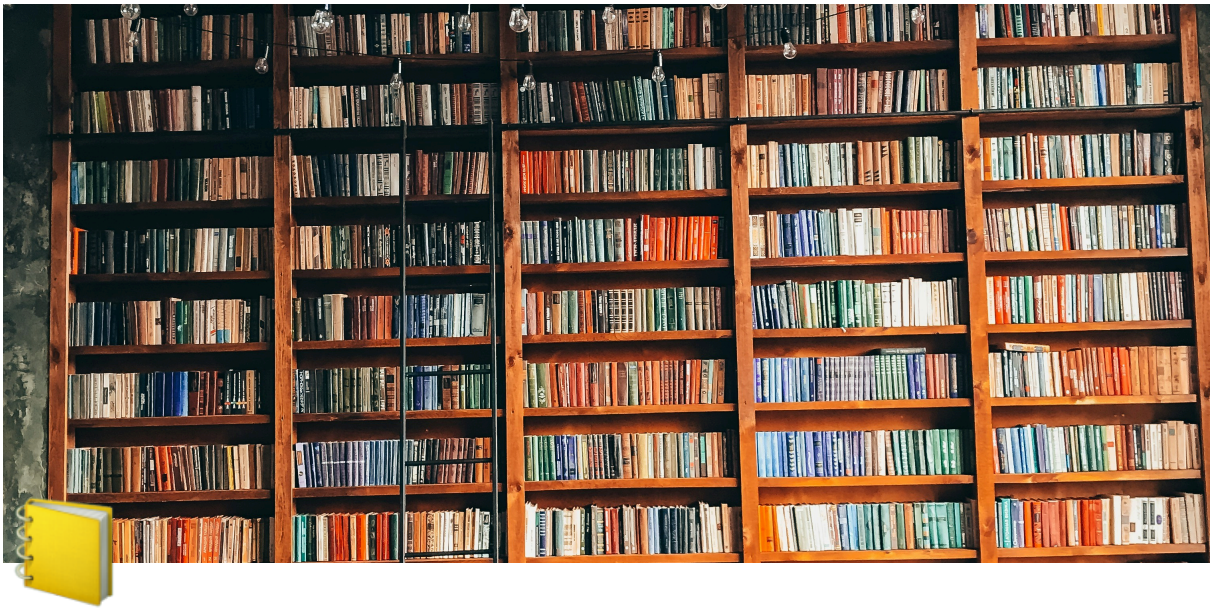

Javascript Programming

Basic Concepts

How Javascript Works & Execution Context ?

- → "Everything in JavaScript happens inside an Execution Context" means that JavaScript divides everything into two parts :
- Memory (Variable Environment) : This is where variables, functions and function definitions are stored in key-value pairs.
- Code Execution : This is the part where the code runs line by line, following a single-threaded execution model.
- Javascript is a synchronous single-threaded language.
- Javascript is scripting (client-side scripting) language.

How Javascript code executed ?

- Basically, whenever the execution start then two things happens or we can say that the execution complete in two phases first Memory Creation Phase and second is that Code Execution Phase.
- In the Memory Creation Phase, all the variables and function definition store in the memory.

- After completing the memory creation phase, then start the code execution phase and executed the code.
- Function in Javascript is like a heart and Javascript handle the function in a different way compare to another programming languages.
- “Call stacks maintains the order of execution of execution contexts.”

Concepts of Hoisting in Javascript

- It is a process to moving all the function declaration to the top of file of JS code.
- It’s a depth concept and we can’t understand easily and only we understand the concept using the example.
- Before understand this concept , we need to know the concept of memory creation and code execution phase because it’s required to understand hoisting concept.

Do we need a compiler for executing the JS Code?

- We need JS Engine for executing JS program.
- Firefox browser → Spider Monkey JS Engine
- Chrome browser → V8 JS Engine

Scope, Scope Chaining & Lexical Environment in JavaScript

🔷 1. What is Scope in JavaScript?

Scope means **where a variable or function is accessible** in your code.

Types of Scope in JavaScript:

1. **Global Scope** → Variables accessible everywhere.
2. **Local/Function Scope** → Variables accessible only inside a function.
3. **Block Scope (ES6 - `let`, `const`)** → Variables accessible only inside `{}`.

Example of Scope:

```
let globalVar = "I am Global";  // Global Scope

function testFunction() {
```

```
    let localVar = "I am Local";  // Function Scope
    console.log(globalVar);  // ✅ Accessible
    console.log(localVar);   // ✅ Accessible
}

console.log(globalVar);  // ✅ Accessible
console.log(localVar);   // ❌ Error (localVar is inside function)
testFunction();
```

🔷 2. What is Scope Chaining?

Scope Chaining means that **JavaScript searches for a variable in the current scope, if not found, it moves up to the parent scope, and so on until it reaches the global scope.**

Example of Scope Chaining:

```
let globalVar = "I am Global";

function outerFunction() {
    let outerVar = "I am Outer";
    function innerFunction() {
        let innerVar = "I am Inner";
        console.log(innerVar);  // ✅ Found in innerFunction
        console.log(outerVar);  // ✅ Found in outerFunction
        console.log(globalVar); // ✅ Found in Global Scope
    }
    innerFunction();
}

outerFunction();
```

🔷 How JavaScript searches for variables?

1. **innerFunction** → First searches in its own scope.
2. **If not found** → Moves up to **outerFunction**.
3. **If still not found** → Moves to **global scope**.
4. **If not found anywhere** → JavaScript throws an **error**.

🔷 3. What is Lexical Environment?

Lexical Environment is a **combination of the current scope and the reference to its outer scope (parent scope).**

Every time a function is created, a new **Lexical Environment** is created.

Example of Lexical Environment:

```
function outer() {
    let a = 10;

    function inner() {
        let b = 20;
        console.log(a);  // ✅ a is accessible due to Lexical Environment
    }

    inner();
}
outer();
```

✅ **Lexical Environment of `inner()` includes:**

1. **Its own variables** (`b = 20`).
2. **Reference to `outer()` variables** (`a = 10`).
3. **Reference to global scope.**

💬 Q: What is the Temporal Dead Zone in JavaScript?

✅ *"Temporal Dead Zone (TDZ) is the period between when a variable is declared using `let` or `const` and when it is initialized. Accessing the variable in this zone results in a `ReferenceError`."*

💬 Q: Why does `let` and `const` have a TDZ but `var` does not?

✅ *"`let` and `const` are hoisted but not initialized, so accessing them before assignment causes an error. `var` is hoisted and initialized with `undefined`, so it doesn't have a TDZ."*

Type of Data-Type

1. Primitive Type
2. Reference Type

Primitive Type

- String → sequence of character
- Number → Integer values, Negative values , Decimal values
- Boolean → true or false
- Undefined → variable declare kia hai but define nahi kia hai
- Null → value empty hai but define hai variable

Reference Type

- Object → Multiple linked values ko ek top level entity me daal dete hai..
- Array → DS used to contain a list of different type of items.

Object in Javascript

- Collection of key-value pair

Object Creation - How ?

→ Using Factory Function

→ Using Constructor Function

Using Factory function → Object Creation

- It is a type of function where built the object and return the object.

```
function createRectangle(len, bred) {
        return rectangle = {
            length : len,
            breadth : bred,

            area : function() {
               let calc = length * breadth;
               console.log(calc);
            }
        }
```

```
}

let obj1 = createRectangle(2, 3);
```

Using Constructor function → Object Creation

- Constructor function should the nomenclature we use the Pascal case.
- Inside these function we only initialise the properties and define the method and isn't return anything only initializing all the data.
- New Keyword ek empty object return karta hai

```
function Rectangle(len, bred) {
        this.len = len,
        this.bred = bred,

        this.area = () => {
            console.log("Area is that");
        }
}

let obj1 = new Rectanle(5, 10);
```

Constructor Property

→ Javascript ke andar kisi v object ki ek property hoti hai constructor jo yeh define karta hai ki yeh object bana kaise hain.

Iterating through objects

→ For-in loop

→ For-of loop (yeh kewal iterables par use kar skte hai - arrays and maps)

Object Cloning

1. Using Iteration

```
let src = { val : 10 };
let dest = {};
```

```
for (let key in src) {
    dest[key] = src[key];
}
```

2. Using Assign method

```
let dest = Object.assign({}, src);
```

3. Using Spread operator

```
let dest = {...src}
```

String Datatype

We can define a string in JavaScript in two ways:

1. **String as a primitive data type** – using string literals (`""` or `''`).
2. **String as an object** – using the `String` constructor (`new String("")`).

The choice depends on how you declare the string.

```
//String as a primitive data-type
let primitiveType = "Primitive DataType";
console.log(primitiveType);
//String as an object data-type
let objectType = new String("Object Type");
console.log(objectType);
```

Arrays

→ It is a collection of different type of values.

→ It is a reference and object type.

Creation of Arrays

```
let nums = [1,2,3];
```

Adding an Elements

→ At Ending Position

```
nums.push(9);
//op - [1,2,3,9]
```

→ At Beginning Position

```
nums.unshift(0);
//op - [0,1,2,3,9]
```

→ At Random Position

```
//syntax - arrayName.splice(index, deleteCount, values)
nums.splice(1, 0, 'a','b','c');
//op - [0,'a','b','c',1,2,3,9]
```

Searching Elements

→ Using indexOf method

```
console.log(nums.indexOf('a')); // 2
console.log(nums.indexOf('z')); // -1

if(nums.indexOf('a') != -1)
        console.log('present');

console.log(nums.includes('a'));
/* true { It's a best approach to
find the values present or not } */
```

Removing an Elements

→ Ending Element

```
//nums - [0,'a','b','c',1,2,3,9]
nums.pop(); // 9 popped element
//nums - [0,'a','b','c',1,2,3]
```

→ Starting Element

```
//nums - [0,'a','b','c',1,2,3]
nums.shift(); // 0 popped element
//nums - ['a','b','c',1,2,3]
```

→ Random Position

```
//syntax - arrayName.splice(index, deleteCount)
//nums - ['a','b','c, 1,2,3]
nums.splice(2, 1); // 0 popped element
//nums - ['a','b',1,2,3]
```

Concept of Empty an array

```
let numbers = [1,2,3,4]; //Array is a object and also reference datatype
let numbers2 = numbers;

/* numbers = []; → numbers array to empty hai it's means yeh
unused variable ho gya to gabage collector ise toh automatically
delete kar dega...but numbers2 abhi v point kar raha hai numbers array
ko so yaha par issue aata hai..*/

numbers.length = 0; //Best Approach
numbers.splice(0, numbers.length());
```

Concept of Combining and Slicing arrays

→ Using Concat Method and Spread Operator

```
let arr1 = [1,2,3,4];
let arr2 = [8,9,7,6];

//Using concat method
let combined = arr1.concat(arr2); // [1,2,3,4,8,9,7,6];

//Using spread method
let combined = [...arr1, ...arr2];
```

```
//Access of some parts
let newSlicePart = combined.slice(2,6); // [3,4,8,9,7]
```

Concept of Join and Split arrays

→ Join method ek string return karta hai

→ Split method ek object return karta hai

```
let nums = [1,2,3,4,5,6,7,8,9];

//Using JOIN method
let joined = nums.join(','); // [1,2,3,4,8,9,7,6];
console.log(joined); // 1,2,3,4,5,6
console.log(typeOf(joined)); // string type

//Using SPLIT method
let spliting = joined.split(',');  //1,2,3,4,5,6
console.log(spliting); // ['1','2',...]
console.log(typeOf(spliting)); //Object Type
```

Functions

→ A block of code that fulfulls a specific task.

→ Two types of function assignment first is that Named Function Assignment and second is that Anonymous Function Assignment.

```
//Named Function Assignment
let rectangle = function area(){
   console.log("Area of rectangle");
}
//Invoke → rectangle()

//Anonymous Function Assignment
let square = () => {
   console.log("Area of square");
```

```
}
//Invoke → square
```

Functions Types

```
//1. Function Statement also known as(aka) Function Declaration
function a(){
    console.log("Function Statement");
}

//2. Function Expression
let v = function b(){
    console.log("Function Expression");
}

/*Function Statement aur Function Expression mein difference hain Hoistin
g ka
means function a() and v varible memory creation phase me a() function de
fine
ho jaayega aur humara v normal undefined hoga. So, then if we call the a()
call karte hai function definition se phele toh koi issue nhi aayega but whi
hum baat kare v call karenge function definiton se phele then give a error
because of Hoisting. */

//3. Anonymous Function
//→ Whenever define any function without naming then is called a anonym
ous func.

//Arguments → The value which we passed inside the function is called arg
um.
//Parameter → The label and identifier which get those these values is calle
d
//parameter.

//4. First Class Function
/*→ The ability of function to be used as values and can be passed as a
```

argument in an another function and can be returned function is this ability known as a first class function in javascript.

Function Statement (Function Declaration)

A **Function Statement**, we also called a **Function Declaration**, is when we define a function using the `function` keyword **without assigning it to a variable**.

🔷 **Example:**

```
function a() {
    console.log("Function Statement");
}
a();
```

✔️ **Function declarations are hoisted**, meaning they are moved to the top during memory creation, so you can call them before they are defined.

Function Expression

A **Function Expression** is when we **assign a function to a variable**.

🔷 **Example :**

```
let v = function b() {  // Function assigned to variable v
    console.log("Function Expression");
};
v();
```

✔️ Unlike function declarations, **function expressions are NOT hoisted**.

✔️ If you try to call `v()` before defining it, you'll get an **error**.

🔷 **Example of Hoisting Issue:**

```
v();  // ❌ ReferenceError: Cannot access 'v' before initialization
let v = function() {
    console.log("Function Expression");
};
```

✔️ The variable `v` is hoisted, but **it is assigned** `undefined`, so calling `v()` before initialization results in an error.

Anonymous Function

An **Anonymous Function** is a function **without a name**. These functions are usually assigned to a variable or used in callbacks.

🔷 **Example:**

```
let fun = function() {
    console.log("Anonymous Function");
};
fun();
```

✔️ You **cannot use anonymous functions as standalone function declarations** because they don't have a name.

🔷 **Invalid Example (❌ Syntax Error):**

```
function () {  // ❌ Error: Function name is missing
    console.log("Invalid Anonymous Function");
}
```

✔️ Anonymous functions are useful for **callbacks** and **function expressions**.

Arguments vs Parameters

🔷 **What is an Argument?**

- The **actual values** passed when calling a function.

🔷 **What is a Parameter?**

- The **variables that receive the arguments inside the function definition**.

🔷 **Example:**

```
function greet(name) {  // 'name' is a parameter
    console.log("Hello, " + name);
}

greet("John");  // "John" is an argument
```

✔️ Here, `"John"` is an **argument**, and `name` is a **parameter**.

First-Class Functions (First-Class Citizens)

🔷 **What is a First-Class Function?**

- JavaScript allows functions to be **treated like values**.
- This means you can **pass functions as arguments**, **return functions from other functions**, and **assign functions to variables**.

🔷 **Example:**

```
// Function assigned to a variable
let sayHello = function() {
    return "Hello!";
};

// Function passed as an argument
function greet(func) {
    console.log(func());  // Calls sayHello()
}

greet(sayHello);  // ✅ Output: "Hello!"
```

✔️ Functions **can be stored in variables, passed as arguments, and returned from other functions** – this is **First-Class Function Behavior**.

What is Higher Order Function ?

→ A function which take another function as an argument or returns a function from it is known as Higher Order Function.

```
//x is a Callback function
function x(){
    console.log("X logged");
}

//y is a Higher Order Function
function y(x){
    x();
}
```

What is Closure ?

→ Functions along with its lexical scope bundled together forms a closure.

→ A closure is a function that retains access to variables from its outer scope even after the outer function has finished executing.

→ When they are returned another this still maintains their lexical scope, they remember where they are actually present.

→ When we are returned code just not returned but a closure was returned. Closure was enclosed function along with its lexical scope and that was returned.

```
function x(){
    let a = 7;
    function y(){
        console.log(a);
    }
    return y;
}

var z = x();
//After writing 10000 Lines of code

console.log(z);
z();
```

- *"Here, `x` defines a variable `a = 7` and an inner function `y` that logs `a`."*
- *"When I call `x()`, it returns the function `y` and assigns it to `z`."*
- *"At this point, even though `x()` has finished execution, the inner function `y` still has access to `a`."*
- *"That is because `y` forms a closure — it keeps the reference of its lexical scope alive."*
- *"So, when I call `z()`, it prints `7`."*

Why are Closures useful ?

→ Closures help in data privacy, maintaining state in functions, and handling asynchronous operations like loops and event listeners.

Can you give a example of a Closure ?

```
function greet(name) {
    return function () {
        console.log("Hello, " + name);
    };
}
const sayHello = greet("Alice");
sayHello(); // ✅ Hello, Alice
```

"Here, `sayHello` is a closure that remembers `name = "Alice"`."

Closures in Loop

Closures **help preserve values inside loops**, avoiding unexpected results.

Without Closure (Issue in `var`):

```
for (var i = 1; i <= 3; i++) {
    setTimeout(() => {
        console.log(i);
    }, 1000);
}
// ❌ Output after 1 second: 4, 4, 4 (not 1, 2, 3)
```

💡 **Why?**

- `var` is function-scoped, so all `console.log(i)` calls **share the same** `i` after the loop finishes.

Using Closures (Fix with `let` or an IIFE):

```
for (let i = 1; i <= 3; i++) {
    setTimeout(() => {
        console.log(i);
    }, 1000);
}
// ✅ Output after 1 second: 1, 2, 3 (correct)
```

- `let` creates a **new** `i` **for each iteration**, so it remembers the correct values.

Class & Objects

What is a Class?

- A **class** in JavaScript is like a **blueprint** (design or template).
- It tells us **how an object should look** and **what it can do**.
- But a class itself is not a real thing; it's just a design.

What is an Object?

- An **object** is a **real thing** made using a class.
- If a class is a **blueprint**, an object is the **actual house/car** built from it.
- Objects have **properties (data)** and **methods (functions)**.

Writing Class in JavaScript

```
// Define a Class
class Car {
  constructor(brand, color) {
    this.brand = brand;   // property
    this.color = color;   // property
  }

  start() {
    console.log(`${this.brand} is starting...`);
  }

  stop() {
    console.log(`${this.brand} is stopping...`);
  }
}
```

Creating Objects from Class

```
// Create objects
const car1 = new Car("Honda", "Red");
const car2 = new Car("BMW", "Blue");
```

```
// Use objects
car1.start();   // Honda is starting...
car2.start();   // BMW is starting...

console.log(car1.color); // Red
console.log(car2.brand); // BMW
```

⚡ Example with a real-life analogy:

```
class Student {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }

  introduce() {
    console.log(`Hi, I'm ${this.name} and I'm ${this.age} years old.`);
  }
}

const s1 = new Student("Sumit", 22);
const s2 = new Student("Rahul", 21);

s1.introduce(); // Hi, I'm Sumit and I'm 22 years old.
s2.introduce(); // Hi, I'm Rahul and I'm 21 years old.
```

Prevent Object Mutation

As seen in the previous challenge, `const` declaration alone doesn't really protect your data from mutation. To ensure your data doesn't change, JavaScript provides a function `Object.freeze` to prevent data mutation.

Any attempt at changing the object will be rejected, with an error thrown if the script is running in strict mode.

```
let obj = {
  name:"FreeCodeCamp",
  review:"Awesome"
};
```

```
Object.freeze(obj);
obj.review = "bad";
obj.newProp = "Test";
console.log(obj);
```

Use Destructuring Assignment to Extract Values from Objects

Destructuring assignment is special syntax introduced in ES6, for neatly assigning values taken directly from an object.

Consider the following ES5 code:

```
const user = { name: 'John Doe', age: 34 };

const name = user.name;
const age = user.age;
```

`name` would have a value of the string `John Doe`, and `age` would have the number `34`.

Here's an equivalent assignment statement using the ES6 destructuring syntax:

```
const { name, age } = user;
```

Again, `name` would have a value of the string `John Doe`, and `age` would have the number `34`.

Here, the `name` and `age` variables will be created and assigned the values of their respective values from the `user` object. You can see how much cleaner this is.

You can extract as many or few values from the object as you want.

Shallow Copy & Deep Copy

1) Shallow Copy

Copies only the **first level**. Nested objects/arrays are **shared** (same reference).

Common shallow copy ways

```
// objects
const o1 = { x: 1, inner: { y: 2 } };
const s1 = { ...o1 };              // using spread operator
// const s1 = Object.assign({}, o1);  // using assign method

// arrays
const a1 = [1, { y: 2 }];
const s2 = [...a1];          // or a1.slice(), Array.from(a1)
```

---

2) Deep Copy

Recursively copies **all nested levels**, so the clone is fully independent.

A) `structuredClone` (best built-in)

```
const original = { d: new Date(), map: new Map([["k", 1]]), a: [1,2,{z:3}] };
const deep = structuredClone(original);

deep.a[2].z = 9;
console.log(original.a[2].z); // 3 (independent)
```

Shallow vs Deep: quick visual

```
const src = { user: { name: "Sumit" }, tags: ["js", "node"] };

// Shallow
const shallow = { ...src };
shallow.user.name = "Aman";
shallow.tags.push("web");
console.log(src.user.name); // "Aman"  (oops)
console.log(src.tags);      // ["js","node","web"] (oops)

// Deep
const deep = structuredClone(src);
deep.user.name = "Riya";
deep.tags.push("db");
```

```
console.log(src.user.name); // "Aman" (unchanged by deep)
console.log(src.tags);      // ["js","node","web"]
```

JSON

What is JSON?

- JSON stands for **JavaScript Object Notation**.
- It is a **lightweight data format** used to store and exchange data between systems.
- It looks like a JavaScript object, but it is always written as a **string**.

🔑 JSON Methods in JavaScript

1. `JSON.stringify()`

👉 Converts a JavaScript object/array into a JSON string.

```
const user = { name: "Sumit", age: 22, isStudent: true };
const jsonString = JSON.stringify(user);

console.log(jsonString);
// Output: '{"name":"Sumit","age":22,"isStudent":true}'
```

⚡ **Real-world use**:

When you send data to a **server** (like filling a form and submitting), you convert it into JSON string first.

2. `JSON.parse()`

👉 Converts a JSON string back into a JavaScript object.

```
const jsonData = '{"name":"Sumit","age":22,"isStudent":true}';
const userObject = JSON.parse(jsonData);

console.log(userObject.name);
// Output: "Sumit"
```

⚡ **Real-world use**:

When your app receives data from an **API**, it usually comes as JSON string.

You need to parse it to use it in JS.

🚀 Real-World Use Case

- **Frontend → Backend:** When you submit a form on a website, the browser converts your details into JSON and sends it to the server.
- **Back-end → Front-end:** When you fetch data from an API (like weather or stock prices), it comes as JSON, and you parse it to use in JavaScript.

What is a Higher Order Function (HOF)?

Higher order functions are functions which take other function as a parameter or return a function as a value. The function passed as a parameter is called callback.

Callback

A callback is a function which can be passed as parameter to other function. See the example below.

```
// a callback function, the name of the function could be any name
const callback = (n) => {
  return n ** 2
}

// function that takes other function as a callback
function cube(callback, n) {
  return callback(n) * n
}

console.log(cube(callback, 3))
```

Returning function

Higher order functions return function as a value

```
// Higher order function returning an other function
const higherOrder = n => {
```

```
  const doSomething = m => {
    const doWhatEver = t => {
      return 2 * n + 3 * m + t
    }
    return doWhatEver
  }
  return doSomething
}
console.log(higherOrder(2)(3)(10))
```

Functional Programming Methods

🔷 map

- **Definition:** Iterates through array elements and **returns a new modified array**.
- **Callback Parameters:** `element`, `index`, `array`.

✅ Example:

```
const numbers = [1, 2, 3];
const doubled = numbers.map(num => num * 2);
console.log(doubled); // [2, 4, 6]
```

🔷 filter

- **Definition:** Creates a new array with elements that **pass a condition**.

✅ Example:

```
const numbers = [1, 2, 3, 4];
const evens = numbers.filter(num => num % 2 === 0);
console.log(evens); // [2, 4]
```

🔷 reduce

- **Definition:** Reduces array into a **single value**.
- **Callback Parameters:**

- `accumulator` (stores result)
- `current` (current element)
- `initialValue` (optional, but recommended)

- **Important:** If no `initialValue` is given, the first array element is used as the accumulator. If array is empty → **Error**.

✅ Example:

```
const numbers = [1, 2, 3, 4];
const sum = numbers.reduce((acc, curr) => acc + curr, 0);
console.log(sum); // 10
```

🔷 every

- **Definition:** Checks if **all elements** satisfy a condition.
- **Return:** Boolean (`true` / `false`).

✅ Example:

```
const numbers = [2, 4, 6];
console.log(numbers.every(num => num % 2 === 0)); // true
```

🔷 some

- **Definition:** Checks if **at least one element** satisfies a condition.
- **Return:** Boolean.

✅ Example:

```
const numbers = [1, 3, 5, 6];
console.log(numbers.some(num => num % 2 === 0)); // true
```

🔷 find

- **Definition:** Returns the **first element** that matches a condition.

✅ Example:

```
const numbers = [1, 4, 6, 8];
console.log(numbers.find(num => num > 5)); // 6
```

🔷 findIndex

- **Definition:** Returns the **index of the first element** that matches a condition.

✅ Example:

```
const numbers = [1, 4, 6, 8];
console.log(numbers.findIndex(num => num > 5)); // 2
```

🔷 flat

- **Definition:** Flattens a **nested array** into a single array.

✅ Example:

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

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

console.log(arr.flat());
// [1, 2, 3, 4, [5, 6]]   → flattened 1 level

console.log(arr.flat(2));
// [1, 2, 3, 4, 5, 6]   → flattened 2 levels

console.log(arr.flat(Infinity));
// [1, 2, 3, 4, 5, 6]   → completely flattened
```

JavaScript String Methods

```
function textUtilityApp(input) {
console.log(" 🔹 Original Text:", "${input}");

// 1. Lowercase
```

```
console.log("1️⃣ Lowercase:", input.toLowerCase());

// 2. Uppercase
console.log("2️⃣ Uppercase:", input.toUpperCase());

// 3. Trim
console.log("3️⃣ Trimmed:", `"${input.trim()}"`);

// 4. Word Count (split by spaces)
const words = input.trim().split(" ");
console.log("4️⃣ Word Count:", words.length);

// 5. Check if includes word
console.log("5️⃣ Includes 'JavaScript'?:", input.includes("JavaScript"));

// 6. Replace
console.log("6️⃣ Replace 'JavaScript' with 'JS':", input.replace("JavaScript",
"JS"));

// 7. Repeat
console.log("7️⃣ Repeat text 2 times:", input.repeat(2));

// 8. Slice
console.log("8️⃣ Slice(0, 10):", input.slice(0, 10));

// 9. Substring
console.log("9️⃣ Substring(5, 15):", input.substring(5, 15));

// 10. PadStart & PadEnd (like formatting)
let num = "7";
console.log("🔟 PadStart (3 digits):", num.padStart(3, "0")); // "007"
console.log("1️⃣1️⃣ PadEnd (3 digits):", num.padEnd(3, "0"));   // "700"
}

// Test
let myText = "   I love JavaScript programming   ";
textUtilityApp(myText);
```

JavaScript Object Utility Methods

1️⃣ Object.values()

👉 Gets **all the values** of an object in an array.

```
const person = { name: "Sumit", age: 23, country: "India" };

console.log(Object.values(person));
// ["Sumit", 23, "India"]
```

🌍 **Analogy**: Imagine a form 📋 with fields → you only want the **answers**, not the questions.

2️⃣ Object.keys()

👉 Gets **all the keys (property names)** of an object in an array.

```
const person = { name: "Sumit", age: 23, country: "India" };

console.log(Object.keys(person));
// ["name", "age", "country"]
```

🌍 **Analogy**: In the same form 📋, these are the **questions/labels** → "name", "age", "country".

3️⃣ Object.entries()

👉 Gets **key–value pairs** as an array of arrays.

```
const person = { name: "Sumit", age: 23, country: "India" };

console.log(Object.entries(person));
// [["name", "Sumit"], ["age", 23], ["country", "India"]]
```

🌍 **Analogy**: Like making a **table** 🧾 → each row is `[question, answer]`.

4️⃣ Object.fromEntries()

👉 Converts an array of key–value pairs **back into an object**.

```
const entries = [["name", "Sumit"], ["age", 23], ["country", "India"]];

console.log(Object.fromEntries(entries));
// { name: "Sumit", age: 23, country: "India" }
```

🌍 **Analogy**: Take that **table** 🧾 and turn it back into a **form object**.

Copy by Value vs Copy by Reference

Copy by Value (Primitive Types)

👉 When you assign a **primitive value** (like number, string, boolean, null, undefined, symbol, bigint) to another variable → JavaScript **copies the actual value**.

- Changing one does **not affect** the other.

```
let a = 10;
let b = a;   // copy by value

b = 20;

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

Copy by Reference (Objects & Arrays)

👉 For **non-primitive types** (object, array, function), JavaScript stores a **reference (memory address)**.

- Assigning an object/array to another variable → both variables point to the **same memory location**.
- Changing one will also change the other.

```
let obj1 = { name: "Sumit" };
let obj2 = obj1;   // copy by reference

obj2.name = "Kaushal";
```

```
console.log(obj1.name); // "Kaushal" (changed!)
console.log(obj2.name); // "Kaushal"
```

Pass by Value vs Pass by Reference (in Functions)

Pass by Value (Primitives)

👉 When you pass a **primitive value** (number, string, boolean, etc.) into a function, JavaScript **creates a copy of that value**.

- Changes inside the function do **not affect** the original variable.

```
function changeValue(x) {
  x = x + 10;
  console.log("Inside function:", x);
}

let num = 5;
changeValue(num);

console.log("Outside function:", num);
```

Pass by Reference (Objects & Arrays)

👉 When you pass an **object or array** into a function, JavaScript passes a **reference to the memory location**.

- Changes inside the function **will affect** the original object/array.

```
function changeObject(obj) {
  obj.city = "Mumbai";
  console.log("Inside function:", obj);
}

let person = { name: "Sumit", city: "Delhi" };
changeObject(person);

console.log("Outside function:", person);
```