🔹 1. JavaScript Fundamentals

✅ Variables (var, let, const)

Variables are **containers for storing data values**. You can think of a variable like a label attached to some data so you can reuse it in your program.

• var is function-scoped, while let and const are block-scoped.

• Use const by default, let when reassigning.

var name = "John"; // function scoped

let age = 25; // block scoped

const country = "India"; // block scoped and constant

| **Keyword** | **Scope** | **Reassignable** | **Hoisted** | **Use Case** |
| --- | --- | --- | --- | --- |
| var | Function | Yes | Yes | Legacy, avoid in modern JS |
| let | Block | Yes | No | Use when value changes |
| const | Block | No | No | Use when value stays constant |

Data Types

JavaScript is **dynamically typed**, meaning you don’t need to declare the type of variable. The interpreter automatically determines it.

• Primitive: string, number, boolean, null, undefined, symbol, bigint

• Reference: object, array, function

**🔹 Two Categories of Data Types:**

| **Category** | **Examples** |
| --- | --- |
| **Primitive** | string, number, boolean, null, undefined, symbol, bigint |
| **Non-Primitive** | object, array, function |

**🟦 A. Primitive Data Types**

**1. String**

Represents textual data.

let city = "Delhi";

let greeting = 'Hello';

let message = `Welcome to ${city}`; // Template literal

Strings can be declared with ", ', or `.

**2. Number**

Represents integers or floating-point values.

let age = 30;

let price = 199.99;

No difference between integers and floats.

**3. Boolean**

Represents true or false values.

let isLoggedIn = true;

let isAdmin = false;

Used in conditions and logic.

**4. Undefined**

A variable that has been declared but not assigned a value.

let score;

console.log(score); // undefined

**5. Null**

Represents an empty or "no value" intentionally.

let data = null;

Commonly used to reset or clear a value.

**6. Symbol (ES6)**

A unique and immutable value often used as object keys.

let sym = Symbol("id");

**7. BigInt (ES11)**

Used for integers larger than Number.MAX\_SAFE\_INTEGER.

let big = 1234567890123456789012345678901234567890n;

**🟦 B. Non-Primitive (Reference) Data Types**

**1. Object**

Used to store collections of data and more complex entities.

let user = {

name: "Alice",

age: 28

};

React heavily uses objects in state and props.

**2. Array**

A special type of object for ordered collections.

let fruits = ["Apple", "Banana", "Mango"];

Used in React to render lists with .map().

**3. Function**

Also a type of object.

function greet(name) {

return `Hello, ${name}`;

}

Functions are first-class citizens and are crucial in React for defining components and handling events.

**JavaScript is Loosely Typed**

JavaScript allows you to reassign a variable with a value of a different type (unless using const).

let data = 42; // number

data = "forty two"; // string

**🟦 Type Checking: typeof Operator**

You can check the data type of a variable using typeof.

console.log(typeof 42); // "number"

console.log(typeof "hello"); // "string"

console.log(typeof true); // "boolean"

console.log(typeof {}); // "object"

console.log(typeof []); // "object"

console.log(typeof null); // "object" (quirk in JS)

console.log(typeof undefined); // "undefined"

**Best Practices with Variables in Modern JS**

* Use const by default.
* Use let only if you need to reassign.
* Avoid var.

Operators

• Arithmetic: +, -, \*, /

• Comparison: ==, ===, !=, !==, <, >

• Logical: &&, ||, !

if (isLoggedIn && isAdmin) { ... }

🔹 2. Control Structures

Conditionals

if (age > 18) {

  console.log("Adult");

} else {

  console.log("Minor");

}

Switch Statement

switch(day) {

  case "Monday":

    break;

  default:

    break;

}

Loops

• for, while, do...while, for...of, for...in

for (let i = 0; i < 5; i++) {

  console.log(i);

}

🔹 3. Functions (Essential for Components)

Function Declaration

function greet(name) {

  return `Hello, ${name}`;

}

Function Expression

const greet = function(name) {

  return `Hello, ${name}`;

};

Arrow Functions (Very common in React)

const greet = (name) => `Hello, ${name}`;

🔹 4. Arrays and Array Methods (Very useful for JSX rendering)

Common Methods:

• map() → for rendering lists

• filter()

• forEach()

• reduce()

• find()

const numbers = [1, 2, 3];

const doubled = numbers.map(n => n \* 2);  // [2, 4, 6]

🔹 5. Objects

Object creation & access

const user = {

  name: "Alice",

  age: 25

};

console.log(user.name);

Destructuring

const { name, age } = user;

Spread and Rest Operators

const newUser = { ...user, location: "India" };

🔹 6. DOM Manipulation (less used in React, but useful for understanding)

• document.querySelector

• addEventListener

• innerHTML, textContent

React handles the DOM via virtual DOM, so direct DOM manipulation is rare.

🔹 7. Events

Event Handling

function handleClick() {

  console.log("Clicked");

}

<button onClick={handleClick}>Click Me</button>

🔹 8. ES6+ Features (Must Know for React)

Template Literals

const name = "React";

console.log(`Welcome to ${name}`);

Default Parameters

function greet(name = "User") {

  return `Hello ${name}`;

}

Destructuring (Array/Object)

const [first, second] = [1, 2];

const { title } = { title: "React Guide" };

Spread / Rest

const arr1 = [1, 2];

const arr2 = [...arr1, 3]; // [1,2,3]

function sum(...args) {

  return args.reduce((a, b) => a + b);

}

Ternary Operator

const message = isLoggedIn ? "Welcome" : "Login";

🔹 9. Asynchronous JavaScript

setTimeout, setInterval

setTimeout(() => console.log("Hello"), 1000);

Promises

fetch("/api")

  .then(response => response.json())

  .then(data => console.log(data));

async / await

async function getData() {

  const response = await fetch("/api");

  const data = await response.json();

  console.log(data);

}

🔹 10. Modules and Import/Export

Exporting and Importing

// math.js

export const add = (a, b) => a + b;

// app.js

import { add } from './math.js';

React uses modules heavily (every component is a module).

🔹 11. Classes (Used in Class-based Components)

Class Declaration

class Person {

  constructor(name) {

    this.name = name;

  }

  greet() {

    return `Hello ${this.name}`;

  }

}

🔹 12. Error Handling

try / catch

try {

  const res = await fetch("/api");

} catch (error) {

  console.error(error);

}

🔹 13. Closures & Scope (Advanced but useful in React Hooks)

Closure

function outer() {

  let count = 0;

  return function inner() {

    count++;

    return count;

  };

}

const counter = outer();

counter(); // 1

🔹 14. Truthy/Falsy and Short-Circuiting

const name = "";

console.log(name || "Guest"); // Guest

**Array Methods ->**

**1. Array Creation**

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

let arr2 = new Array(3, 6, 9); // [3, 6, 9]

**🔹 2. Adding/Removing Elements**

**push() – Adds to the end**

arr.push(6); // [1, 2, 3, 4, 5, 6]

**pop() – Removes from the end**

arr.pop(); // [1, 2, 3, 4, 5]

**unshift() – Adds to the beginning**

arr.unshift(0); // [0, 1, 2, 3, 4, 5]

**shift() – Removes from the beginning**

arr.shift(); // [1, 2, 3, 4, 5]

**🔹 3. Searching and Testing**

**includes() – Checks if value exists**

arr.includes(3); // true

**indexOf() – Returns index of first match**

arr.indexOf(3); // 2

**lastIndexOf() – Returns last match index**

[1, 2, 3, 2].lastIndexOf(2); // 3

**find() – Returns first element matching condition**

arr.find(x => x > 3); // 4

**findIndex() – Index of first match by condition**

arr.findIndex(x => x > 3); // 3

**some() – Is any element true for condition?**

arr.some(x => x > 4); // true

**every() – Are all elements true for condition?**

arr.every(x => x > 0); // true

**🔹 4. Looping/Iteration**

**forEach() – Executes function for each element**

arr.forEach(el => console.log(el)); // prints 1 2 3 4 5

**🔹 5. Transforming Arrays**

**map() – Returns new array from function**

let doubled = arr.map(x => x \* 2); // [2, 4, 6, 8, 10]

**filter() – New array with elements matching condition**

let evens = arr.filter(x => x % 2 === 0); // [2, 4]

**reduce() – Reduces array to a single value**

let sum = arr.reduce((acc, curr) => acc + curr, 0); // 15

**flat() – Flattens nested arrays**

[1, [2, 3], [4, [5]]].flat(); // [1, 2, 3, 4, [5]]

[1, [2, 3], [4, [5]]].flat(2); // [1, 2, 3, 4, 5]

**flatMap() – Map + Flat (1 level deep)**

[1, 2, 3].flatMap(x => [x, x \* 2]); // [1, 2, 2, 4, 3, 6]

**🔹 6. Sorting and Reversing**

**sort() – Sorts in place**

arr.sort((a, b) => b - a); // [5, 4, 3, 2, 1]

**reverse() – Reverses array**

arr.reverse(); // [1, 2, 3, 4, 5]

**🔹 7. Slicing and Splicing**

**slice(start, end) – Returns portion (non-destructive)**

arr.slice(1, 4); // [2, 3, 4]

**splice(start, deleteCount, ...items) – Changes original array**

arr.splice(2, 1, 99); // removes index 2, inserts 99

console.log(arr); // [1, 2, 99, 4, 5]

**🔹 8. Joining and Splitting**

**join() – Converts to string**

arr.join('-'); // "1-2-99-4-5"

**toString() – Similar to join()**

arr.toString(); // "1,2,99,4,5"

**🔹 9. Length Property**

console.log(arr.length); // 5

**🔹 10. Array.isArray() – Checks if variable is array**

Array.isArray(arr); // true

**🔹 11. fill() – Fill array with static values**

[1, 2, 3].fill(0); // [0, 0, 0]

new Array(5).fill(7); // [7, 7, 7, 7, 7]

**🔹 12. copyWithin() – Copies array part within same array**

[1, 2, 3, 4, 5].copyWithin(0, 3); // [4, 5, 3, 4, 5]

**🔹 13. at() – Returns item at specific index (can use negative)**

[10, 20, 30].at(-1); // 30

**🔹 14. from() – Creates array from iterable**

Array.from('abc'); // ['a', 'b', 'c']

**🔹 15. Array.of() – Creates array from arguments**

Array.of(1, 2, 3); // [1, 2, 3]

**More examples on array methods ->**

**🔹 1. push()**

**Adds** elements to the **end** of an array.

let fruits = ["apple", "banana"];

fruits.push("orange");

console.log(fruits); // ["apple", "banana", "orange"]

**🔹 2. pop()**

**Removes** the **last** element from an array.

let fruits = ["apple", "banana", "orange"];

fruits.pop();

console.log(fruits); // ["apple", "banana"]

**🔹 3. shift()**

**Removes** the **first** element from an array.

let numbers = [1, 2, 3];

numbers.shift();

console.log(numbers); // [2, 3]

**🔹 4. unshift()**

**Adds** elements to the **beginning** of an array.

let numbers = [2, 3];

numbers.unshift(1);

console.log(numbers); // [1, 2, 3]

**🔹 5. concat()**

**Merges** two or more arrays.

let a = [1, 2];

let b = [3, 4];

let result = a.concat(b);

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

**🔹 6. join()**

**Joins** all elements into a string.

let words = ["hello", "world"];

let sentence = words.join(" ");

console.log(sentence); // "hello world"

**🔹 7. slice(start, end)**

**Returns a shallow copy** of a portion of an array.

let arr = [10, 20, 30, 40, 50];

let sliced = arr.slice(1, 4);

console.log(sliced); // [20, 30, 40]

**🔹 8. splice(start, deleteCount, ...items)**

**Changes** the contents of an array by removing or replacing elements.

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

arr.splice(2, 1, 99); // remove 1 item at index 2 and insert 99

console.log(arr); // [1, 2, 99, 4]

**🔹 9. indexOf()**

Returns the **first index** at which a given element is found.

let names = ["Ali", "John", "Ali"];

console.log(names.indexOf("Ali")); // 0

**🔹 10. includes()**

Checks if an array **contains** a value.

let nums = [1, 2, 3];

console.log(nums.includes(2)); // true

**🔹 11. forEach()**

Executes a function **once** for each array element.

let arr = [10, 20, 30];

arr.forEach(item => console.log(item));

// Output: 10, 20, 30

**🔹 12. map()**

Creates a **new array** by applying a function to each element.

let numbers = [1, 2, 3];

let squared = numbers.map(x => x \* x);

console.log(squared); // [1, 4, 9]

**🔹 13. filter()**

Creates a new array with **elements that pass a condition**.

let nums = [5, 12, 8, 130, 44];

let result = nums.filter(n => n > 10);

console.log(result); // [12, 130, 44]

**🔹 14. reduce()**

Applies a function to **reduce** the array to a **single value**.

let nums = [1, 2, 3, 4];

let sum = nums.reduce((acc, val) => acc + val, 0);

console.log(sum); // 10

**🔹 15. find()**

Returns the **first element** that satisfies the condition.

let users = [{id: 1}, {id: 2}];

let user = users.find(u => u.id === 2);

console.log(user); // {id: 2}

**🔹 16. findIndex()**

Returns the **index** of the first element that satisfies the condition.

let ages = [3, 10, 18, 20];

let index = ages.findIndex(age => age > 18);

console.log(index); // 3

**🔹 17. sort()**

**Sorts** the elements of an array.

let nums = [40, 100, 1, 5, 25];

nums.sort((a, b) => a - b);

console.log(nums); // [1, 5, 25, 40, 100]

**🔹 18. reverse()**

**Reverses** the order of elements.

let arr = [1, 2, 3];

arr.reverse();

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

**🔹 19. every()**

Returns true if **all elements** pass the condition.

let nums = [2, 4, 6];

let allEven = nums.every(n => n % 2 === 0);

console.log(allEven); // true

**🔹 20. some()**

Returns true if **at least one** element passes the condition.

let nums = [1, 3, 5, 6];

let hasEven = nums.some(n => n % 2 === 0);

console.log(hasEven); // true

Top of Form

Bottom of Form

**🔶 What is Hoisting?**

**Hoisting** is a JavaScript mechanism where **variable and function declarations** are **moved to the top** of their scope **before code execution**.

This means you can **use variables or functions before declaring them**, but **only declarations are hoisted**, **not initializations**.

**🔶 1. Hoisting with var**

**✅ Example:**

console.log(a); // undefined

var a = 10;

console.log(a); // 10

**🔍 Explanation:**

JavaScript **hoists** the declaration var a; to the top:

var a; // hoisted

console.log(a); // undefined

a = 10;

console.log(a); // 10

* Only the **declaration** is hoisted, not the **value** (10).

**🔶 2. Hoisting with let and const**

**❌ Example:**

console.log(b); // ❌ ReferenceError: Cannot access 'b' before initialization

let b = 20;

**❌ Example:**

console.log(c); // ❌ ReferenceError

const c = 30;

**🔍 Explanation:**

* let and const are **also hoisted**, but they are in a **"Temporal Dead Zone" (TDZ)** from the start of the block until the declaration is encountered.
* Accessing them before the line of declaration gives a **ReferenceError**.

**🔶 3. Hoisting with Functions**

**✅ Function Declaration:**

sayHi(); // "Hello!"

function sayHi() {

console.log("Hello!");

}

**🔍 Explanation:**

Function **declarations** are **hoisted entirely**, so you can call them before they are defined in the code.

**❌ Function Expression:**

sayHello(); // ❌ TypeError: sayHello is not a function

var sayHello = function () {

console.log("Hello");

};

**🔍 Explanation:**

* Only the **variable** sayHello is hoisted (var sayHello;)
* The function **assignment** is **not hoisted**, so at the time of sayHello() it's still undefined.

**OBJECTS ->**

**🔹 1. Object.keys(obj)**

**Returns an array of the keys (property names) of an object.**

**const user = { name: "Ali", age: 25 };**

**console.log(Object.keys(user)); // ["name", "age"]**

**✅ Use in MERN: For looping over object properties (e.g., state or request body in React/Express).**

**🔹 2. Object.values(obj)**

**Returns an array of the values of an object.**

**const user = { name: "Ali", age: 25 };**

**console.log(Object.values(user)); // ["Ali", 25]**

**✅ Use to display values or compare them (e.g., in filtering/searching).**

**🔹 3. Object.entries(obj)**

**Returns an array of key-value pairs.**

**const user = { name: "Ali", age: 25 };**

**console.log(Object.entries(user));**

**// [["name", "Ali"], ["age", 25]]**

**✅ Useful for rendering dynamic form fields or debugging API responses.**

**🔹 4. Object.assign(target, source)**

**Copies properties from one or more source objects to a target object.**

**js**

**CopyEdit**

**const a = { name: "Ali" };**

**const b = { age: 25 };**

**const merged = Object.assign({}, a, b);**

**console.log(merged); // { name: "Ali", age: 25 }**

**✅ Used in React to update state immutably or merge config/data.**

**🔹 5. Object.freeze(obj)**

**Prevents changes to an object (it becomes immutable).**

**const config = { apiURL: "https://api.example.com" };**

**Object.freeze(config);**

**config.apiURL = "https://newapi.com"; // Won’t change**

**console.log(config.apiURL); // "https://api.example.com"**

**✅ Protect sensitive config (API keys, constants) in both backend and frontend.**

**🔹 6. Object.seal(obj)**

**Prevents adding or removing properties, but allows modifying existing ones.**

**const user = { name: "Ali" };**

**Object.seal(user);**

**user.name = "John"; // allowed**

**user.age = 30; // not added**

**console.log(user); // { name: "John" }**

**✅ Useful in backend validation and object safety.**

**🔹 7. Object.hasOwn(obj, prop) or obj.hasOwnProperty(prop)**

**Checks if the object has a specific property.**

**const user = { name: "Ali" };**

**console.log(Object.hasOwn(user, "name")); // true**

**console.log(user.hasOwnProperty("age")); // false**

**✅ Used in backend (Express/Node) when processing dynamic objects from MongoDB or requests.**

**🔹 8. Object.defineProperty(obj, prop, descriptor)**

**Defines or modifies a property with control over writable, enumerable, and configurable.**

**const user = {};**

**Object.defineProperty(user, "id", {**

**value: 101,**

**writable: false**

**});**

**console.log(user.id); // 101**

**user.id = 202;**

**console.log(user.id); // 101 (no change)**

**✅ Use for creating read-only settings or secure object structures.**

**🔹 9. Object.fromEntries()**

**Opposite of Object.entries(): converts key-value pairs array to object.**

**const entries = [["name", "Ali"], ["age", 25]];**

**const obj = Object.fromEntries(entries);**

**console.log(obj); // { name: "Ali", age: 25 }**

**✅ Often used with new FormData() in React when processing forms.**

**🔹 10. JSON.stringify(obj)**

**Converts a JS object to a JSON string.**

**const user = { name: "Ali", age: 25 };**

**console.log(JSON.stringify(user));**

**// '{"name":"Ali","age":25}'**

**✅ Used to send data over APIs or store in localStorage/sessionStorage.**

**🔹 11. JSON.parse(jsonString)**

**Parses a JSON string into a JS object.**

**const json = '{"name":"Ali","age":25}';**

**const obj = JSON.parse(json);**

**console.log(obj.name); // "Ali"**

**✅ Used to handle responses from MongoDB via API in React or Express.**

**🔹 Bonus: Spread Operator { ...obj }**

**Clones or merges objects (commonly used in React and Node).**

**const user = { name: "Ali" };**

**const updatedUser = { ...user, age: 25 };**

**console.log(updatedUser); // { name: "Ali", age: 25 }**

**✅ Used heavily in React state updates and MongoDB operations.**

**✅ Use Case in MERN**

| **Use Case** | **Object Method** |
| --- | --- |
| **Handle form data** | **Object.entries, Object.fromEntries** |
| **Update state immutably (React)** | **Object.assign, Spread {...}** |
| **Merge API data (Node/Express)** | **Object.assign, Object.keys** |
| **Process MongoDB document** | **Object.keys, JSON.stringify, hasOwnProperty** |
| **Protect config/constants** | **Object.freeze, Object.seal** |
| **Validate dynamic input** | **Object.hasOwn, Object.entries** |

**📘 1. Basic Promise Example**

**Goal: Simulate a delayed task using Promise.**

**function fetchData() {**

**return new Promise((resolve, reject) => {**

**setTimeout(() => {**

**const data = { name: "Manish", age: 25 };**

**resolve(data);**

**}, 2000); // Simulates 2 second delay**

**});**

**}**

**fetchData().then(result => {**

**console.log("Data received:", result);**

**});**

**📘 2. Promise with Error Handling using .catch()**

**Goal: Simulate a failed API call.**

**function fetchUser(isError) {**

**return new Promise((resolve, reject) => {**

**setTimeout(() => {**

**if (isError) {**

**reject("Failed to fetch user!");**

**} else {**

**resolve({ user: "Manish", status: "Active" });**

**}**

**}, 1000);**

**});**

**}**

**fetchUser(true)**

**.then(user => console.log("User:", user))**

**.catch(err => console.error("Error:", err));**

**📘 3. Using async/await with try...catch**

**Goal: Make code look synchronous and catch errors.**

**async function getUser() {**

**try {**

**const user = await fetchUser(false); // Change to true to test error**

**console.log("User found:", user);**

**} catch (error) {**

**console.error("Caught Error:", error);**

**}**

**}**

**getUser();**

**📘 4. Chaining Promises**

**Goal: Use .then() to process data in steps.**

**function doubleNumber(num) {**

**return new Promise((resolve) => {**

**setTimeout(() => {**

**resolve(num \* 2);**

**}, 500);**

**});**

**}**

**doubleNumber(5)**

**.then(result => {**

**console.log("First Result:", result); // 10**

**return doubleNumber(result);**

**})**

**.then(finalResult => {**

**console.log("Second Result:", finalResult); // 20**

**});**

**📘 5. Multiple Promises with Promise.all()**

**Goal: Run multiple async tasks in parallel.**

**function delayMsg(msg, time) {**

**return new Promise(resolve => {**

**setTimeout(() => resolve(msg), time);**

**});**

**}**

**Promise.all([**

**delayMsg("First message", 1000),**

**delayMsg("Second message", 2000),**

**delayMsg("Third message", 1500)**

**]).then(messages => {**

**console.log("All messages:", messages);**

**});**

**📘 6. Simulated API Call with fetch and async/await**

**This example works in browsers or Node.js with node-fetch.**

**async function getPosts() {**

**try {**

**const response = await fetch("https://jsonplaceholder.typicode.com/posts");**

**const posts = await response.json();**

**console.log("First Post:", posts[0]);**

**} catch (error) {**

**console.error("API Error:", error);**

**}**

**}**

**getPosts();**

**🧠 Bonus Challenge for Practice:**

**// 1. Create a function that fetches a username with a delay**

**// 2. Add a chance to fail the fetch randomly**

**// 3. Use try-catch to handle success and failure**