JavaScript Foundation – Training Day-4

Course content and duration:

Duration: 16 hours | Schedule: 8 days @ 2 hours/day

S. No	Day	Module	Topics
1	WED (10/12/2025)	Module 1: Introduction to JavaScript and Basics	JavaScript Overview, Syntax, and Variables
2	FRI (12/12/2025)	Module 2: Control Flow and Loops	Conditional Statements and Iteration
3	MON (15/12/2025)	Module 3: Functions and Scope	Function Declaration, Expressions, and Scope
4	TUE (16/12/2025)	Module 4: Arrays and Array Methods	Array Manipulation and Higher-Order Functions
5	WED (17/12/2025)	Module 5: Objects and Object-Oriented Programming	Objects, Properties, Methods, and Prototypes
6	THU (18/12/2025)	Module 6: DOM Manipulation and Events	Document Object Model and Event Handling
7	FRI (19/12/2025)	Module 7: Asynchronous JavaScript	Callbacks, Promises, and Async/Await
8	MON (22/12/2025)	Module 8: ES6+ Features and Best Practices	Modern JavaScript Features and Code Quality

Day4: Work effectively with arrays and array methods (JavaScript)

Contents
Creating and accessing arrays
Array methods: push, pop, shift, unshift
Array iteration: forEach, map, filter, reduce
Array searching: find, findIndex, includes
Array transformation: slice, splice, concat, join
Sorting and reversing arrays
Spread operator with arrays
Array destructuring

- Understand **what arrays are and why they are needed**
- Confidently **create, access, modify, and traverse arrays**
- Use **modern array methods** instead of traditional loops
- Distinguish between **mutating and non-mutating methods**

Creating and Accessing Arrays (Expanded Explanation)

Why Arrays Exist

In real applications, we rarely deal with single values.
Examples:

- List of students
- Product prices
- User names
- Marks of a student

Without arrays, we would need many variables:

```
let mark1 = 80;
let mark2 = 85;
let mark3 = 90;
```

This becomes **hard to manage**, **hard to loop**, and **error-prone**.

Arrays solve this problem by grouping related values together.

Creating Arrays

```
let marks = [80, 85, 90];
```

- Square brackets [] indicate an array
- Values are separated by commas
- Arrays can store **any data type**

```
let mixed = [10, "Hello", true, null];
```

Accessing Array Elements

Arrays are **zero-indexed**:

- First element → index 0
- Second element → index 1

```
console.log(marks[0]); // 80
console.log(marks[2]); // 90
```

Index is NOT the count.
Index = position − 1

Updating Array Values

Arrays are **mutable** (can be changed):

```
marks[1] = 88;
console.log(marks); // [80, 88, 90]
```

Array Length

```
console.log(marks.length); // 3
```

Explain:

- length gives total number of elements
- Last index = length - 1

```
let marks = [80, 85, 90];

marks[1] = 88;
console.log(marks);
console.log(marks.length);
```

Next topic: Array methods: push, pop, shift, unshift

Array Methods: push, pop, shift, unshift

These are **basic building blocks** of array manipulation.

push() – Add at the End

```
let queue = ["A", "B"];
queue.push("C");
```

- Adds element at the **end**
- Changes original array
- Returns new length

pop() – Remove from the End

```
let queue = ["A", "B"];
queue.pop();
```

- Removes **last element**
- Useful in **stack operations (LIFO)**

unshift() – Add at the Beginning

```
queue.unshift("Start");
```

- Inserts element at index 0
- Shifts all elements to the right
- Slower for large arrays

shift() – Remove from the Beginning

```
queue.shift();
```

- Removes first element
- All elements shift left


```
let queue = ["A", "B", "C", "D"];
queue.push("E");
console.log('push', queue); // Adds element at the end
queue.pop();
console.log('pop', queue); // Removes last element
queue.unshift("a");
console.log('unshift', queue); // Inserts element at index 0
queue.shift();
console.log('shift', queue); // Removes first element
```

Comparison

Method	Adds / Removes	Position	Common Use
push	Add	End	Stack
pop	Remove	End	Undo
unshift	Add	Start	Priority items
shift	Remove	Start	Queue

If performance matters → avoid shift/unshift on big arrays.

Next topic: Array iteration: forEach, map, filter, reduce

Array Iteration Methods

Iteration means **going through each element**.

for()

```
let arr = [20,30,40];
for (let i = 0; i < arr.length; i++) {
  console.log(arr[i]);
}
```

forEach()

```
[5,3,8].forEach(item => {
  console.log(item);
});
```

map() – Transform Data

```
let arr = [20, 30, 40];
let squared = arr.map(n => n * n);
```

Concept:

- Takes input array
- Transforms each element
- Returns **new array**
- Original remains unchanged

Real-world use:

- Convert prices to GST prices
- Convert names to uppercase
- Convert API data for UI

```
const names = ["john", "alice", "michael", "sara"];
const upperCaseNames = names.map(name => name.toUpperCase());
console.log(upperCaseNames);
```

filter() – Select Subset

```
let ages = [30, 20, 50, 60, 12];
let adults = ages.filter(age => age >= 18);
console.log(adults);
```

Concept:

- Condition returns true or false
- Only true values remain
- Original array untouched

reduce() – Most Powerful Method

reduce() can return any data structure (object, array, number)

```
let nums = [5, 8, 10];
let total = nums.reduce((sum, n) => sum + n, 0);
console.log(total);
```

- sum → accumulator
- n → current element
- 0 → initial value

Use cases:

- Sum
- Average
- Grouping data
- Counting occurrences

reduce collapses an array into a single result.

Example: Grouping Numbers by Even and Odd

```
const numbers = [1, 2, 3, 4, 5, 6];

const grouped = numbers.reduce((result, num) => {
  if (num % 2 === 0) {
    result.even.push(num);
  } else {
    result.odd.push(num);
  }
  return result;
}, { even: [], odd: [] });

console.log(grouped);
```

result → accumulator object

num → current element

Initial value → { even: [], odd: [] }

Elements are **grouped into properties**

Next topic: Array searching: find, findIndex, includes

Array Searching Methods

includes() – Check if value exists

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

console.log(fruits.includes("banana")); // true
console.log(fruits.includes("orange")); // false
```

- Simple boolean check (true/false)
- Best for simple existence check

indexOf() – Find index of a value

```
const numbers = [10, 20, 30, 20];

console.log(numbers.indexOf(20)); // 1
console.log(numbers.indexOf(50)); // -1
```

Key Points

- Returns first index
- Returns -1 if not found

lastIndexOf() – Find last occurrence

```
const numbers = [1, 2, 3, 2, 4];

console.log(numbers.lastIndexOf(2)); // 3
```

find() – Find first matching element

```
const users = [
  { id: 1, name: "Ravi" },
  { id: 2, name: "Anita" },
  { id: 3, name: "Kiran" }
];
const user = users.find(u => u.id === 2);
console.log(user);
```

- Returns **element itself**
- Returns **undefined if not found**
- Stops after first match

findIndex() – Find index using condition

```
const users = [
  { id: 1, name: "Ravi" },
  { id: 2, name: "Anita" }
];

const index = users.findIndex(u => u.name === "Anita");
console.log(index); // 1
```

- Returns index
- Returns -1 if not found

some() – Check if **any** element matches

```
const ages = [18, 21, 16, 25];

const hasMinor = ages.some(age => age < 18);
console.log(hasMinor); // true
```

every() – Check if **all** elements match

```
const scores = [80, 90, 75];

const allPassed = scores.every(score => score >= 50);
console.log(allPassed); // true
```

- Use **find()** for objects
- Use **includes()** for simple values
- Use **some()** / **every()** for validations

Next topic: Array Transformation Methods

Array Transformation Methods

slice() – Extract part of array (non-mutating)

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

const selected = fruits.slice(1, 3);

console.log(selected); // ["banana", "mango"]
```

Explain parameters:

slice(start, end)

- Extracts portion
- End index NOT included
- Does NOT modify original

splice() – Add / remove elements (mutates array)

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

fruits.splice(1, 1, "orange", "grapes");

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

Explain parameters:

splice(startIndex, deleteCount, itemsToAdd)

⚠ Mutates array

concat()– Merge arrays

```
const a = [1, 2];
const b = [3, 4];
const combined = a.concat(b);

console.log(combined);
```

- Combines arrays
- Returns new array

flat() – Flatten nested arrays

```
const nested = [1, [2, [3, 4]]];

const flatArray = nested.flat(2);

console.log(flatArray);
```

join() - Convert array to string

```
const words = ["JavaScript", "is", "awesome"];

const sentence = words.join(" ");

console.log(sentence);
```

words.join("-");

- Converts array to string
- Common in CSV, display text

sort() – Rearrange elements (mutates)

```
const numbers = [30, 5, 100];

numbers.sort((a, b) => a - b);
console.log(numbers);
```

Next topic: Sorting and Reversing Arrays in JavaScript

Sorting and Reversing Arrays in JavaScript

Sorting and reversing are **core array operations** used in data processing, UI lists, tables.

sort() – Sort array elements

Default behavior (❗important)

```
const numbers = [10, 5, 40, 25];

numbers.sort();
console.log(numbers);
```

sort() converts elements to **strings by default**

Sorting Numbers (Correct Way)

```
const numbers = [10, 5, 40, 25];

numbers.sort((a, b) => a - b);
console.log(numbers);
```

Descending Order

```
numbers.sort((a, b) => b - a);
```

Sorting Strings

```
const names = ["Ravi", "anita", "Kiran", "suresh"];
names.sort();
console.log(names);
```

Case-insensitive sort

```
const names = ["Ravi", "anita", "Kiran", "suresh"];
names.sort((a, b) =>
  a.toLowerCase().localeCompare(b.toLowerCase())
);
```

Sorting Objects by Property (Very Common)

Example: Sort users by age

```
const users = [
  { name: "Ravi", age: 30 },
  { name: "Anita", age: 25 },
  { name: "Kiran", age: 35 }
];

users.sort((a, b) => a.age - b.age);

console.log(users);
```

reverse() – Reverse array order

```
const numbers = [1, 2, 3, 4];

numbers.reverse();

console.log(numbers);
```

Reverse Without Mutating Original Array

```
const original = [1, 2, 3];

const reversed = [...original].reverse();

console.log(reversed);
console.log(original);
```

Sorting + Reversing Together

```
const scores = [70, 90, 60, 85];

const sortedDesc = scores.sort((a, b) => a - b).reverse();

console.log(sortedDesc);
```

⚠ Important Notes

- sort() **mutates** the original array
- reverse() **mutates** the original array
- Always **copy array first** if mutation is not allowed

Operation	Method	Mutates?
Sort numbers	sort(compareFn)	✅
Sort strings	sort()	✅
Reverse order	reverse()	✅
Safe reverse	[...arr].reverse()	❌

Next topic: Spread Operator with Arrays

Spread Operator (...) with Arrays in JavaScript

The **spread operator (...)** expands array elements and is widely used for **copying, merging, and transforming arrays**.

Copy an Array (Shallow Copy)

```
const original = [1, 2, 3];

const copy = [...original];

console.log(copy);
```

Why use spread?

- Avoids mutating the original array
- Better than assigning (=), which copies reference

Merge (Concatenate) Arrays

```
const a = [1, 2];
const b = [3, 4];
const merged = [...a, ...b];
console.log(merged);
```

Add Elements While Copying

```
const numbers = [2, 3, 4];
const updated = [1, ...numbers, 5];
console.log(updated);
```

Spread with Math Functions

```
const numbers = [10, 5, 30];
const max = Math.max(...numbers);
const min = Math.min(...numbers);
console.log(max, min);
```

Convert String to Array

```
const word = "HELLO";
const letters = [...word];
console.log(letters);
```

Remove Duplicates

```
const nums = [1, 2, 2, 3, 3, 4];
const unique = [...new Set(nums)];
console.log(unique);
```

Spread with Nested Arrays

```
const arr = [[1], [2], [3]];
const copy = [...arr];
copy[0].push(99);
console.log(arr);
```

Output

[[1, 99], [2], [3]]

Spread makes a **shallow copy** — nested objects/arrays still share reference.

Spread in Function Arguments

```
function sum(a, b, c) {
  return a + b + c;
}
const nums = [10, 20, 30];
console.log(sum(...nums));
```

Next topic: Array Destructuring

Array Destructuring in JavaScript

Array destructuring lets you **extract values from arrays** and assign them to variables in a **clean and readable way** and it **improves performance indirectly** in real applications.

Basic Array Destructuring

```
const colors = ["red", "green", "blue"];

const [first, second, third] = colors;

console.log(first);  // red
console.log(second); // green
console.log(third);  // blue
```

Skip Elements

```
const numbers = [10, 20, 30, 40];

const [first, , third] = numbers;

console.log(first); // 10
console.log(third); // 30
```

Default Values

```
const scores = [85];

const [math = 0, science = 0] = scores;

console.log(math);    // 85
console.log(science); // 0
```

Swap Variables

```
let a = 5;
let b = 10;

[a, b] = [b, a];

console.log(a, b);
```

Using Rest Operator (...) in Destructuring

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

const [first, second, ...rest] = fruits;

console.log(first); // apple
console.log(second); // banana
console.log(rest); // ["mango", "orange"]
```

Destructuring in Function Return

```
function getCoordinates() {
  return [12.97, 77.59];
}

const [lat, lng] = getCoordinates();

console.log(lat, lng);
```

Destructuring in Loops

```
const pairs = [[1, 2], [3, 4], [5, 6]];

for (const [a, b] of pairs) {
  console.log(a, b);
}
```

Nested Array Destructuring

```
const data = [1, [2, 3]];

const [x, [y, z]] = data;

console.log(x, y, z);
```

Use **array destructuring** to write **cleaner and more readable code**, especially with function returns.