**Array Introduction**

**Basic terminologies of Array**

* **Array Element:**Elements are items stored in an array.
* **Array Index:** Elements are accessed by their indexes. Indexes in most of the programming languages start from 0.

👉 In **JavaScript, all arrays are objects** (there’s no difference between primitive vs object arrays like Java).

**🧠 JavaScript Memory Model**

JavaScript divides memory into **two areas**:

1. **Stack (Call Stack / Execution Context)**
   * Stores **primitive values** (Number, String, Boolean, null, undefined, Symbol, BigInt).
   * Also stores **references (pointers)** to objects in the heap.
   * Data here is small and copied **by value**.
2. **Heap (Dynamic Memory)**
   * Stores **objects, arrays, functions**.
   * Data here is big/complex, stored once in memory.
   * Variables hold only a **reference (memory address)** to the object.

**🔹 Case 1: Array of Primitives**

let nums = [10, 20, 30];

**Memory layout:**

**Stack**

nums ───────────────► (ref #101) // pointer to heap

**Heap**

#101 (Array Object)

[0]: 10 // stored directly as value

[1]: 20

[2]: 30

👉 Each element is a **primitive**, so the array stores values directly.  
👉 But the **array object itself** is in the heap, and nums is just a reference in the stack.

**🔹 Case 2: Array of Objects**

let people = [{name: "Lakshit"}, {name: "Rahul"}];

**Memory layout:**

**Stack**

people ──────────────► (ref #201)

**Heap**

#201 (Array Object)

[0] ─► ref #301

[1] ─► ref #302

#301 (Object)

name: "Lakshit"

#302 (Object)

name: "Rahul"

👉 Here, the **array doesn’t store the full objects**.  
👉 It only stores **references (pointers)** to where those objects live in the heap.  
👉 If you copy people[0], you’re copying the reference, not the actual object.

**🔹 Copying (by value vs by reference)**

**Example with primitives**

let a = 10;

let b = a; // copy the value

b = 20;

console.log(a); // 10

console.log(b); // 20

Memory:

Stack:

a = 10

b = 20 (separate copy)

👉 **Primitives are copied by value.**

**Example with objects**

let obj1 = {name: "Lakshit"};

let obj2 = obj1; // copy the reference

obj2.name = "Rahul";

console.log(obj1.name); // Rahul

console.log(obj2.name); // Rahul

Memory:

Stack:

obj1 ─► #401

obj2 ─► #401 (same reference)

Heap:

#401 { name: "Rahul" }

👉 Both variables point to the **same object** in the heap.  
👉 Changing through one reference affects the other.

**⚖️ Summary Table**

| **Data Type** | **Stored in** | **Copy Behavior** |
| --- | --- | --- |
| **Primitive** | Stack | Copied by **value** |
| **Object/Array** | Heap | Copied by **reference** |

✅ So in JS, arrays always live in the **heap**.

1. If elements are **primitives → values stored directly**.
2. If elements are **objects → references stored**

**More detailed explanation:**

**🟢 First — Think of Your Brain Like a Room**

* You have **two shelves** in your brain room:
  1. **Stack shelf** → for small things (toys, blocks) that can fit directly.
  2. **Heap shelf** → for big boxes (like a box of LEGO). Instead of putting the whole box in the stack, you put just the **address tag** that says "Go check that box on the heap shelf".

**🟢 Case 1: Array of Small Toys (Primitives)**

let nums = [10, 20, 30];

* nums is a **box (array)** → this box itself is kept on the **heap shelf**.
* On your **stack shelf**, you don’t store the whole box, only a **pointer (like address #101)** that says "nums → look at heap shelf box #101".

📦 Inside the heap box #101:

* Slot 0: 10
* Slot 1: 20
* Slot 2: 30

👉 Since numbers are **small toys**, they are stored directly inside the box.

**🟢 Case 2: Array of Big Boxes (Objects)**

let people = [{name: "Lakshit"}, {name: "Rahul"}];

* people (the array box) lives in the **heap**.
* On the **stack**, we just have a note: people → #201.

📦 Inside heap box #201:

* Slot 0: 👉 points to another box (#301)
* Slot 1: 👉 points to another box (#302)

📦 Heap box #301:

* name: "Lakshit"

📦 Heap box #302:

* name: "Rahul"

👉 Notice here: the array doesn’t keep the full people inside itself. It just keeps **addresses (pointers)** to where the people’s boxes live.

**🟢 Copying Toys (By Value vs By Reference)**

**Example 1: Small Toys (Primitives)**

let a = 10;

let b = a; // copy

b = 20;

* a has a toy “10”
* b gets its **own copy** of toy “10”
* You change b → now it has “20”, but a still has “10”

👉 **Two separate toys.**

**Example 2: Big Boxes (Objects)**

let obj1 = {name: "Lakshit"};

let obj2 = obj1; // copy reference

obj2.name = "Rahul";

* obj1 is a sticky note saying → "Go to heap box #401"
* obj2 is another sticky note saying → "Go to heap box #401"
* You open box #401 → there is **only one box!**

So if you change the box from one sticky note, the change shows up in the other too.

👉 **One box, two notes pointing to it.**

**⚖️ Baby-Simple Summary**

| **Thing** | **Where it lives** | **What happens if you copy** |
| --- | --- | --- |
| **Primitive** (numbers, strings, booleans) | Stored directly (stack or inside heap box) | Copy makes a new toy (separate) |
| **Object/Array** | Always on heap | Copy just gives you another note (reference) to the same box |

✅ Arrays in JS **always live in the heap**.

* If they hold primitives → values are stored inside directly.
* If they hold objects → only addresses are stored inside.

👉 So:  
Think of **stack = sticky notes** (small + addresses)  
Think of **heap = big toy boxes** (where the real stuff lives).

**Declaration of Array**

**// JS code**

**let arr = []**

**Initialization of Array**

**let arr = [ 1, 2, 3, 4, 5 ];**

**let arr = [ 'a', 'b', 'c', 'd', 'e' ];**

**let arr = [ 1.4, 2.0, 24, 5.0, 0.0 ];**

**Types of Arrays**

**Arrays can be classified in two ways:**

* **On the basis of Size**
* **On the basis of Dimensions**

**A diagram of different types of array

AI-generated content may be incorrect.**

**In JavaScript, arrays are always dynamic in size (they grow/shrink automatically). Unlike Java, C, or C++, you don’t declare a fixed-size array. But we can *simulate* fixed-size arrays if needed.**

**🔹 1. Dynamic Array (default in JS)**

**let arr = [1, 2, 3];**

**arr.push(4); // add at end**

**arr.push(5); // add more**

**arr.pop(); // remove last**

**console.log(arr); // [1, 2, 3, 4]**

**✅ No size limit → automatically resizes.  
✅ Stored in heap, reference in stack.**

**🔹 2. Simulated Fixed-Size Array**

**JS doesn’t enforce size, but you can create an array of fixed length using new Array(size).**

**let fixedArr = new Array(5); // array with length = 5**

**console.log(fixedArr); // [ <5 empty items> ]**

**console.log(fixedArr.length); // 5**

**But still:**

**fixedArr[0] = 10;**

**fixedArr[1] = 20;**

**fixedArr[8] = 30; // JS allows this, expands array**

**console.log(fixedArr.length); // 8 (not fixed!)**

** JS automatically resizes the array to hold that index.**

** All skipped indexes ([5], [6], [7]) stay as empty slots (called “holes” in JS arrays).**

**⚠️ So "fixed size" isn’t strict in JS.**

**Types of Arrays on the basis of Dimensions**

**1. One-dimensional Array(1-D Array): You can imagine a 1d array as a row, where elements are stored one after another.**

**A diagram of a number array

AI-generated content may be incorrect.**

**2. Multi-dimensional Array: A multi-dimensional array is an array with more than one dimension. We can use multidimensional array to store complex data in the form of tables, etc. We can have 2-D arrays, 3-D arrays, 4-D arrays and so on.**

* **Two-Dimensional Array(2-D Array or Matrix): 2-D Multidimensional arrays can be considered as an array of arrays or as a matrix consisting of rows and columns.**

**A diagram of rows and columns

AI-generated content may be incorrect.**

* **Three-Dimensional Array(3-D Array): A 3-D Multidimensional array contains three dimensions, so it can be considered an array of two-dimensional arrays.**

**A diagram of a row of columns

AI-generated content may be incorrect.**