the last index, Java throws an exception called:

`ArrayIndexOutOfBoundsException`

I still pause before writing loop bounds: `for (int i = 0; i < num.length; i++)` is safer than `<=` because it naturally stops at the last valid index. Drawing three boxes labeled 0, 1, 2 helps me visualize where the loop should end.

Length of an Array

```
num.length
```

- `length` gives the **number of elements** present in the array.
- It is a **property**, not a method.

To find the last index of an array:

```
int lastIndex = num.length - 1;
```

`length` is a field, not a method, so there are no parentheses. Using `num.length()` by accident will not compile, which is a helpful reminder when switching between arrays and `ArrayList` (which uses `size()`).

Default Values in Arrays

If an array is created without explicitly assigning values, Java automatically assigns **default values**.

Example:

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

Since `int` is a primitive data type, each element is initialized to `0`.

For reference types:

```
String[] names = new String[5];
```

Each element is initialized to `null`.

This demonstrates the difference in behavior between **primitive types** and **reference types** in arrays.

Defaults are great for quick experiments, but they can also hide missing data. When I see `0` or `null` in logs, I double-check if that value is intentional or just the default placeholder I forgot to overwrite.

String Arrays and Memory

```
String[] languages = {"Java", "Python", "C++"};
```

- The array stores **references** to `String` objects.
- Each `String` object is stored separately in the heap.
- The array itself is an object and is also stored in the heap.

If I reassign `languages[1] = "Go"`, only that slot changes. The original `"Python"` string still exists until garbage collection cleans it up, which is why arrays of references feel lightweight to modify.

Primitive Types vs Objects

- Primitive values can be stored in the **stack** (for local variables) or inside objects in the **heap**, depending on where they are declared.
- Objects and arrays are **always stored in the heap**.
- Reference variables that point to objects are stored in the **stack**.

Thinking in two layers helps me: the reference lives on the stack (quick to access), while the actual array object lives on the heap (shared and resizable only by creating new objects). When I pass an array into a method, I am really passing the reference, so changes inside the method affect the original array.

Mutability of Arrays

Arrays in Java are **mutable**, meaning their elements can be changed after creation.

However, the **size of an array cannot be modified** once it is created.

Example:

```
int[] scores = {10, 20, 30};
scores[1] = 25; // updates the second element
```

`scores` still has three elements, but the middle value is now `25`. If I need a fourth score, I must create a new array or use a dynamic structure like `ArrayList`.

Day 1 Takeaway

Arrays form the foundation for many data structures in DSA. A clear understanding of how arrays work—especially indexing, memory behavior, and limitations—is essential before moving on to more advanced topics.

Free Resources I Found Helpful

- Oracle Java Tutorials on arrays: https://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html
- GeeksforGeeks introduction to arrays: https://www.geeksforgeeks.org/introduction-to-arrays/
- W3Schools Java arrays overview: https://www.w3schools.com/java/java_arrays.asp
- FreeCodeCamp guide on array basics: https://www.freecodecamp.org/news/arrays-explained/

If you made it here, thanks for learning with me. I am still treating this like a lab notebook: simple drawings, small code snippets, and honest mistakes I catch along the way. If you spot gaps or have a favorite beginner-friendly resource, let me know so I can keep improving this journal.

Tags: learning, cs, dsa

Comments