DAY 15 Notes

Key Takeaway	Array
Learning Date	@July 27, 2025
Module	**Module 2: Deep Dive into Functions & Objects**
Status	In progress
✓ Topic	✅ **Day 15:** Mastering Array Course
Video Link	https://www.youtube.com/watch?v=t05NguKFKo0&list=PLIJrr73KDmRw2Fwwjt6cPC_tk5vcSICCu&index=16

JavaScript Array Master Course

In JavaScript, an "index" in an array refers to the position of an element within that array. Array indices start at 0, so the first element is at index 0, the second at index 1, and so on.

📝 Array Add and Remove Methods (Mutating the Original Array)

✅ Add Elements:

1. `push()` – Adds an element to the **end** of the array.

   Example: `arr.push(5);`

2. `unshift()` – Adds an element to the **beginning** of the array.

   Example: `arr.unshift(1);`

❌ Remove Elements:

3. `pop()` – Removes the **last** element from the array.

Example: `arr.pop();`

1. `shift()` – Removes the **first** element from the array.

Example: `arr.shift();`

> 📌 All of these methods mutate (change) the original array.

📝 `slice()` – Immutable Array Method

🔷 **Purpose:**

The `slice()` The method is used to **copy** a portion of an array **without changing** the original array.

📌 **Key Points:**

- It does **not mutate** the original array.
- It returns a **new array**.
- Syntax: `array.slice(startIndex, endIndex)`
  - `startIndex` Where to start (inclusive)
  - `endIndex` Where to stop (exclusive)

✅ **Example:**

```
js
CopyEdit
const numbers = [10, 20, 30, 40, 50];
const sliced = numbers.slice(1, 4); // [20, 30, 40]
console.log(numbers); // [10, 20, 30, 40, 50] (unchanged)
```

📝 Array Destructuring in JavaScript

🔷 **What is it?**

Array destructuring allows you to **unpack** values from an array into **separate variables** in a clean and readable way.

Basic Syntax:

```
js
CopyEdit
const arr = [10, 20, 30];
const [a, b, c] = arr;

console.log(a); // 10
console.log(b); // 20
console.log(c); // 30
```

📌 Skip Elements:

```
js
CopyEdit
const arr = [1, 2, 3, 4];
const [first, , third] = arr;

console.log(first); // 1
console.log(third); // 3
```

📌 Default Values:

```
js
CopyEdit
const arr = [5];
const [x, y = 100] = arr;

console.log(x); // 5
console.log(y); // 100
```

📌 Destructure in Function Parameters:

```
js
CopyEdit
function printCoords([x, y]) {
  console.log(`X: ${x}, Y: ${y}`);
}

printCoords([10, 20]); // X: 10, Y: 20
```

Rest Operator (`...`)

📌 **Used to collect multiple values into a single variable (usually an array).**

🔷 In Function Parameters:

```
js
CopyEdit
function sum(...numbers) {
  return numbers.reduce((total, num) => total + num, 0);
}

sum(1, 2, 3); // 6
```

🧠 **It consolidates the remaining arguments into a single array.**

🔷 In Destructuring:

```
js
CopyEdit
const [a, b, ...rest] = [10, 20, 30, 40];
console.log(rest); // [30, 40]
```

Spread Operator (`...`)

📌 **Used to spread (break apart) elements from an array or object.**

🔷 For Arrays:

```
js
CopyEdit
const nums1 = [1, 2];
const nums2 = [3, 4];
const all = [...nums1, ...nums2];
console.log(all); // [1, 2, 3, 4]
```

🔷 For Objects:

```
js
CopyEdit
const obj1 = { a: 1 };
const obj2 = { b: 2 };
const combined = { ...obj1, ...obj2 };
console.log(combined); // { a: 1, b: 2 }
```

📝 `concat()` Method – For Combining Arrays

🔷 What it does:

The `concat()` The method is used to **merge two or more arrays**. It returns a **new array** and **does not change** the original one.

Basic Syntax:

```
js
CopyEdit
const newArray = array1.concat(array2, array3, ...);
```

🔷 Example:

```
js
CopyEdit
const fruits = ['apple', 'banana'];
const moreFruits = ['orange', 'mango'];

const allFruits = fruits.concat(moreFruits);
console.log(allFruits); // ['apple', 'banana', 'orange', 'mango']
```

`join()` – Convert Array to String

🔷 **Purpose:** Combines all elements of an array into a single string.

🔷 Syntax:

```
js
CopyEdit
array.join(separator)
```

🔷 Example:

```
js
CopyEdit
const words = ['I', 'love', 'coding'];
console.log(words.join(' ')); // "I love coding"
```

- Default separator is a **comma (,)** if not provided.
- Does **not** change the original array.

`fill()` – Fill Array with a Static Value

🔷 **Purpose:** Replaces all (or part of) the array with a single value.

🔷 Syntax:

```
js
CopyEdit
array.fill(value, startIndex, endIndex)
```

🔷 Example:

```
js
CopyEdit
const nums = [1, 2, 3, 4];
nums.fill(0, 1, 3);
console.log(nums); // [1, 0, 0, 4]
```

- **Mutates** the original array.
- Fills from `startIndex` to `endIndex` (not included).

`includes()` – Check if Value Exists in Array

🔷 **Purpose:** Returns `true` if the value exists in the array, `false` otherwise.

🔷 Syntax:

```
js
CopyEdit
array.includes(value)
```

🔷 Example:

```
js
CopyEdit
const colors = ['red', 'green', 'blue'];
```

```
console.log(colors.includes('green')); // true
console.log(colors.includes('yellow')); // false
```

`indexOf()` – First Occurrence

🔷 **Purpose:**

Finds the **first index** of a value in an array.

Returns `-1` if the value is **not found**.

🔷 Syntax:

```
js
CopyEdit
array.indexOf(value, startIndex)
```

`lastIndexOf()` – Last Occurrence

🔷 **Purpose:**

Finds the **last index** of a value in an array.

🔷 Syntax:

```
js
CopyEdit
array.lastIndexOf(value, fromIndex)
```

🔷 Example:

```
js
CopyEdit
const nums = [10, 20, 30, 20];
console.log(nums.lastIndexOf(20)); // 3
```

`reverse()` – Reverses the Array Order

🔷 **Purpose:**

Reverses the elements in the array **in-place** (changes the original array).

🔷 Syntax:

```
js
CopyEdit
array.reverse()
```

`sort()` – Sorts the Array

🔷 **Purpose:**

Sorts the array elements.

By default, it sorts **as strings in ascending (A-Z or 0-9)** order.

🔷 Syntax:

```
js
CopyEdit
array.sort(compareFunction)
```

🔷 Basic Example (strings):

```
js
CopyEdit
const names = ['Charlie', 'Alice', 'Bob'];
names.sort();
console.log(names); // ['Alice', 'Bob', 'Charlie']
```

🔷 Sorting Numbers (Correct Way):

```
js
CopyEdit
```

```
const nums = [10, 5, 20];
nums.sort((a, b) => a - b); // ascending
console.log(nums); // [5, 10, 20]
```

> ⚠️ Without a compare function, numbers sort wrongly as strings:
>
> `[10, 5, 20].sort()` → `[10, 20, 5]`

🔧 `splice()` Method in JavaScript

The `splice()` The method is used to **add, remove, or replace elements** in an array. It **modifies (mutates)** the original array and returns the removed elements (if any).

📌 Syntax:

```
js
CopyEdit
array.splice(start, deleteCount, item1, item2, ...)
```

Parameter	Description
`start`	Index at which to start changing the array
`deleteCount`	Number of elements to remove (0 if you just want to insert)
`item1, ...`	Optional. Elements to add at the `start` position (can add multiple items)

`at()`

`at()` Returns the element at a given index. It supports **negative indexing** (counts from the end).

✅ Syntax:

```
js
CopyEdit
array.at(index)
```

💡 Example:

```
js
CopyEdit
const arr = [10, 20, 30, 40];

console.log(arr.at(1));   // 20
console.log(arr.at(-1));  // 40 (last item)
```

---

copyWithin()

`copyWithin()` Copy part of an array **to another location in the same array**, without changing its length.

✅ Syntax:

```
js
CopyEdit
array.copyWithin(target, start, end)
```

Parameter	Description
`target`	Index to copy data to
`start`	Start index to copy from
`end`	(Optional) End index (not included)

💡 Example:

```
js
CopyEdit
let numbers = [1, 2, 3, 4, 5];

numbers.copyWithin(1, 3);
// Copies from index 3 to the end, into index 1
```

```
console.log(numbers); // [1, 4, 5, 4, 5]
```

`flat()`

`flat()` creates a new array with all **nested sub-arrays flattened** up to the specified depth.

✅ Syntax:

```
array.flat(depth)
```

- `depth` (optional) = how deep to flatten (default is `1`)

💡 Example:

```
js
CopyEdit
const nested = [1, 2, [3, 4, [5, 6]]];

console.log(nested.flat());      // [1, 2, 3, 4, [5, 6]]
console.log(nested.flat(2));     // [1, 2, 3, 4, 5, 6]
```

✅ `toReversed()` , `toSorted()` , and `toSpliced()` — introduced in **ES2023**

These methods **do NOT modify the original array** — instead, they return a **new, modified copy**, keeping the original array **immutable**.

🔷 1. `toReversed()`

Returns a **reversed copy** of the array without changing the original.

✅ Syntax:

```
js
CopyEdit
```

```
const newArray = array.toReversed();
```

💡 Example:

```
js
CopyEdit
const nums = [1, 2, 3];
const reversed = nums.toReversed();

console.log(reversed); // [3, 2, 1]
console.log(nums);     // [1, 2, 3] (unchanged)
```

🔷 2. `toSorted()`

Returns a **sorted copy** of the array. Accepts an optional **compare function**, just like `sort()` .

✅ Syntax:

```
js
CopyEdit
const newArray = array.toSorted(compareFn);
```

💡 Example:

```
js
CopyEdit
const nums = [4, 2, 1];
const sorted = nums.toSorted();

console.log(sorted); // [1, 2, 4]
console.log(nums);   // [4, 2, 1] (unchanged)
```

With the compare function:

```
js
CopyEdit
const desc = nums.toSorted((a, b) ⇒ b - a);
console.log(desc); // [4, 2, 1]
```

🔷 3. `toSpliced()`

Returns a **new array** with items **added/removed/replaced**, like `splice()`, but **does not mutate** the original.

✅ Syntax:

```
js
CopyEdit
const newArray = array.toSpliced(start, deleteCount, ...itemsToAdd)
```

💡 Example:

```
js
CopyEdit
const nums = [1, 2, 3, 4];

const spliced = nums.toSpliced(1, 2, 99);
console.log(spliced); // [1, 99, 4]
console.log(nums);    // [1, 2, 3, 4] (unchanged)
```

`with(index, newValue)`

✅ What it does:

Returns a **new array** where the element at the given index is **replaced** with the new value — **without modifying the original array**.

📘 Syntax:

```
js
CopyEdit
const newArray = array.with(index, newValue);
```

💡 Example:

```
js
CopyEdit
const fruits = ["apple", "banana", "mango"];

const newFruits = fruits.with(1, "orange");

console.log(newFruits); // ["apple", "orange", "mango"]
console.log(fruits);    // ["apple", "banana", "mango"] (unchanged)
```

⚠️ Notes:

- If `index` It is **out of bounds**, it throws a **RangeError**.

```
js
CopyEdit
fruits.with(5, "grape"); // ❌ RangeError
```

Static Array Methods in JavaScript

🟢 Dense Array:

A **dense array** has elements at **every index**, with no gaps.

```
js
CopyEdit
const denseArray = [1, 2, 3, 4];
console.log(denseArray.length); // 4
console.log(denseArray);       // [1, 2, 3, 4]
```

- ✅ All indexes `0` to `3` are filled.
- ✅ No missing or undefined "holes".

🔴 Sparse Array:

A **sparse array** has **missing elements** (holes) — indexes without values.

```
js
CopyEdit
const sparseArray = [1, , 3, , 5];
console.log(sparseArray.length); // 5
console.log(sparseArray);       // [1, <1 empty item>, 3, <1 empty item>, 5]
```

`Array.of()` – Create Array from Values

✅ Purpose:

The `Array.of()` The method creates a **new array** instance from the given **values**, regardless of how many values you pass.

🔷 Syntax:

```
js
CopyEdit
Array.of(element1, element2, ..., elementN)
```

`Array.of()` gives **predictable behavior** and avoids the confusion of `Array(length)`.

Array-like Objects

🔷 What is it?

An **array-like** object is **not a real array**, but it:

- Has **indexed elements** (`0`, `1`, `2`, ...)
- Has a **length** property

🔷 **Examples:**

- `arguments` inside functions
- `NodeList` from `document.querySelectorAll()`
- Strings

🔷 Example:

```
js
CopyEdit
function showArgs() {
  console.log(arguments); // array-like object
}
showArgs(1, 2, 3);
```

You **cannot** use array methods like `map()` or `forEach()` directly on array-like objects.

`Array.from()` – Convert Array-like or Iterable to Real Array

🔷 **Purpose:**

Creates a **real array** from array-like or iterable objects (like `arguments`, `NodeList`, `Set`, `Map`, etc.)

🔷 Syntax:

```
js
CopyEdit
Array.from(arrayLike, mapFn, thisArg)
```

🔷 Example 1 – Convert string to array:

```
js
CopyEdit
const str = "hello";
const chars = Array.from(str);
console.log(chars); // ['h', 'e', 'l', 'l', 'o']
```

🔷 Example 2 – With map function:

```
js
CopyEdit
const doubled = Array.from([1, 2, 3], x ⇒ x * 2);
console.log(doubled); // [2, 4, 6]
```

`Array.fromAsync()` – Asynchronous Version

🔷 Purpose:

Creates an array from an **async iterable**, like streaming or fetching data.

🔷 Syntax (🔒 Only in modern environments):

```
js
CopyEdit
await Array.fromAsync(asyncIterable)
```

🔷 Example:

```
js
CopyEdit
async function* generate() {
  yield 1;
  yield 2;
  yield 3;
}
```

```
const result = await Array.fromAsync(generate());
console.log(result); // [1, 2, 3]
```

> ⚠️ Array.fromAsync() works with asynchronous sources and returns a Promise.

`filter()` – Filter Elements

🔷 **Purpose:**

Creates a new array containing only the elements that **pass a condition**.

🔷 Syntax:

```
js
CopyEdit
array.filter(callback)
```

🔷 Example:

```
js
CopyEdit
const nums = [1, 2, 3, 4, 5];
const even = nums.filter(n => n % 2 === 0);
console.log(even); // [2, 4]
```

- **Does NOT mutate** the original array.
- Returns a **new array**.

`map()` – Transform Elements

🔷 **Purpose:**

Creates a new array by **modifying each element** of the original array.

🔷 Syntax:

```
js
CopyEdit
array.map(callback)
```

🔷 Example:

```
js
CopyEdit
const nums = [1, 2, 3];
const squared = nums.map(n ⇒ n * n);
console.log(squared); // [1, 4, 9]
```

- **Does NOT mutate** the original array.
- Returns a **new array** of the same length.

`reduce()` – Reduce to a Single Value

🔷 Purpose:

Processes all elements and returns **one final result** (sum, product, object, etc.)

🔷 Syntax:

```
js
CopyEdit
array.reduce(callback, initialValue)
```

🔷 Example – Sum:

```
js
CopyEdit
const nums = [1, 2, 3, 4];
const total = nums.reduce((acc, curr) ⇒ acc + curr, 0);
console.log(total); // 10
```

`Find()` – Find First Match

🔷 **Purpose:**

Returns the **first element** that matches a condition.

If nothing matches, it returns `undefined`.

🔷 Syntax:

```
js
CopyEdit
array.find(callback)
```

🔷 Example:

```
js
CopyEdit
const numbers = [5, 12, 8, 130, 44];
const found = numbers.find(num ⇒ num > 10);
console.log(found); // 12
```

- Stops at the **first match**
- Returns the **element itself**, not the index

`some()` – Check If At Least One Matches

🔷 **Purpose:**

Returns `true` If **any** element passes the condition, otherwise `false`.

🔷 Syntax:

```
js
CopyEdit
array.some(callback)
```

🔷 Example:

```
js
CopyEdit
const nums = [1, 3, 5, 8];
const hasEven = nums.some(n ⇒ n % 2 === 0);
console.log(hasEven); // true
```

- Stops as soon as it finds a match

`every()` – Check If All Match

🔷 Purpose:

Returns `true` Only if **every** element passes the condition.

🔷 Syntax:

```
js
CopyEdit
array.every(callback)
```

🔷 Example:

```
js
CopyEdit
const nums = [2, 4, 6];
const allEven = nums.every(n ⇒ n % 2 === 0);
console.log(allEven); // true
```