Karyvio
LEARN | BUILD | GROW | GET HIRED

JS JavaScript

Complete Notes - Beginner to Advanced

Small Steps Everyday Make You Better ! ☺

1 What is JavaScript ?

Ans →
- JavaScript is a high-level, lightweight, interpreted programming language.
- It is mainly used to make web pages interactive and dynamic.
- It can run in web browsers as well as on servers (using Node.js).
- JavaScript follows the ECMAScript specification.
- It is versatile and used in web development, mobile apps, desktop apps, game development and more.

Example :-

```
console.log("Hello JavaScript!");
```

Output :-

Hello JavaScript!

↳ Prints text in the console

2 Why JavaScript exists ?

Ans →
- Originally, web pages were static (only HTML + CSS).
- There was a need to make web pages interactive (e.g. respond to user actions).
- So JavaScript was created to add behavior to web pages.

Key Purpose :-
- Make web pages dynamic
- Handle user interactions
- Manipulate content on the page
- Communicate with servers (APIs)

3 History of JavaScript

Ans →
- Created by Brendan Eich in 1995 at Netscape.
- Initially called Mocha, then LiveScript, and finally named JavaScript (for marketing reasons).
- Standardized by ECMAScript (ECMA-262).
- Now it is a widely used language with continuous improvements (ES6, ES7 ... ES2023, etc.).

Timeline :-

1995	→	JavaScript created
1997	→	ECMAScript (ES1)
2009	→	ES5
2015	→	ES6 (major update)
2020+	→	Regular new versions

4 Who is Brendan Eich ?

Ans →
- Brendan Eich is the creator of JavaScript.
- He created JavaScript in just 10 days.
- He was working at Netscape Communications Corporation.
- Later, he also co-founded the Mozilla project and worked on the Firefox browser.

Interesting Facts :-
- JavaScript was created in very less time (around 10 days).
- It was not initially planned to be called JavaScript.
- Today, Brendan Eich is a well-known figure in the tech community.

5 JavaScript vs ECMAScript

Ans →
- JavaScript is the implementation of the ECMAScript specification.
- ECMAScript defines the rules, syntax, features and behavior of the language.
- JavaScript engines (like V8, SpiderMonkey, etc.) implement this specification.
- So, all JavaScript code is ECMAScript, but ECMAScript is not only JavaScript (it's a specification).

Example :-

```
// Both are valid JavaScript (which follows ECMAScript)
let x = 10;
const name = "Karyvio";
console.log(x, name);
```

↳ This code follows ECMAScript rules

Output :-

10 Karyvio

6 Where JavaScript runs ?

Ans →
- In web browsers (Chrome, Firefox, Edge, Safari, etc.)
- On servers (using Node.js)
- In mobile apps (React Native, etc.)
- In desktop apps (Electron, etc.)
- In game development (Phaser, Three.js, etc.)
- Almost everywhere !

Environments :-
- Browser → Client-side JavaScript (DOM, BOM)
- Node.js → Server-side JavaScript
- Others → Mobile, Desktop, IoT, Games, etc.

Karyvio
LEARN | BUILD | GROW | GET HIRED

JS JavaScript

Variables, Data Types & Operators

Good Variables Lead to Clean Code

1 What are variables in JavaScript ?

Ans →
- Variables are used to store data in memory.
- In JavaScript, we can declare variables using var, let or const.
- The value stored in a variable can change (depending on the keyword).
- Variables are case-sensitive (name ≠ Name).

Example :-

```
let name = "Karyvio";
const age = 21;
var isStudent = true;
console.log(name, age, isStudent);
// Output :-
Karyvio 21 true
```

→ Variables store different types of data

2 Difference between var, let and const ?

Ans →
- var is function scoped and can be re-declared and updated.
- let is block scoped and can be updated but not re-declared in the same scope.
- const is block scoped and cannot be re-declared or updated (used for constant values).

Keyword	Scope	Re-declare ?	Update ?
var	Function	Yes	Yes
let	Block	No	Yes
const	Block	No	No

Note :- Prefer let and const (avoid var in modern JS).

3 What are the data types in JavaScript ?

Ans →
- JavaScript has two types of data types:

1. Primitive (store single values)
   → string, number, boolean, undefined,
   → null, symbol, bigint
2. Non-Primitive (store multiple values)
   → object, array, function, etc.

Examples :-

```
let name = "Sovit";          // string
let age = 21;                // number
let isActive = true;         // boolean
let x;                       // undefined
let y = null;                // null
let id = Symbol("id");       // symbol
let big = 1234567890123n;    // bigint
let user = { name: "Karyvio"}; // object
let nums = [1, 2, 3];        // array
```

→ Use typeof to check the data type

4 What is the typeof operator ?

Ans →
- The typeof operator is used to check the data type of a value.
- It returns a string representing the type.
- It is useful while debugging and validating data.

Example :-

```
console.log(typeof "Karyvio");      // string
console.log(typeof 100);            // number
console.log(typeof true);           // boolean
console.log(typeof undefined);      // undefined
console.log(typeof null);           // object (special case)
console.log(typeof {});             // object
console.log(typeof []);             // object
console.log(typeof function(){});   // function
```

5 What are operators in JavaScript ?

Ans →
- Operators are used to perform operations on values and variables.
- JavaScript supports different types of operators like arithmetic, comparison, logical, assignment, etc.
- Operators help in manipulating data and making decisions in programs.

Type	Operators	Example	Result
Arithmetic	+ - * / %	5 + 3	8
Assignment	= += -= *=	x += 2	x = x + 2
Comparison	== === != > <	5 > 3	true
Logical	&& || !	true && false	false
Unary	+ - ! typeof	typeof 10	"number"
Ternary	? :	age > 18 ? "Yes" : "No"	"Yes"

6 What is a string and how to use it ?

Ans →
- A string is a sequence of characters used to represent text.
- Strings can be written using single quotes (''), double quotes ("") or backticks (``).
- Backticks allow template literals (easy string interpolation).

Example :-

```
let firstName = "Sovit";
let lastName = 'Lenka';
let fullName = `Hello, I am ${firstName} ${lastName}`;
console.log(fullName);

// Output :-
Hello, I am Sovit Lenka
```

→ Use backticks for template literals

Karyvio
LEARN | BUILD | GROW | GET HIRED

JS Variables & Data Types

Understanding how to store and work with data

Good Fundamentals Make You a Better Developer !

1 What are the different ways to declare variables in JavaScript ?

Ans → • JavaScript provides three ways to declare variables:
- **var** – function scoped (older way, avoid now)
- **let** – block scoped (can be updated)
- **const** – block scoped (cannot be re-assigned)

Example :-

```
var x = 10;     // function scoped
let y = 20;     // block scoped
const z = 30;   // block scoped (constant)
console.log(x, y, z);
// Output :-
10  20  30
```

→ Use let and const in modern JavaScript

2 What is the difference between var, let and const ?

Ans → These keywords differ in scope, re-declaration and re-assignment.

Feature	var	let	const
Scope	Function	Block	Block
Re-declare	Yes	No	No
Re-assign	Yes	Yes	No
Hoisted	Yes (undefined)	Yes (not initialized)	Yes (not initialized)
Use case	Avoid	Use (default)	Use for constants

Example :-

```
var a = 5;
var a = 10;      // allowed
console.log(a);  // 10

let b = 5;
// let b = 10;   // ✗ Error (already declared)
b = 10;          // allowed

const c = 5;
// c = 10;       // ✗ Error (cannot re-assign)
console.log(b, c);  // 10 5
```

3 What are primitive data types in JavaScript ?

Ans → • Primitive data types are the basic data types that store single values. There are 7 primitive types:

- string → "Hello"
- number → 42, 3.14
- boolean → true, false
- undefined → let x;
- null → null
- symbol → Symbol("id")
- bigint → 1234567890123n

Example :-

```
let name = "Karyvio";       // string
let age = 21;               // number
let isActive = true;        // boolean
let job;                    // undefined
let data = null;            // null
let id = Symbol("id");      // symbol
let big = 9876543210123n;   // bigint
console.log(typeof name);   // "string"
```

4 What are non-primitive (reference) data types ?

Ans → • Non-primitive data types can store multiple values and are called reference types.
- object → { }
- array → [ ]
- function → function() {}
- and some special objects like Date, RegExp, Map, Set, etc.

Example :-

```
let nsee = { name: "Sovit", age: 21 };  // object
let nums = [1, 2, 3];                   // array
function greet() { console.log("Hi"); } // function
console.log(typeof user);  // "object"
console.log(typeof nums);  // "object" (arrays are objects)
console.log(typeof greet); // "function"
```

5 What is the typeof operator ?

Ans → • The typeof operator is used to check the data type of a value. It returns a string indicating the type.

Examples :-

```
console.log(typeof 42);          // "number"
console.log(typeof "Hello");     // "string"
console.log(typeof true);        // "boolean"
console.log(typeof undefined);   // "undefined"
console.log(typeof null);        // "object" (known JS bug)
console.log(typeof {});          // "object"
console.log(typeof []);          // "object"
console.log(typeof function(){}); // "function"
```

Important Note :-
- typeof null returns "object" (this is a known bug in JavaScript and has not been fixed for backward compatibility).
- To check if a value is an array, use Array.isArray().

Check Array Example :-

```
let arr = [1, 2, 3];
console.log(Array.isArray(arr)); // true
console.log(typeof arr);         // "object"
```

6 How to convert between data types ?

Ans → JavaScript allows type conversion between different data types using built-in methods.
- String() → convert to string
- Number() → convert to number
- Boolean() → convert to boolean

Example :-

```
let num = "42";
console.log(Number(num));  // 42
console.log(String(100));  // "100"
console.log(Boolean(0));   // false
console.log(Boolean("Hello")); // true
```

→ Type Conversion is useful while working with forms and APIs

Karyvio
LEARN | BUILD | GROW | GET HIRED

JS Functions in JavaScript

Reusable Blocks of Code

Code Once
Use Many Times
That's the Power
of Functions !

1 What is a function in JavaScript ?

Ans →
- A function is a reusable block of code that performs a specific task.
- It helps in organizing code and avoiding repetition.
- A function can take inputs (parameters) and can return an output.
- Functions are first-class citizens in JavaScript (can be assigned to variables, passed as arguments, returned from other functions).

Example :-

```
function greet(name) {
  console.log("Hello " + name + "!");
}
greet("Karyvio");
```

// Output :-
Hello Karyvio!

Key Point :-
- Functions make code modular, readable and reusable.
- Follow the DRY Principle (Don't Repeat Yourself).

2 Types of functions in JavaScript ?

Ans →
- JavaScript has different types of functions :

1. Function Declaration (or Function Statement)
2. Function Expression
3. Arrow Function (ES6)
4. Anonymous Function
5. Immediately Invoked Function Expression (IIFE)
6. Generator Function (Advanced)
7. Async Function (for asynchronous code)

Example :- Function Declaration

```
function add(a, b) {
  return a + b;
}
console.log(add(5, 3));
```

// Output :-
8

Note :-
- Function declaration is hoisted (can be used before declaration).
- Other types (like expressions) are not hoisted.

3 Function Expression

Ans →
- A function expression is a function assigned to a variable. It is not hoisted.
- Useful when you want to control when the function is created.
- Can be named or anonymous.

Example :-

```
const multiply = function(a, b){
  return a * b;
};
console.log(multiply(4, 5));
```

// Output :-
20

Note :-
- Cannot be used before declaration.
- Useful for callbacks and higher-order functions.

4 Arrow Function (ES6)

Ans →
- Arrow functions provide a shorter syntax and do not have their own this.
- Introduced in ES6.
- Best for small, simple functions.
- Inherits this from the surrounding scope.

Example :-

```
const square = (n) => {
  return n * n;
};
console.log(square(6));
```

// Output :-
36

Shorter Syntax :-

```
const square = n => n * n;
```

- Good for small functions.
- Not suitable when you need your own this (e.g. in objects).

5 Function Parameters and Return

Ans →
- Functions can take parameters (inputs) and can return a value using the return keyword.
- A function can have multiple parameters.
- If no return is specified, it returns undefined.
- JavaScript is dynamically typed, so we don't need to specify the type of parameters.

Example :-

```
function add(a, b) {
  return a + b;
}
const result = add(10, 15);
console.log(result);
```

// Output :-
25

Multiple Parameters :-

```
function fullName(first, last) {
  return first + " " + last;
}
console.log(fullName("Sovit", "Lenka"));
```

// Output :-
Sovit Lenka

6 Default Parameters and Rest Parameters

Ans →
- Default parameters allow you to set a default value if no argument is passed.
- Rest parameters (. . .) allow a function to accept any number of arguments as an array.

Default Parameter :-

```
function greet(name = "Guest") {
    console.log("Hello " + name + "!");
}
greet();              // Hello Guest
greet("Karyvio");     // Hello Karyvio
```

Rest Parameter :-

```
function sum(...nums) {
  return nums.reduce((a, b) => a + b, 0);
}
console.log(sum(1, 2, 3, 4));    // 10
```

When to use ?
- Default parameters → optional values.
- Rest parameters → when you don't know how many arguments will be passed.

Karyvio
LEARN | BUILD | GROW | GET HIRED

JS Scope & Hoisting

Understanding how JavaScript manages variables

Understand Scope & Hoisting Write Cleaner Code ! ☺

1 What is Scope in JavaScript ?

Ans →
- Scope determines where a variable can be accessed in your code.
- It controls the visibility and lifetime of variables.
- JavaScript has 3 types of scope:
  1. Global Scope
  2. Function Scope
  3. Block Scope (introduced in ES6)

Example :-

```
let globalVar = "I am global";
function test() {
    let localVar = "I am local";
    console.log(globalVar);  // accessible
    console.log(localVar);   // accessible
}
test();
console.log(globalVar);  // accessible
// console.log(localVar);  // Error: not defined
```

localVar is not accessible outside the function

2 Global Scope

Ans →
- Variables declared outside any function or block are in the global scope.
- They can be accessed from anywhere in the code.
- In browsers, global variables become properties of the window object.

Example :-

```
var x = 10;    // global scope
function show() {
  console.log(x);   // accessible
}
show();  // 10
console.log(window.x);  // 10 (in browser)
```

Note :-
- Avoid creating too many global variables.
- They can cause conflicts in large projects.

3 Function Scope

Ans →
- Variables declared with var, let or const inside a function are in function scope.
- They can only be accessed inside that function.
- Each function creates its own scope.

Example :-

```
function demo() {
   var a = 5;
   let b = 10;
   const c = 15;
   console.log(a, b, c);  // 5 10 15
}
demo();
// console.log(a);  // Error: a is not defined
```

Key Point :-
- Variables in function scope are not accessible outside the function.

4 Block Scope (ES6)

Ans →
- Variables declared with let and const inside a block { } are in block scope.
- They can only be accessed inside that block.
- var does not have block scope (it is function scoped).

Example :-

```
if (true) {
   let x = 100;
   const y = 200;
   console.log(x, y);  // 100 200
}
// console.log(x);  // Error: x is not defined
```

Note :-
- Use let and const to avoid accidental global variables.
- Block scope helps in writing safer and cleaner code.

5 What is Hoisting ?

Ans →
- Hoisting is a JavaScript behavior where variable and function declarations are moved to the top of their scope during execution.
- It means you can use a variable or function before declaring it (but with some rules).

Example :-

```
console.log(a);   // undefined
var a = 10;

console.log(b);   // Error (cannot access)
let b = 20;
```

Important :-
- Only declarations are hoisted, not initializations.

6 Hoisting Behavior with var, let, const and Functions

Ans →

Type	Hoisted ?	Initial Value	Can use before declaration ?	Scope
var	Yes	undefined	Yes (but value is undefined)	Function scope
let	Yes	Not initialized (TDZ)	No (ReferenceError)	Block scope
const	Yes	Not initialized (TDZ)	No (ReferenceError)	Block scope
Function Declaration	Yes	Function body	Yes	Function scope
Function Expression	No (acts like let)	Not initialized (TDZ)	No	Block scope

TDZ :-
- Temporal Dead Zone (TDZ) is the time between the start of the block and the variable declaration.

Karyvio
LEARN | BUILD | GROW | GET HIRED

JS Type Conversion & Coercion

Understanding how JavaScript converts values between types

Know How JavaScript Thinks !

1 What is Type Conversion ?

Ans →
- Type conversion means converting a value from one data type to another.
- JavaScript can convert types automatically (implicit) or manually (explicit).
- It is useful when working with user input, operations and comparisons.

Example :-

```
let num = "100";
console.log(typeof num);  // string
let n = Number(num);
console.log(typeof n);    // number
```

Key Point :-
- Same value can exist in different types.
- Be careful with automatic conversion.

2 Types of Type Conversion

Ans → There are two types :

1. Implicit Conversion (Type Coercion)
   → Done automatically by JavaScript.
   → Happens behind the scenes.
2. Explicit Conversion (Type Casting)
   → Done manually by the developer.
   → Uses built-in methods.

Example :- Implicit Conversion

```
console.log("5" + 2);   // "52"  (string)
console.log("5" - 2);   // 3     (number)
console.log(true + 1);  // 2     (number)
console.log(false + 1); // 1     (number)
```

Example :- Explicit Conversion

```
console.log(Number("10") + 5); // 15
console.log(String(100));      // "100"
console.log(Boolean(0));       // false
```

Remember :-
- + with a string performs concatenation.
- - * / perform numeric conversion.
- Always check the type using typeof.

3 Implicit Conversion Rules

Ans → JavaScript automatically converts types in many situations.
- When using arithmetic operators (+, -, *, /).
- When using comparison operators (==, !=, >, <).
- When using logical operators (&&, ||, !).
- When a value is used in a context that expects a different type.

Example :-

```
console.log("10" + 5);      // "105"
console.log("10" - 5);      // 5
console.log("2" * "3");     // 6
console.log(true == 1);     // true
console.log(false == 0);    // true
console.log(null == undefined); // true
```

Be Careful !
- Implicit conversion can lead to unexpected results.
- Always use === for safe comparisons.

4 Explicit Conversion Methods

Ans → JavaScript provides several built-in methods to convert types.

Method	Description	Example	Result
String()	Converts value to string	String(123)	"123"
Number()	Converts value to number	Number("123")	123
Boolean()	Converts value to boolean	Boolean(1)	true
parseInt()	Parses and converts to integer	parseInt("10.5")	10
parseFloat()	Parses and converts to float	parseFloat("10.5")	10.5

More Examples :-

```
console.log(String(true));      // "true"
console.log(Number(false));     // 0
console.log(Boolean(""));       // false
console.log(Boolean("Hello")); // true
console.log(parseInt("100px")); // 100
console.log(parseFloat("3.14abc")); // 3.14
```

Note :-
- parseInt() and parseFloat() ignore non-numeric characters at the end.
- Boolean(0), Boolean(""), Boolean(null), Boolean(undefined) all return false.

5 Truthy and Falsy Values

Ans →
- In JavaScript, some values are treated as false when converted to boolean. All other values are true.

Falsy Values (convert to false)
- false
- 0
- -0
- 0n (BigInt zero)
- "" (empty string)
- null
- undefined
- NaN

Truthy Values (convert to true)
- true
- 1, -1, 123, ... (any non-zero number)
- "hello" (non-empty string)
- [] (empty array)
- {} (empty object)
- function() {} (any function)
- new Date()
- "0" (string with zero)

Examples :-

```
console.log(Boolean(0));   // false
console.log(Boolean(""));  // false
console.log(Boolean([]));  // true
console.log(Boolean({}));  // true
console.log(Boolean("0")); // true
console.log(Boolean(NaN)); // false
```

6 Real-world Use Cases

Ans →
- Converting user input (strings from forms) to numbers.
- Checking if a value exists (truthy/falsy) before using it.
- Parsing data from APIs (JSON).
- Handling conditions in if statements.

Example :-

```
let age = prompt("Enter your age:");
age = Number(age);
if (!isNaN(age) && age > 0) {
    console.log("Valid age:", age);
} else {
    console.log("Invalid age!");
}
```

Pro Tip :-
- Always validate and convert user input to the correct type before using it in your code.

Karyvio
LEARN | BUILD | GROW | GET HIRED

JS Strings in JavaScript

Working with text, methods and real-world examples

A String is a sequence of characters used to represent text ! :)

1 What is a String ?

Ans →
- A string is a sequence of characters (letters, numbers, symbols, spaces, etc.) used to represent text in JavaScript.
- Strings are written inside single quotes (' '), double quotes (" ") or backticks (` `).
- Strings are immutable, which means once created, they cannot be changed. We can only create a new string based on the old one.

Example :-
```
let name = "Karyvio";
let msg = 'Learn JavaScript';
let tagline = `Learn | Build | Grow`;
console.log(name);
console.log(msg);
console.log(tagline);
// Output :-
Karyvio
Learn JavaScript
Learn | Build | Grow
```

Key Points :-
- Strings can be written using ' ', " " or ` `.
- Use backticks for multi-line strings and template literals.
- Strings are immutable.

2 Creating Strings

Ans →
- You can create strings using single quotes, double quotes or backticks.
- Backticks allow multi-line strings and variable interpolation (template literals).

Example :-
```
let a = 'Hello';
let b = "World";          // Multi-line string
let c = `Hello
        Karyvio`;         // Output :-
console.log(c);           Hello
                          Karyvio
```

Tip :-
- Use backticks (` `) when you need multi-line strings or want to embed variables inside a string. :)

3 String Length

Ans →
- The length property returns the number of characters in a string (including spaces).
- It is useful for validation (e.g. password length, input limits).

Example :-
```
let str = "Karyvio";
console.log(str.length);    // 7
console.log("Hello World".length); // 11
```

Note :-
- Spaces are also counted in length.

4 Accessing Characters

Ans →
- You can access individual characters using indexing (starts from 0).
- You can also use charAt(index) method.

Example :-
```
let str = "Karyvio";
console.log(str[0]);        // K
console.log(str[3]);        // y
console.log(str.charAt(5)); // i
```

Remember :-
- Index always starts from 0 in JavaScript.
- If the index is out of range, it returns undefined.

5 String Methods (Commonly Used)

Ans →

Method	Description	Example	Result
toUpperCase()	Converts to uppercase	"karyvio".toUpperCase()	"KARYVIO"
toLowerCase()	Converts to lowercase	"KARYVIO".toLowerCase()	"karyvio"
trim()	Removes spaces from both ends	" hi ".trim()	"hi"
slice(start, end)	Extracts part of string	"Karyvio".slice(0, 4)	"Kary"
substring(start, end)	Similar to slice (no negative index)	"Karyvio".substring(0, 4)	"Kary"
replace()	Replaces part of string	"Hello JS".replace("JS","World")	"Hello World"
includes()	Checks if string contains value	"Hello JS".includes("JS")	true
indexOf()	Returns index of first occurrence	"Hello".indexOf("l")	2

More Examples :-
```
let str = "  Learn JS  ";
console.log(str.trim());    // "Learn JS"
console.log(str.toUpperCase());
console.log(str.toLowerCase());
console.log(str.includes("JS"));
console.log(str.replace("Learn","Master"));
// Output :-
Learn JS
  LEARN JS
  learn js
  true
  Master JS
```

6 Template Literals (String Interpolation)

Ans →
- Template literals (backticks) allow you to embed variables and expressions inside strings using ${ }.
- They make the code more readable and easier to write.

Example :-
```
let name = "Sovit";
let age = 21;
let msg = `My name is ${name} and I am ${age} years old.`;
console.log(msg);
// Output :-
My name is Sovit and I am 21 years old.
```

Pro Tip :-
- Use template literals instead of string concatenation for better readability.
- You can also use expressions inside ${ }. (e.g. ${2 + 3}).

Karyvio
LEARN | BUILD | GROW | GET HIRED

JS String Methods in JavaScript

Useful built-in methods to work with strings effectively

Small String Methods Big Power in Real Projects !

1 What are String Methods ?

Ans →
- String methods are built-in functions in JavaScript that allow you to perform various operations on strings.
- They do not modify the original string (strings are immutable). Instead, they return a new string.
- We call them using dot notation on a string.

Example :-
```
let str = "Karyvio";
let newStr = str.toUpperCase();
console.log(newStr);
// KARYVIO
```

Key Point :-
- String methods return a new string.
- Original string remains unchanged.
- Use dot (.) notation to call a method.

2 Case Conversion Methods

Ans

Method	Description	Example	Result
toUpperCase()	Converts to uppercase	"Karyvio".toUpperCase()	KARYVIO
toLowerCase()	Converts to lowercase	"KARYVIO".toLowerCase()	karyvio"
toLocaleUpperCase()	Uppercase (locale aware)	"i".toLocaleUpperCase('tr')	İ
toLocaleLowerCase()	Lowercase (locale aware)	"I".toLocaleLowerCase('tr')	ı

Example :-
```
let str = "Learn Js";
console.log(str.toUpperCase());
console.log(str.toLowerCase());
console.log(str.toLocaleUpperCase());
// Output :-
  LEARN JS
  learn js
  LEARN JS
```

3 Finding and Checking Methods

Ans

Method	Description	Example	Result
indexOf()	Returns first index	"Karyvio".indexOf("y")	3
lastIndexOf()	Returns last index	"Karyvio".lastIndexOf("o")	6
includes()	Checks if substring exists	"Karyvio".includes("vio")	true
startsWith()	Checks if starts with	"Karyvio".startsWith("Kar")	true
endsWith()	Checks if ends with	"Karyvio".endsWith("vio")	true
search()	Searches using regex	"Karyvio".search(/v/)	4
match()	Returns matches (array)	"Karyvio".match(/a/g)	['a']

Example :-
```
let str = "Learn with Karyvio";
console.log(str.includes("Karyvio"));
console.log(str.startsWith("Learn"));
console.log(str.endsWith("vio"));
console.log(str.indexOf("with"));
// Output :-
  true
  true
  true
  6
```

4 Extracting Parts of a String

Ans

Method	Description	Example	Result
slice(start, end)	Extracts part (end not included)	"Karyvio".slice(0, 4)	Kary
substring(start, end)	Similar to slice (no negative index)	"Karyvio".substring(0, 4)	Kary
substr(start, length)	Extracts with length (deprecated)	"Karyvio".substr(0, 3)	Kar

Note :-
- slice() supports negative index.
- substring() does not support negative index (treats negatives as 0).
- substr() is deprecated (avoid using in new projects).

5 Modifying Strings

Ans

Method	Description	Example	Result
concat()	Joins two or more strings	"Hello ".concat("World")	Hello World
trim()	Removes spaces from both ends	" Karyvio ".trim()	Karyvio
trimStart()	Removes spaces from start	" Hello".trimStart()	Hello
trimEnd()	Removes spaces from end	"Hello ".trimEnd()	Hello
padStart()	Pads string at start	"5".padStart(3, "0")	005
padEnd()	Pads string at end	"5".padEnd(3, "0")	500
repeat()	Repeats a string	"Hi ".repeat(3)	Hi Hi Hi
replace()	Replaces first match	"Hello JS".replace("JS","World")	Hello World
replaceAll()	Replaces all matches (ES2021)	"a-a-a".replaceAll("-", " ")	a a a

Example :-
```
let msg = "  Keep Learning ";
console.log(msg.trim());
console.log("JS".repeat(4));
console.log("a-a-a".replaceAll("-", " "));
// Output :-
  Keep Learning
  JSJSJSJS
  a a a
```

Tip :-
- Use trim() while handling user input (forms).
- Use replaceAll() if you want to replace all occurrences.
- Use padStart() to format numbers (e.g. 005).

6 Splitting and Joining

Ans

Method	Description	Example	Result
split()	Splits string into array	"a,b,c".split(",")	['a', 'b', 'c']
join()	Joins array into string	["a","b","c"].join("-")	a-b-c

Example :-
```
let str = "HTML,CSS,JS";
let arr = str.split(",");
console.log(arr);
console.log(arr.join(" | "));
```

Output :-
```
['HTML', 'CSS', 'JS']
HTML | CSS | JS
```

Karyvio
LEARN | BUILD | GROW | GET HIRED

JS Numbers and Math in JavaScript

Working with numbers, useful math methods and real-world examples

Numbers may look simple but they have many useful tricks !

1 What is a Number ?

Ans →
- In JavaScript, a number is a data type used to represent both integer and floating-point values.
- JavaScript uses the IEEE 754 standard for representing numbers (64-bit floating point).
- Numbers can be positive, negative, whole numbers, decimals, and special values like Infinity, -Infinity and NaN.
- There is no separate integer type in JavaScript.

Example :-

```
let a = 10;       // integer
let b = 3.14;     // decimal
let c = -5;       // negative
let d = 0;        // zero
let e = 1e3;      // 1000 (exponential)
let f = -2.5e-3; // -0.0025
console.log(typeof a); // number
console.log(typeof b); // number
```

Key Point :-
- Numbers are stored as 64-bit floating point values.
- No separate integer type.
- You can perform arithmetic, compare and use many built-in Math methods.

2 Special Number Values

Ans →
- JavaScript has some special numeric values that are useful in certain situations.
- Infinity : Represents a value greater than any other number.
- -Infinity : Represents a value smaller than any other number.
- NaN : Stands for "Not a Number". It is returned when a mathematical operation fails or results in an invalid number.

Example :-

```
console.log(1 / 0);     // Infinity
console.log(-1 / 0);    // -Infinity
console.log("abc" / 2); // NaN
console.log(typeof NaN); // number
console.log(Number("hello")); // NaN
console.log(isNaN("hello")); // true
```

Note :-
- typeof NaN is "number" (this is a known quirk).
- Use isNaN() to check for NaN.
- Infinity is useful in comparisons and limits.

3 Number Methods

Ans → JavaScript provides several built-in methods to work with numbers.

Method	Description	Example	Result
toString()	Converts number to string	(123).toString()	"123"
toFixed(n)	Formats number with n decimal places	(3.14159).toFixed(2)	"3.14"
toPrecision(n)	Formats number with n significant digits	(3.14159).toPrecision(3)	"3.14"
parseInt()	Parses string and returns integer	parseInt("42.8")	42
parseFloat()	Parses string and returns float	parseFloat("42.8")	42.8
isNaN()	Checks if value is NaN	isNaN("hello")	true
isFinite()	Checks if value is a finite number	isFinite(100)	true

Example :-

```
let num = 12.3456;
console.log(num.toFixed(2));  // 12.35
console.log(num.toPrecision(4)); // 12.35
console.log(parseInt("100px")); // 100
console.log(parseFloat("5.6abc")); // 5.6
console.log(isNaN(123));  // false
console.log(isFinite(1/0));  // false
```

Tip :-
- Use toFixed() for currency values (e.g. 99.99).
- parseInt() ignores non-numeric characters after the number.
- parseFloat() works with decimals.

4 Math Object

Ans → JavaScript has a built-in Math object that provides many useful methods and constants.

Property / Method	Description	Example	Result
Math.PI	Value of π	Math.PI	3.1415926535
Math.E	Value of e (Euler's number)	Math.E	2.718281828
Math.round()	Rounds to nearest integer	Math.round(4.6)	5
Math.floor()	Rounds down (smallest integer)	Math.floor(4.9)	4
Math.ceil()	Rounds up (largest integer)	Math.ceil(4.1)	5
Math.abs()	Returns absolute value	Math.abs(-10)	10
Math.min()	Returns smallest number	Math.min(2, 5, 1)	1
Math.max()	Returns largest number	Math.max(2, 5, 10)	10
Math.random()	Returns a random number (0 to 1)	Math.random()	0.7321 (example)
Math.pow()	Returns base raised to exponent	Math.pow(2, 3)	8
Math.sqrt()	Returns square root	Math.sqrt(16)	4

Example :-

```
console.log(Math.PI);
console.log(Math.round(4.7));
console.log(Math.floor(4.7));
console.log(Math.ceil(4.1));
console.log(Math.abs(-25));
console.log(Math.max(10, 20, 5));
console.log(Math.random());
console.log(Math.pow(2, 5));
console.log(Math.sqrt(49));
// Output :-
3.1415926535
5
4
5
25
20
0.3819 (random)
32
7
```

Common Use Cases :-
- Rounding values (e.g. marks, prices).
- Generating random numbers (e.g. OTP, games).
- Finding min/max values.
- Working with geometric calculations (e.g. area, distance).

Pro Tip :-
- Math.random() * 10 gives a random number between 0 and 10.
- Use Math.floor() for a whole number.

JS Conditions and Ternary Operator

Make decisions in your code using conditions

Good Developers write conditions, Great Developers handle all cases !

1 Why do we need Conditions ?

Ans →
- In real life, we make decisions based on conditions (e.g. if it's raining, take an umbrella).
- Similarly, in JavaScript, we use conditions to execute code only when a certain condition is true.
- Conditions help make our programs dynamic and intelligent.
- We use comparison operators to compare values, which returns true or false.

Example :-

```
let age = 18;
if (age >= 18) {
   console.log("You are an adult");
} else {
   console.log("You are a minor");
}
// Output :-
You are an adult
```

Real-life Analogy :-

- If it's hot, turn on the fan.
- If balance is low, show a warning.
- If user is logged in, show dashboard.
- If item is in stock, allow purchase.

2 Comparison Operators

Ans → Used to compare two values. They return a boolean value (true or false).

Operator	Meaning	Example	Result
==	Equal to (value)	5 == '5"	true
===	Equal to (value & type)	5 === '5"	false
!=	Not equal to (value)	5 != '5"	false
!==	Not equal to (value & type)	5 !== '5"	true
>	Greater than	10 > 5	true
<	Less than	3 < 8	true
>=	Greater than or equal to	5 >= 5	true
<=	Less than or equal to	4 <= 2	false

Example :-

```
console.log(10 > 5);      // true
console.log(10 < 5);      // false
console.log(5 == "5");    // true
console.log(5 === "5");   // false
console.log(7 != 8);      // true
console.log(7 !== "7");   // true
```

Note :-

- Use === instead of == to avoid unexpected results.
- Comparison operators always return a boolean value (true/false).

3 if Statement

Ans → The if statement executes a block of code only if the condition is true.

Example :-

```
let age = 20;
if (age >= 18) {
   console.log("You can vote");
}
// Output :-
You can vote
```

4 if...else Statement

Ans → The if...else statement executes one block if the condition is true and another block if it is false.

Example :-

```
let age = 16;
if (age >= 18) {
   console.log("Adult");
} else {
   console.log("Minor");
}
// Output :-
Minor
```

5 if...else if...else

Ans → Used when you have multiple conditions to check.

Example :-

```
let score = 85;
if (score >= 90) {
   console.log("Grade A");
} else if (score >= 75) {
   console.log("Grade B");
} else if (score >= 50) {
   console.log("Grade C");
} else {
} console.log("Fail");
// Output :-
Grade B
```

6 Ternary Operator (Conditional Operator)

Ans → A shorter way to write an if...else statement.

Syntax :-

```
condition ? expressionIfTrue : expressionIfFalse
```

Example :-

```
let age = 18;
let result = age >= 18 ? "Adult" : "Minor";
console.log(result);
// Output :-
Adult
```

Note :-

- It's a compact way to write simple conditions.
- Use ternary only for small conditions. For complex logic, use if...else.

7 Nested Ternary Operator

Ans → You can also use ternary operator inside another ternary, but it can be hard to read.

Example :-

```
let age = 20;
let result = age >= 18 ? (age >= 60 ? "Senior" : "Adult") : "Minor";
console.log(result);
// Output :-
Adult
```

Tip :-

- Avoid too many nested ternary operators.
- Keep your conditions clean and readable.
- Use parentheses to make the logic clear.

Karyvio
LEARN | BUILD | GROW | GET HIRED

JS Loops in JavaScript

Repeat a block of code multiple times

Loops save time, reduce repetition and make code powerful !

1 Why do we need Loops ?

Ans →
- When we need to repeat a block of code multiple times.
- Helps in automation and reduces code duplication.
- Useful for working with arrays, strings, objects, etc.
- Very commonly used in real-world applications (e.g. showing a list of items, processing data).

Real-life Examples :-
- Show a list of students.
- Print numbers from 1 to 100.
- Process items in a cart (e-cmmerce).
- Read data from an API and display it on UI.
- Perform calculations (e.g. sum of numbers).

2 for Loop

Ans → The most commonly used loop. It runs a block of code a fixed number of times.

Syntax :-
```
for (initialization; condition; update) {
    // code to be executed
}
```

Example :-
```
for (let i = 1; i <= 5; i++) {
    console.log("Hello " + i);
}
// Output :-
Hello 1
Hello 2
Hello 3
Hello 4
Hello 5
```

Key Points :-
- Executes from initialization to condition.
- Condition is checked before each iteration.
- Update runs after each iteration.
- If condition becomes false, the loop stops.

3 while Loop

Ans → Executes a block of code as long as the condition is true.

Syntax :-
```
let i = 1;
while (condition) {
    // code to be executed
    // update (important)
}
```

Example :-
```
let i = 1;
while (i <= 5) {
    console.log("Value: " + i);
    i++;    // update
}
// Output :-
Value: 1
Value: 2
Value: 3
Value: 4
Value: 5
```

Things to Remember :-
- If you forget to update the variable, it will cause an infinite loop.
- Condition is checked before executing the block.
- Useful when the number of iterations is not known in advance.

4 do...while Loop

Ans → Similar to while loop, but it always executes the block at least once, even if the condition is false.

Syntax :-
```
let i = 1;
do {
    // code to be executed
    // update
} while (condition);
```

Example :-
```
let i = 1;
do {
    console.log("Count: " + i);
    i++;
} while (i <= 3);

// Output :-
Count: 1
Count: 2
Count: 3
```

Key Points :-
- Condition is checked after executing the block.
- So, the block runs at least once.
- Useful when you want to show a menu or take input from user at least once.

5 for...in Loop

Ans → Used to iterate over the keys (property names) of an object.

Example :-
```
const person = { name: "Sovit", age: 21};
for (let key in person) {
    console.log(key + " : " + person[key]);
}
// Output :-
name : Sovit
age : 21
```

6 for...of Loop

Ans → Used to iterate over the values of an iterable (like arrays, strings, etc.).

Example :-
```
const colors = ["Red", "Green", "Blue"];
for (let color of colors) {
    console.log(color);
}
// Output :-
Red
Green
Blue
```

for...in vs for...of

for...in	for...of
Iterates over keys (property names)	Iterates over values
Used with objects (and arrays)	Used with arrays, strings, maps, sets, etc.
Returns index/key	Returns value
Not recommended for arrays	Preferred for arrays

Pro Tip :-
- Use for loop when number of iterations is known.
- Use while loop when condition is based.
- Use do...while when you want at least one execution.
- Use for...of for arrays and for...in for objects.
- Avoid infinite loops (always update the variable).

Karyvio
LEARN | BUILD | GROW | GET HIRED

JS Arrays in JavaScript

Store, manage and work with multiple values in a single variable

Arrays make it easy to store multiple values and work with them efficiently !

1 What is an Array ?

Ans →
- An array is a special variable that can hold multiple values in a single variable.
- Values can be of any data type (numbers, strings, booleans, objects, etc).
- Arrays are ordered (indexed) and start from 0.
- Arrays are mutable (we can change them).

Example :-

```
let fruits = ["Apple", "Banana", "Mango"];
console.log(fruits);
console.log(fruits[0]);
console.log(fruits[2]);
// Output :-
[ 'Apple', 'Banana', 'Mango' ]
Apple
Mango
```

Key Points :-

- Arrays are written using square brackets [ ]
- Each value is separated by a comma.
- Index starts from 0.
- Arrays can store different data types.
- Arrays are objects in JavaScript (typeof returns "object").

2 Creating Arrays

Ans → We can create arrays in different ways.

Examples :-

```
let arr1 = [1, 2, 3];                  // array literal
let arr2 = new Array(4, 5, 6);         // using constructor
let arr3 = ["Sovit", 21, true, null]; // mixed types
let arr4 = [];                         // empty array
console.log(arr1, arr2, arr3, arr4);
// Output :-
[ 1, 2, 3 ] [ 4, 5, 6 ] [ 'Sovit", 21, true, null ] []
```

Note :-

- Using [ ] (array literal) is the most common and preferred way.
- new Array() is less commonly used.
- An empty array [ ] can be filled later.

Different Data Types in Array :-

```
let mixed = [
  1,
  "Hello",
  true,
  { name: "Karyvio" },
  [10, 20]
];
console.log(mixed[3].name); // Karyvio
console.log(mixed[4][1]);   // 20
```

3 Accessing Array Elements

Ans → We can access elements using their index number.

Example :-

```
let arr1 = ["BMW", "Audi", "Tesla"];
console.log(cars[0]);   // BMW
console.log(cars[1]);   // Audi
console.log(cars[2]);   // Tesla
console.log(cars[3]);   // undefined (out of bounds)
```

Important :-

- Index starts from 0.
- If index is out of range, it returns undefined.
- We can also access last element using arr[arr.length - 1].

Accessing Last Element :-

```
let cars = ["BMW", "Audi", "Tesla"];
console.log(cars[cars.length - 1]);
// Output :-
Tesla
```

4 Array Length

Ans → The length property returns the number of elements in an array.

Example :-

```
let nums = [10, 20, 30, 40, 50];
console.log(nums.length); // 5
nums.length = 3;          // can also modify length
console.log(nums);        // [10, 20, 30]
```

Note :-

- length is a property, not a method.
- We can also change the length to remove or add elements.
- If length is increased, new empty slots are added (undefined).

Example (Increasing Length) :-

```
let a = [1, 2, 3];
a.length = 5;
console.log(a);
// Output :-
[ 1, 2, 3, <2 empty items> ]
```

5 Modifying Elements

Ans → We can change values using index.

Example :-

```
let colors = ["Red", "Green", "Blue"];
colors[1] = "Yellow";
console.log(colors);
// Output :-
[ 'Red', 'Yellow', 'Blue' ]
```

6 Adding Elements

Ans → We can add elements using index or array methods (we will learn more in next page).

Example :-

```
let arr = [1, 2, 3];
arr[3] = 4;
console.log(arr);
// Output :-
[ 1, 2, 3, 4 ]
```

7 Removing Elements

Ans → We can remove elements by changing length or using methods (covered in next page).

Example :-

```
let arr = [1, 2, 3, 4];
arr.length = 2;
console.log(arr);
// Output :-
[ 1, 2 ]
```

JS Array Methods in JavaScript

Powerful built-in methods to work with arrays easily

Arrays are more powerful with methods ! Use them and write cleaner code

1 push() & pop()

- push() adds element at the end.
- pop() removes the last element.

Example :-

```
let arr = [1, 2, 3];
arr.push(4);
console.log(arr);    // [1, 2, 3, 4]
arr.pop();
console.log(arr);    // [1, 2, 3]
```

Key Point :-

- push() returns the new length of the array.
- pop() returns the removed element.

Output :-

```
4  (length after push)
[1, 2, 3, 4]
4  (removed element)
[1, 2, 3]
```

2 unshift() & shift()

- unshift() adds element at the beginning.
- shift() removes the first element.

Example :-

```
let arr = [2, 3, 4];
arr.unshift(1);
console.log(arr);    // [1, 2, 3, 4]
arr.shift();
console.log(arr);    // [2, 3, 4]
```

Key Point :-

- unshift() is slower than push() for large arrays.
- shift() removes and returns the first element.

Output :-

```
4  (length after unshift)
[1, 2, 3, 4]
1  (removed element)
[2, 3, 4]
```

3 map()

- Creates a new array by applying a function to each element.
- Does not change the original array.

Example :-

```
let arr = [1, 2, 3];
let doubled = arr.map(x => x * 2);
console.log(doubled);    // [2, 4, 6]
console.log(arr);        // [1, 2, 3]
```

Key Point :-

- Returns a new array.
- Very useful for transformation.

4 filter()

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

Example :-

```
let arr = [1, 2, 3, 4, 5];
let even = arr.filter(x => x % 2 === 0);
console.log(even);        // [2, 4]
```

5 reduce()

- Reduces array to a single value.
- Useful for sum, product, count, etc.

Example :-

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

Key Point :-

- acc = accumulator
- curr = current value
- Second argument (0) is the initial value.

6 forEach()

- Executes a function for each element.
- Does not return a new array (it returns undefined).

```
let arr = ["A", "B", "C"];
arr.forEach((item, index) => {
  console.log(index + " : " + item);
});
```

7 find() & findIndex()

- find() returns the first element that matches the condition.
- findIndex() returns the index of the first matching element.

Example :-

```
let arr = [10, 20, 30, 40];
let num = arr.find(x => x > 25);
console.log(num);        // 30
let index = arr.findIndex(x => x > 25);
console.log(index);      // 2
```

8 includes()

- Checks if an element exists in the array.
- Returns true or false.

Example :-

```
let arr = ["JS", "React", "Node"];
console.log(arr.includes("React")); // true
console.log(arr.includes("Java")); // false
```

9 slice()

- Returns a shallow copy of a portion of the array.
- Does not modify the original array.

Example :-

```
let arr = [1, 2, 3, 4, 5];
let part = arr.slice(1, 4);
console.log(part);    // [2, 3, 4]
console.log(arr);     // [1, 2, 3, 4, 5]
```

10 splice()

- Adds, removes or replaces elements in an array.
- Modifies the original array.

Example :-

```
let arr = [1, 2, 3, 4, 5];
let removed = arr.splice(1, 2, 99);
console.log(removed); // [2, 3]
console.log(arr);     // [1, 99, 4, 5]
```

Quick Comparison :-

Method	Returns	Modifies Original ?	Use Case
push()	length	Yes	Add at end
pop()	element	Yes	Remove last
unshift()	length	Yes	Add at start
shift()	element	Yes	Remove first
map()	new array	No	Transform elements
filter()	new array	No	Filter elements
reduce()	single value	No	Calculate single value

Method	Returns	Modifies Original ?	Use Case
forEach()	undefined	No	Execute for each
find()	element	No	Find first match
findIndex()	index	No	Find index
includes()	boolean	No	Check existence
slice()	new array	No	Copy part of array
splice()	removed elements	Yes	Add/Remove/Replace

Pro Tips :-

- Use map() instead of forEach() when you need a new array.
- Use filter() to filter data instead of writing loops.
- Be careful with splice() as it changes the original array.
- Chain methods for cleaner code !

Karyvio
LEARN | BUILD | GROW | GET HIRED

JS map(), filter() and reduce() in JavaScript

Powerful array methods for real-world programming

Same array
Different purpose
Big possibilities !

1 map()

Ans → map() creates a new array by applying a function to each element of the original array.

- It does not modify the original array.
- It returns a new array of the same length.
- Useful when we want to transform each element.

Syntax :-

```
array.map((element, index, array) => {
   // return new value
});
```

Example :-

```
let nums = [1, 2, 3, 4];
let squares = nums.map(n => n * n);
console.log(squares);   // [1, 4, 9, 16]
console.log(nums);      // [1, 2, 3, 4]
```

Key Points :-

- Returns a new array.
- Original array remains unchanged.
- Callback gets 3 arguments: element, index, array
- Commonly used for transformation.
- Length of new array is same as original.

Real-World Example :-

```
let users = [
  { name: "Sovit", age: 21 },
  { name: "Smruti", age: 22 }
];

let names = users.map(u => u.name);
console.log(names);
// ["Sovit", "Smruti"]
```

2 filter()

Ans → filter() creates a new array with elements that pass a certain condition.

- It does not modify the original array.
- It returns a new array with zero or more elements.
- Useful when we want to select certain elements from an array.

Syntax :-

```
array.filter((element, index, array) => {
   // return true to keep the element
});
```

Example :-

```
let nums = [1, 2, 3, 4, 5, 6];
let even = nums.filter(n => n % 2 === 0);
console.log(even);      // [2, 4, 6]
console.log(nums);      // [1, 2, 3, 4, 5, 6]
```

Key Points :-

- Returns a new array.
- Only elements that return true are included.
- Original array remains unchanged.
- Resulting array can have less elements (or even 0).
- Commonly used for searching, filtering, validation, etc.

Real-World Example :-

```
let products = [
  { name: "Laptop", price: 50000 },
  { name: "Phone", price: 20000 },
  { name: "Tablet", price: 30000 }
];
let affordable = products.filter(p => p.price < 40000);
console.log(affordable);
// [ { name: "Phone", price: 20000 },
//   { name: "Tablet", price: 30000 } ]
```

3 reduce()

Ans → reduce() reduces an array to a single value by applying a function to each element (accumulator pattern).

- It can be used for sum, product, finding max/min, counting, or even building objects.
- It does not modify the original array.

Syntax :-

```
array.reduce((accumulator, element, index, array) => {
   // return new accumulator value
}, initialValue);
```

Example 1 - Sum :-

```
let nums = [1, 2, 3, 4, 5];
let sum = nums.reduce((acc, n) => acc + n, 0);
console.log(sum);       // 15
```

Example 2 - Find Max :-

```
let nums = [10, 5, 8, 20, 3];
let max = nums.reduce((acc, n) =>
            n > acc ? n : acc, nums[0]);
console.log(max);       // 20
```

Key Points :-

- Returns a single value (not an array).
- Takes an initial value (recommended).
- Callback gets 4 arguments: accumulator, element, index, array
- Very powerful and flexible.

Example 3 - Build Object :-

```
let fruits = ["apple", "banana",
              "mango", "apple"];
let count = fruits.reduce((acc, fruit) => {
  acc[fruit] = (acc[fruit] || 0) + 1;
  return acc;
}, {});
console.log(count);
// { apple: 2, banana: 1, mango: 1 }
```

Comparison Table :-

Method	Returns	Purpose	Result Type	Original Array	Common Use Case
map()	New array	Transform each element	Same length	Not modified	Modify values (e.g. square)
filter()	New array	Select elements	Same or less	Not modified	Filter by condition
reduce()	Single value	Reduce to a value	Single value	Not modified	Sum, max, count, etc.

Chaining Methods :-

```
let nums = [1, 2, 3, 4, 5];
let result = nums
    .filter(n => n % 2 === 0)
    .map(n => n * 10)
    .reduce((acc, n) => acc + n, 0);
console.log(result);      // 60
// [2,4] -> [20,40] -> 60)
```

Real-Life Use Case :-

- Get total price of products more than ₹1000 and apply 10% discount.

```
let products = [
  { price: 500 }, { price: 1500 },
  { price: 2000 };
let total = products
    .filter(p => p.price > 1000)
    .map(p => p.price * 0.9)
    .reduce((acc, p) => acc + p, 0);
console.log(total);       // 3150
```

Pro Tips :-

- Always use meaningful variable names (acc, curr, item, etc.).
- Use an initial value in reduce() to avoid unexpected results.
- You can chain multiple array methods (map, filter, reduce) for clean and readable code.
- These methods do not modify the original array (immutable).
- Arrow functions make the code shorter and cleaner.
- Very commonly used in real-world projects (e.g. data transformation, API handling, UI rendering).

Karyvio
LEARN | BUILD | GROW | GET HIRED

Numbers, Math, NaN & Infinity

Numbers are a fundamental data type in JavaScript. Along with basic arithmetic, JavaScript provides a powerful Math object and special numeric values like NaN and Infinity to handle edge cases.

"Numbers power the logic. Math makes it possible. NaN and Infinity handle the edge cases."

1 Numbers in JavaScript

- JavaScript has a single numeric type: Number (64-bit floating point).
- It can represent integers and decimals.
- It can also represent very large and very small numbers using exponential notation.
- Supports special values: NaN, Infinity, -Infinity.

Example :-

```
let a = 10;          // integer
let b = 3.14;        // decimal
let c = 1.2e3;       // 1200 (exponential)
let d = -5;          // negative number

console.log(typeof a); // number
```

2 Number Methods & Properties

- JavaScript provides useful properties and methods on the Number object.
- These help with checking, converting, and formatting numbers.

Example :-

```
let num = 123.456;
console.log(Number.MAX_VALUE);
console.log(Number.MIN_VALUE);
console.log(Number.isInteger(num));
console.log(Number.isNaN(num));
console.log(Number.parseInt("42"));
console.log(Number.parseFloat("3.14"));
```

```
// Output:
1.7976931348623157e+308
5e-324
false
false
42
3.14
```

3 Math Object

- The Math object provides mathematical constants and functions.
- It is not a constructor, so you don't create it using new.
- All methods and properties are static (used directly on Math).

Common Math Properties:

- Math.PI // 3.14159...
- Math.E // 2.71828...
- Math.SQRT2 // 1.4142...
- Math.SQRT1_2 // 0.7071...

Common Math Methods:

- Math.abs(), Math.round(), Math.floor(),
- Math.ceil(), Math.min(), Math.max(),
- Math.random(), Math.pow(), Math.sqrt(),
- Math.log(), Math.exp(), Math.trunc()

Math.random() returns a number between 0 (inclusive) and 1 (exclusive).

4 Basic Math Operations

JavaScript supports all basic arithmetic operators.

Operator	Symbol	Example	Result
Addition	+	5 + 3	8
Subtraction	-	10 - 4	6
Multiplication	*	5 * 3	15
Division	/	10 / 2	5
Modulus (Remainder)	%	10 % 3	1
Exponentiation	**	2 ** 3	8

Example :-

```
console.log(10 % 3);   // 1
console.log(2 ** 3);   // 8
console.log(5 + 2 * 3); // 11 (PEMDAS)
```

5 NaN (Not-a-Number)

- NaN is a special value that means "Not a Number".
- It is returned when a value cannot be converted to a number.
- NaN is of type number.
- NaN is not equal to itself (you must use Number.isNaN()).

Example :-

```
console.log(0 / 0);            // NaN
console.log(parseInt("abc")); // NaN
console.log(typeof NaN);       // number
console.log(NaN === NaN);      // false
console.log(Number.isNaN(NaN)); // true
```

Use Number.isNaN() to check for NaN instead of using == or ===.

6 Infinity

- Infinity represents a value greater than any number.
- It is returned when a number exceeds the maximum value.
- -Infinity is less than any number (e.g., result of dividing a negative number by 0).

Example :-

```
console.log(1 / 0);    // Infinity
console.log(-1 / 0);   // -Infinity
console.log(1e309);    // Infinity
console.log(-1e309);   // -Infinity
```

Infinity is useful for checking limits and handling edge cases.

7 Number Conversion

- JavaScript can convert between strings and numbers using built-in functions.
- Useful when taking input or working with user data.

Example :-

```
console.log(Number("123"));     // 123
console.log(Number("12.34"));   // 12.34
console.log(parseInt("10px"));  // 10
console.log(parseFloat("3.14abc")); // 3.14
```

parseInt() and parseFloat() ignore non-numeric characters at the end.

8 Common Use Cases

1. Working with user input (converting strings to numbers).
2. Performing calculations (e.g., price, score, percentage).
3. Avoiding NaN in computations.
4. Checking for Infinity in division or large calculations.
5. Formatting numbers for display (using toFixed(), toLocaleString()).

Example :-

```
let price = 99.99;
console.log(price.toFixed(2));       // 99.99
console.log(price.toLocaleString()); // 99.99
```

9 Quick Tips

- ✓ Use Number.isNaN() instead of isNaN().
- ✓ Use parseInt() / parseFloat() for string to number conversion.
- ✓ Use Math object for mathematical operations.
- ✓ Check for Infinity when dividing by zero or large numbers.
- ✓ Always validate user input before performing calculations.

"Numbers, Math, NaN and Infinity form the backbone of numerical operations in JavaScript."

Karyvio
LEARN | BUILD | GROW | GET HIRED

JS Objects in JavaScript

Store and manage real-world data using key-value pairs

Objects let us store related data and functionality together. They are one of the most important parts of JavaScript !

1 What is an Object ?

Ans →
- An object is a collection of key-value pairs.
- Keys (also called properties) are strings (or symbols) and values can be of any type (number, string, array, function, another object, etc).
- Objects help us represent real-world entities like a person, product, etc.
- Objects are mutable (we can change their values).

Example :-

```
let person = {
    name: "Sovit",
    age: 21,
    isStudent: true,
    hobbies: ["Coding", "Music"]
};
console.log(person);
// Output :-
{ name: "Sovit", age: 21,
 isStudent: true,
 hobbies: ["Coding", "Music"] }
```

Key Points :-
- Keys are strings (written without quotes in most cases).
- Values can be any data type.
- Objects are mutable.
- Properties can be added, updated or deleted.
- Objects are unordered (but modern JS maintains insertion order).

2 Accessing Object Properties

Ans → We can access object properties using dot notation or bracket notation.

Example :-

```
let person = { name: "Sovit", age: 21 };
console.log(person.name);   // Sovit
console.log(person["age"]); // 21
// We can also use a variable as key
let key = "name";
console.log(person[key]);   // Sovit
```

Note :-
- Use dot notation when you know the key name.
- Use bracket notation when the key name is stored in a variable or has spaces or special characters.

Example :-

```
let user = { "full name": "Sovit" };
console.log(user["full name"]);
```

3 Adding & Updating Properties

Ans → We can add new properties or update existing ones.

Example :-

```
let person = { name: "Sovit", age: 21 };
// Add new property
person.city = "Bhubaneswar";
// Update existing property
person.age = 22;
console.log(person);
// Output :-
{ name: "Sovit", age: 22, city: "Bhubaneswar"}
```

4 Deleting Properties

Ans → Use the delete keyword to remove a property from an object.

Example :-

```
let person = { name: "Sovit", age: 21 };
delete person.age;
console.log(person);
// Output :-
{ name: "Sovit" }
```

5 Checking Property Existence

Ans → We can check if a property exists using in operator or hasOwnProperty().

Example :-

```
let person = { name: "Sovit", age: 21 };
console.log("name" in person);                  // true
console.log("age" in person);                   // true
console.log(person.hasOwnProperty("name")); // true
console.log(person.hasOwnProperty("city")); // false
```

6 Object Methods

Ans → Objects can also have methods (functions as properties).

Example :-

```
let person = {
    name: "Sovit",
    greet: function() {
        console.log("Hello, I am " + this.name);
    }
};
person.greet();    // Hello, I am Sovit
```

Note :- this refers to the current object.

7 Nested Objects

Ans → An object can contain another object as a property.

Example :-

```
let pruden = {
    name: "Sovit",
    address: {
        city: "Bhubaneswar",
        state: "Odisha"
    }
};
console.log(student.address.city); // Bhubaneswar
```

8 Object Destructuring

Ans → We can extract properties from an object using destructuring (ES6).

Example :-

```
let person = { name: "Sovit", age: 21, city: "BBSR"};
let { name, age } = person;
console.log(name);    // Sovit
console.log(age);     // 21
// We can also set default values
let { country = "India" } = person;
console.log(country); // India
```

9 Spread Operator with Objects

Ans → We can copy or merge objects using the spread operator (...).

Example :-

```
let obj1 = { a:1, b: 2 };
let obj2 = { c:3, d: 4 };
let obj3 = { ...obj1, ...obj2 };
console.log(obj3);
// Output :-
{ a: 1, b: 2, c: 3, d: 4 }
```

10 Object.keys(), values(), entries()

Ans → Useful built-in methods to work with objects.

Example :-

```
let person = { name: "Sovit", age: 21, city: "BBSR"};
console.log(Object.keys(person));    // [ 'name', 'age', 'city' ]
console.log(Object.values(person)); // [ 'Sovit', 21, "BBSR" ]
console.log(Object.entries(person));
// [ [ 'name', 'Sovit" ], [ 'age', 21 ], [ 'city', "BBSR" ] ]
```

Real-Life Example :-
- Let's create a product object for an e-commerce website.

```
let product = {
    id: 101,
    name: "Laptop",
    price: 50000,
    inStock: true,
    details: { brand: "Dell", rating: 4.5 }
};
console.log(product.name); // Laptop
```

Practice Tasks :-
1. Create a student object with name, age, and marks.
2. Add a new property 'grade' to it.
3. Update the marks value.
4. Delete the age property.
5. Check if 'grade' exists.
6. Use destructuring to extract name and marks.
7. Merge two objects using spread operator.

Karyvio
LEARN | BUILD | GROW | GET HIRED

Spread & Rest Operator

Use ... to work with arrays, objects and function parameters easily.

"Small syntax...
Big possibilities...
That's JavaScript!"

1 What are Spread and Rest?

- Both use the same syntax ... but work differently depending on the context.
- Spread → expands an array/object into individual elements.
- Rest → collects multiple elements into a single array.

Think of it as:
- Spread → "Unpack"
- Rest → "Pack"

2 Spread Operator with Arrays

- Helps to copy, merge or expand arrays.
- Does not change the original array.

Example :-

```
const arr1 = [1, 2, 3];
const arr2 = [4, 5, 6];
const merged = [...arr1, ...arr2];
console.log(merged);   // [1, 2, 3, 4, 5, 6]

const copy = [...arr1];
console.log(copy);     // [1, 2,3]
```

...[1, 2, 3]
↙ ↓ ↘
1 2 3

Spread expands elements.

3 Spread Operator with Objects

- Helps to copy or merge objects.
- Useful for adding/updating properties.

Example :-

```
const user = { name: "Sovit", age: 22 };
const updatedUser = { ...user, city: "Bhubaneswar" };
console.log(updatedUser);
// { name: "Sovit", age: 22, city: "Bhubaneswar" }
```

If the same key exists, the new value will override the old one.

4 Rest Operator in Functions

- Collects multiple arguments into an array.
- Useful when we don't know how many arguments will be passed.

Example :-

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

console.log(sum(1, 2, 3));          // 6
console.log(sum(10, 20, 30, 40));   // 100
```

Rest collects elements into an array.

5 Rest Operator with Array Destructuring

- Collects the remaining elements into a new array.

Example :-

```
const [first, second, ...rest] = [10, 20, 30, 40, 50];
console.log(first);     // 10
console.log(second);    // 20
console.log(rest);      // [30, 40, 50]
```

6 Rest Operator with Object Destructuring

- Collects the remaining properties into a new object.

Example :-

```
const { name, age, ...others } = {
    name: "Sovit",
    age: 22,
    city: "Bhubaneswar",
    job: "Developer"
};
console.log(name);      // Sovit
console.log(age);       // 22
console.log(others);    // { city: "Bhubaneswar", job: "Developer" }
```

7 Shallow Copy vs Deep Copy

- Spread creates a shallow copy (copies only first level).
- For nested objects/arrays, use deep copy (structuredClone or JSON methods).

Shallow Copy (using spread)

```
const obj1 = { a: 1, b: { c: 2 }};
const obj2 = { ...obj1 };
obj2.b.c = 100;
console.log(obj1);  // { a: 1, b: { c: 100 }}
// nested object is still shared!
```

Deep Copy (using structuredClone)

```
const obj1 = { a: 1, b: { c: 2 }};
const obj2 = structuredClone(obj1);
obj2.b.c = 100;
console.log(obj1);  // { a: 1, b: { c: 2 }}
// original remains unchanged
```

8 Use Cases & Best Practices

- ✓ Merging arrays and objects
- ✓ Copying data without modifying original
- ✓ Passing multiple arguments
- ✓ Destructuring arrays and objects
- ✓ Handling function parameters
- ✓ Useful in React and modern JavaScript

Important Tips :-
- Spread = expands elements.
- Rest = collects elements.
- Spread makes a shallow copy (not deep).
- Use structuredClone() or JSON.parse(JSON.stringify()) for deep copy (with limitations).

Karyvio
LEARN | BUILD | GROW | GET HIRED

Prototypes and Prototypal Inheritance in JavaScript

JavaScript uses prototypal inheritance instead of classical inheritance. Every object in JavaScript has an internal link to another object called its prototype. This allows objects to inherit properties and methods from other objects.

"Prototypes make JavaScript flexible, powerful, and unique."

1 What is a Prototype?

- A prototype is another object that an object inherits properties and methods from.
- Every JavaScript object has an internal [[Prototype]] link (also accessible via __proto__).
- When you try to access a property or method, JavaScript first looks in the object itself, then in its prototype, then in the prototype's prototype, and so on (prototype chain).

Example :-

```
const person = {
  name: "Sovit"
};
console.log(person.__proto__); // {} (Object prototype)
```

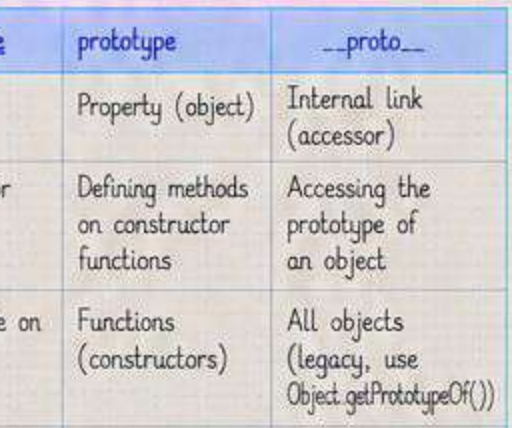
2 Prototype Chain

- If a property is not found in the current object, JavaScript looks up in the prototype.
- This continues until it finds the property or reaches null.
- The top of the chain is Object.prototype, which has no prototype.

Example :-

```
const arr = [1, 2, 3];
console.log(arr.toString(); // from Array.prototype
console.log(arr.__proto__); // Array.prototype
console.log(arr.__proto__.__proto__);
          // Object.prototype
console.log(arr.__proto__.__proto__.__proto__);
          // null
```

Prototype Chain (Array Example)



3 Creating Objects with Prototypes

- You can set the prototype of an object using Object.create().
- This allows an object to inherit from another object.

Example :-

```
const animal = {
  speak() {
    console.log("Animal makes a sound");
  }
};

const dog = Object.create(animal);
dog.speak();   // Animal makes a sound
console.log(dog.__proto__ === animal); // true
```

Note :-

- Object.create(proto) creates a new object with the given prototype.
- The new object does not copy properties, it inherits them through the prototype chain.

4 Constructor Functions and Prototype

- When you create objects using constructor functions, methods are added to the prototype so that all instances can share them.
- This is more memory efficient than defining methods inside the constructor.

Example :-

```
function Person(name, age) {
  this.name = name;
  this.age = age;
}

Person.prototype.greet = function() {
  console.log(`Hello, I'm ${this.name}`);
};

const p1 = new Person("Sovit", 22);
const p2 = new Person("Smruti", 21);

p1.greet();   // Hello, I'm Sovit
p2.greet();   // Hello, I'm Smruti
```

5 prototype vs __proto__

Feature	prototype	__proto__
Type	Property (object)	Internal link (accessor)
Used for	Defining methods on constructor functions	Accessing the prototype of an object
Available on	Functions (constructors)	All objects (legacy, use Object.getPrototypeOf())
Editable	Yes	Yes (but not recommended)

Tip :-

- Use Object.getPrototypeOf(obj) instead of __proto__ (modern approach).
- Use Object.setPrototypeOf(obj, proto) to set prototype (use with caution).

6 Inheritance Example

- You can achieve inheritance by setting up the prototype chain.

Example :-

```
function Animal(name) {
  this.name = name;
}
Animal.prototype.sayHello = function() {
  console.log(`Hi, I am ${this.name}`);
};

function Dog(name, breed) {
  Animal.call(this, name);  // call parent constructor
  this.breed = breed;
}
Dog.prototype = Object.create(Animal.prototype);
Dog.prototype.constructor = Dog; // reset constructor

Dog.prototype.bark = function() {
  console.log("Woof! Woof!");
};

const d = new Dog("Tommy", "Golden Retriever");
d.sayHello(); // Hi, I am Tommy
d.bark();     // Woof! Woof!
```

7 Built-in Prototypes

- All built-in objects (Array, String, Number, etc.) have prototypes.
- You can add custom methods to built-in prototypes (not recommended in production).

Example :-

```
Array.prototype.last = function() {
  return this[this.length - 1];
};

const numbers = [10, 20, 30];
console.log(numbers.last());    // 30
```

8 Key Takeaways

- ✓ Every object has a prototype.
- ✓ JavaScript uses prototypal inheritance.
- ✓ Use prototype to add shared methods.
- ✓ Use Object.create() for inheritance.
- ✓ Understand the prototype chain.
- ✓ Avoid modifying built-in prototypes.
- ✓ Use modern methods like Object.getPrototypeOf().

9 Common Interview Questions

1. What is a prototype in JavaScript?
2. Explain the prototype chain.
3. What is the difference between prototype and __proto__?
4. How does inheritance work in JavaScript?
5. Why is Object.create() useful?
6. What is Object.getPrototypeOf()?
7. Can you modify built-in prototypes? Should you?

Karyvio
LEARN | BUILD | GROW | GET HIRED

Date & Time in JavaScript

Work with dates, times, timestamps and perform useful operations.

"Dates are just numbers, but JavaScript makes them useful."

1 What is Date Object?

- In JavaScript, Date is a built-in object used to work with dates and times.
- It stores the number of milliseconds since 1 January 1970 (Unix Epoch).
- Date objects can be created in different ways and provide many useful methods.

2 Creating Date Objects

Example :-

```
const now = new Date();                          // current date & time
const specific = new Date('2025-09-12');         // specific date (YYYY-MM-DD)
const dateTime = new Date(2025, 8, 12, 10, 30, 0); // year, month(0-11), day, hour....
const timestamp = new Date(167253120000);        // from timestamp (ms)

console.log(now);          // e.g. Fri Sep 12 2025 17:30:00 GMT+0530 (IST)
console.log(specific);     // Fri Sep 12 2025 00:00:00 GMT+0530 (IST)
console.log(dateTime);     // Fri Sep 12 2025 10:30:00 GMT+0530 (IST)
console.log(timestamp);    // Sun Jan 01 2023 05:30:00 GMT+0530 (IST)
```

3 Important Methods

Method	Description
getFullYear()	Returns the year (e.g. 2025)
getMonth()	Returns the month (0 - 11)
getDate()	Returns the day of the month (1 - 31)
getDay()	Returns the day of the week (0 - 6)
getHours()	Returns hours (0 - 23)
getMinutes()	Returns minutes (0 - 59)
getSeconds()	Returns seconds (0 - 59)
getMilliseconds()	Returns milliseconds (0 - 999)
getTime()	Returns timestamp (ms since 1970)
toDateString()	Returns date as string
toTimeString()	Returns time as string
toLocaleString()	Returns local date and time
toISOString()	Returns date in ISO format (UTC)

4 Examples - Getting Date Parts

Example :-

```
const now = new Date();
console.log(now.getFullYear());      // 2025
console.log(now.getMonth());         // 8  (September, 0-indexed)
console.log(now.getDate());          // 12
console.log(now.getDay());           // 5  (Friday)
console.log(now.getHours());         // 17
console.log(now.getMinutes());       // 30
console.log(now.getSeconds());       // 0
console.log(now.toDateString());     // Fri Sep 12 2025
console.log(now.toTimeString());     // 17:30:00 GMT+0530 (IST)
console.log(now.toLocaleString());   // 12/9/2025, 5:30:00 pm
console.log(now.toISOString());      // 2025-09-12T12:00:00.000Z
```

5 Setting Date Parts

Example :-

```
const d = new Date();
d.setFullYear(2026);   // set year
d.setMonth(11);        // set month (December)
d.setDate(25);         // set day
d.setHours(10);        // set hours
d.setMinutes(45);      // set minutes
d.setSeconds(30);      // set seconds

console.log(d);  // Fri Dec 25 2026 10:45:30 ...
```

Note :-

- Month is 0-indexed (0 = January, 11 = December).
- If you set a value out of range, JavaScript automatically adjusts it.

6 Timestamps & Calculations

Example :-

```
const now = new Date();
const timestamp = now.getTime();   // milliseconds
console.log(timestamp);

// Convert to seconds
console.log(Math.floor(timestamp / 1000));

// Add 7 days
const nextWeek = new Date(now.getTime() + 7 * 24 * 60 * 60 * 1000);
console.log(nextWeek);

// Difference between two dates
const d1 = new Date('2025-01-01');
const d2 = new Date('2025-09-12');
const diffMs = d2 - d1;
console.log(diffMs / (1000 * 60 * 60 * 24));   // days
```

7 Real-world Use Cases

- ✔ Displaying current date and time on a website.
- ✔ Scheduling events, deadlines or reminders.
- ✔ Calculating time difference (e.g., age, days left).
- ✔ Formatting dates for users.
- ✔ Working with timestamps from APIs (e.g., JSON data).
- ✔ Building countdown timers.

8 Common Interview Questions

1. What is the purpose of the Date object?
2. Why is month 0-indexed in JavaScript?
3. How do you get the current timestamp?
4. How do you calculate the difference between two dates?
5. How do you format a date in YYYY-MM-DD format?
6. How can you add days/hours to a date?

9 Tips & Best Practices

- ✔ Always be careful with timezones (use toISOString() for UTC).
- ✔ Use meaningful date formats for users (e.g., toLocaleString()).
- ✔ Remember month is 0-indexed.
- ✔ For date calculations, use timestamps (millisecond values).
- ✔ Consider using libraries like date-fns or Moment.js for complex date handling.

Karyvio
LEARN | BUILD | GROW | GET HIRED

Set & Map in JavaScript

Set and Map are modern JavaScript data structures introduced in ES6. They help us store unique values (Set) and key-value pairs (Map) with more powerful features than regular objects and arrays.

"Use the right data structure for the right problem."

1 What is a Set?

- A Set is a collection of unique values.
- It does not allow duplicate values.
- It can store values of any type (primitive or object).
- The insertion order is maintained.

Think of it as:

- A list of unique items
- Useful when you only care about values, not key-value pairs.

2 Creating a Set

Example :-

```
const s1 = new Set();                    // empty set
const    = new Set([1, 2, 3, 3, 2, 1]);  // duplicates removed
console.log(s2);   // Set(3) { 1, 2, 3 }
```

Adding values to a Set

```
s1.add(10);
s1.add(20);
s1.add(10);          // duplicate, ignored
console.log(s1);     // Set(2) { 10, 20 }
```

Duplicates are automatically ignored.

3 Set Methods

Method	Description
add(value)	Adds a value to the set
delete(value)	Removes a value from the set
has(value)	Checks if a value exists
clear()	Removes all values
size	Returns the number of values

4 Set Example in Practice

Example :-

```
const fruits = new Set(["apple", "banana", "apple"]);
fruits.add("mango");
console.log(fruits.has("banana"));    // true
console.log(fruits.size);             // 3
fruits.delete("apple");
console.log(fruits);    // Set(2) { 'banana', 'mango' }
fruits.clear();
console.log(fruits);    // Set(0) { }
```

Set is great for removing duplicates from an array!

5 Removing Duplicates from an Array

Example :-

```
const numbers = [1, 2, 2, 3, 4, 4, 5];
const unique = [...new Set(numbers)];
console.log(unique);          // [1, 2, 3, 4, 5]
```

Very common use case in interviews!

6 What is a Map?

- A Map is a collection of key-value pairs.
- Keys can be of any type (not just strings).
- Remembers the insertion order.
- Provides better performance and more features than plain objects in many cases.

7 Creating a Map

Example :-

```
const m1 = new Map();      // empty map
const m2 = new Map([
  ["name", "Sovit"],
  ["age", 22],
]);
console.log(m2);   // Map(2) { 'name' => 'Sovit', 'age' => 22 }
```

Map can have keys of any type (strings, numbers, objects, functions).

8 Map Methods

Method	Description
set(key, value)	Adds or updates a key-value pair
get(key)	Returns the value for a key
has(key)	Checks if a key exists
delete(key)	Removes a key-value pair
clear()	Removes all entries
size	Returns the number of key-value pairs

9 Map Example in Practice

Example :-

```
const user = new Map();
user.set("name", "Sovit");
user.set(101, "Student");
user.set(true, "Active");
console.log(user.get("name"));     // Sovit
console.log(user.has(101));        // true
user.delete(true);
console.log(user.size);            // 2
console.log(user);   // Map(2) { 'name' => 'Sovit', 101 => 'Student' }
```

10 Set vs Map (Quick Comparison)

Feature	Set	Map
Stores	Unique values	Key-value pairs
Duplicate values	Not allowed	Keys must be unique
Key type	Values only	Any type (string, number, object, etc.)
Order	Maintains insertion order	Maintains insertion order
Methods	add, delete, has, clear, size	set, get, has, delete, clear, size
Use case	Remove duplicates, membership check	Store related data (like IDs and values)

Karyvio
LEARN | BUILD | GROW | GET HIRED

DOM Selection and Traversal in JavaScript

"Select elements wisely, because every great interaction starts with finding the right element."

The Document Object Model (DOM) represents the structure of a web page as a tree of objects. JavaScript can select, navigate, and manipulate these elements using various methods. Selecting and traversing elements is the first step before you can change their content, styles, or attributes.

1 Ways to Select Elements

- JavaScript provides multiple ways to select elements from the DOM:
  - getElementById()
  - getElementsByClassName()
  - getElementsByTagName()
  - querySelector()
  - querySelectorAll()

2 getElementById()

- Selects a single element by its unique id.
- Returns the element or null if not found.

Example :-

```
const heading = document.getElementById("title");
console.log(heading);  // <h1 id="title">Hello</h1>
```

3 getElementsByClassName()

- Selects all elements with a given class name.
- Returns a live HTMLCollection.

Example :-

```
const items = document.getElementsByClassName("item");
console.log(items);     // HTMLCollection(3)
console.log(items[0]);  // first element
```

4 getElementsByTagName()

- Selects all elements with a given tag name.
- Returns a live HTMLCollection.

Example :-

```
const paragraphs = document.getElementsByTagName("p");
console.log(paragraphs);     // HTMLCollection
console.log(paragraphs[0]);  // first <p> element
```

5 querySelector()

- Selects the first element that matches a CSS selector.
- Returns the element or null.

Example :-

```
const firstItem = document.querySelector(".item");
console.log(firstItem);  // first element with class

const btn = document.querySelector("#submit");
console.log(btn);     // element with id="submit"
```

6 querySelectorAll()

- Selects all elements that match a CSS selector.
- Returns a static NodeList.

Example :-

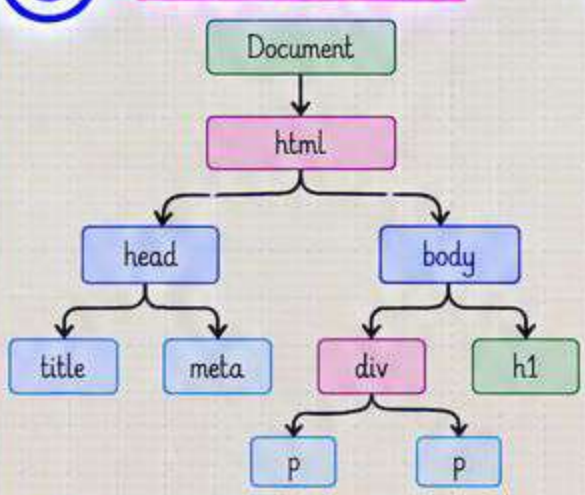
```
const items = document.querySelectorAll(".item");
console.log(items);     // NodeList
items.forEach((el, index) => {
  console.log(index, el);
});
```

7 CSS Selectors You Can Use

You can use any valid CSS selector:

Selector	Example
#id	document.querySelector("#title")
.class	document.querySelector(".card")
tag	document.querySelector("div")
.class1.class2	document.querySelector(".box.active")
attribute	document.querySelector("input[type='text']")
descendant	document.querySelector("ul li")
child	document.querySelector("ul > li")
pseudo-class	document.querySelector("a:hover")

8 DOM Tree Structure



Note :-

- The DOM is a tree structure.
- You can select the parent, child, or sibling elements using traversal properties.

9 Why Traversal?

- After selecting an element, you often need to navigate to related elements: parent, children, siblings, etc.
- Traversal helps you move through the DOM tree.

Tip :-

Think of the DOM like a family tree: each element has parents, children, and siblings.

10 Common Traversal Properties

Property	Description
parentElement	Returns the parent element.
children	Returns HTMLCollection of child elements.
firstElementChild	Returns the first child element.
lastElementChild	Returns the last child element.
nextElementSibling	Returns the next sibling element.
previousElementSibling	Returns the previous sibling element.
closest(selector)	Returns the nearest ancestor that matches the selector.

11 Traversal Examples

Example :-

```
const div = document.querySelector(".box");

// Parent element
console.log(div.parentElement);  // <body>

// Children
console.log(div.children);          // HTMLCollection
console.log(div.firstElementChild); // first child

// Siblings
console.log(div.nextElementSibling);     // next sibling
console.log(div.previousElementSibling); // previous sibling

// Closest ancestor
console.log(div.closest("section"));  // nearest <section>
```

12 Key Takeaways

- ✓ Use the right method to select elements (ID, class, tag, or querySelector).
- ✓ querySelector() and querySelectorAll() are more flexible (use CSS selectors).
- ✓ Traversal helps you navigate between related elements.
- ✓ Understand the DOM tree structure.
- ✓ Always check if an element exists before using it.
- ✓ Use closest() to find the nearest matching ancestor.

Karyvio
LEARN | BUILD | GROW | GET HIRED

DOM Manipulation in JavaScript

DOM Manipulation means changing the content, structure, and style of HTML elements using JavaScript. After selecting elements from the DOM, we can modify them by adding, removing, or updating content, attributes, and styles.

"With JavaScript, we can turn a static web page into a dynamic and interactive experience."

1 Changing Content

- We can change the content inside an element using:
  - innerHTML (parses HTML)
  - textContent (only text)
  - innerText (visible text, respects CSS)

Example :-

```
const title = document.getElementById("title");

title.innerHTML = "<b>Welcome to Karyvio</b>";
title.textContent = "Learn JavaScript";
title.innerText = "Build, Grow, Get Hired");
```

Note :-

- Use textContent to avoid XSS attacks when inserting user input.
- innerHTML can be used when you want to insert actual HTML.

2 Changing Attributes

- We can get, set, or remove attributes of an element using:
  - getAttribute()
  - setAttribute()
  - removeAttribute()

Example :-

```
const link = document.getElementById("link");
link.setAttribute("href", "https://karyvio.com");
link.setAttribute("target", "_blank");

console.log(link.getAttribute("href"));
link.removeAttribute("target");
```

Tip :-

- You can also use element.id, element.className, element.src, etc. directly for common attributes.

3 Changing Styles

- We can change CSS styles using the style property.
- Styles are written in camelCase (e.g., backgroundColor).

Example :-

```
const box = document.getElementById("box");
box.style.backgroundColor = "lightblue";
box.style.color = "darkblue";
box.style.fontSize = "20px";
box.style.padding = "10px";
box.style.border = "2px solid blue";
```

Note :-

- Inline styles override external CSS.
- For multiple styles, you can also use classList (recommended).

4 Adding and Removing Elements

- We can create new elements, add them to the DOM, and also remove existing elements.

Example :-

```
// Create a new element
const newPara = document.createElement("p");
newPara.textContent = "This is a new paragraph!";
newPara.style.color = "green";

// Add it to the DOM
const container = document.getElementById("container");
container.appendChild(newPara);

// Remove the element
document.getElementById("removeBtn").addEventListener("click", () => {
  container.removeChild(newPara);
});
```

5 Useful DOM Manipulation Methods

Method	Description
createElement()	Creates a new HTML element
appendChild()	Adds a child element at the end
removeChild()	Removes a child element
replaceChild()	Replaces a child element with another
insertBefore()	Inserts an element before another
innerHTML	Sets or returns HTML content
textContent	Sets or returns text content
setAttribute()	Sets an attribute value
getAttribute()	Gets an attribute value
removeAttribute()	Removes an attribute
classList.add()	Adds a CSS class
classList.remove()	Removes a CSS class
classList.toggle()	Toggles a CSS class
style.property	Changes inline CSS styles

6 Working with classList

- The classList property provides easy methods to add, remove, toggle, or check CSS classes.

Example :-

```
const btn = document.getElementById("btn");

btn.classList.add("active");      // add class
btn.classList.remove("active");   // remove class
btn.classList.toggle("active");   // toggle class
console.log(btn.classList.contains("active"));
                                  // check if class exists
```

7 Real Example

- Let's create a button that changes text and style when clicked.

Example :-

```
const btn = document.getElementById("changeBtn");
const heading = document.getElementById("heading");

btn.addEventListener("click", () => {
  heading.textContent = "Welcome to Karyvio!";
  heading.style.color = "purple";
  heading.style.fontSize = "24px";
  heading.classList.add("highlight");
});
```

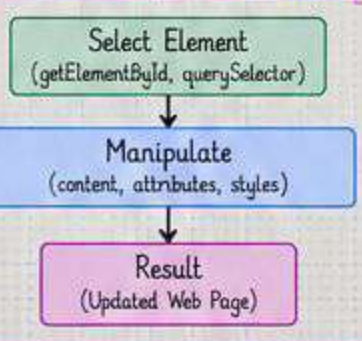
8 Key Takeaways

- ✓ Use innerHTML, textContent, or innerText to change content.
- ✓ Use setAttribute(), getAttribute(), removeAttribute() for attributes.
- ✓ Use the style property or classList for styling.
- ✓ Use createElement(), appendChild(), removeChild() to add or remove elements.
- ✓ DOM manipulation makes web pages dynamic and interactive.
- ✓ Always prefer classList and external CSS for cleaner code.

9 Common Interview Questions

1. What is DOM manipulation?
2. Difference between innerHTML, innerText and textContent?
3. How to create and add a new element?
4. How to remove an element from the DOM?
5. How to change an element's style using JavaScript?
6. What is the difference between classList.add() and className?
7. How do you check if an element has a class?
8. Why is textContent safer than innerHTML?

DOM Manipulation Flow



Select
→ Change
→ See the result in the browser

Karyvio
LEARN | BUILD | GROW | GET HIRED

Forms and Form Validation in JavaScript

Forms are used to collect user input. JavaScript helps us validate the form data before submitting it to ensure correct, secure, and user-friendly input.

"Good forms don't just collect data, they create a better user experience."

1 HTML Form Basics

- A form is created using the <form> tag.
- It contains input elements like text, email, password, checkbox, radio, etc.
- The action attribute defines where to send the data, and method specifies the HTTP method (GET or POST).

Example :-

```
<form id="signupForm" action="/submit" method="POST">
  <label>Name: <input type="text" id="name" /></label>
  <label>Email: <input type="email" id="email" /></label>
  <label>Password: <input type="password" id="password" /></label>
  <button type="submit">Register</button>
</form>
```

2 Accessing Form Elements

- We can access form elements using getElementById, querySelector, or form.elements.
- We can also access all form data using the FormData API.

Example :-

```
const form = document.getElementById("signupForm");
const nameInput = document.getElementById("name");
const emailInput = document.querySelector("#email");
const passwordInput = document.querySelector("#password");

// Using FormData
const formData = new FormData(form);
console.log(formData.get("name"));
```

3 Handling Form Submission

- We can handle form submission using the submit event.
- Use event.preventDefault() to stop default page reload.
- Then validate and process the data.

Example :-

```
form.addEventListener("submit", (e) => {
  e.preventDefault(); // stop page reload
  console.log("Form submitted!");
  // validate and process data here
});
```

Note :-

- Always prevent default if you want to handle form data using JavaScript.

4 Common Input Types

- HTML provides different input types for different kinds of data.

Type	Purpose
text	Single line text input
email	Email address (built-in validation)
password	Hides the entered text
number	Numeric input (with up/down)
tel	Phone number
date	Date picker
checkbox	Multiple selection (true/false)
radio	Single selection form options
file	Upload files
submit	Submit the form

5 Basic Form Validation

- Validate input values before submitting.
- Check for empty fields, correct format, length, etc.
- Show meaningful error messages to the user.

Example :-

```
function validateForm() {
  const name = nameInput.value.trim();
  const email = emailInput.value.trim();
  const password = passwordInput.value;

  if (name === "") {
    showError("nameError", "Name is required");
    return false;
  }
  if (!email.includes("@")) {
    showError("emailError", "Valid email is required");
    return false;
  }
  if (password.length < 6) {
    showError("passwordError", "Password must be at least 6 characters");
    return false;
  }
  return true;
}
```

6 Displaying Error Messages

- We can show error messages near each input.
- Use a or to display errors.
- Clear the error when the input is corrected.

Example :-

```
function showError(id, message) {
  document.getElementById(id).textContent = message;
  document.getElementById(id).style.color = "red";
}

function clearError(id) {
  document.getElementById(id).textContent = "";
}
```

Tip :-

- Show clear and user-friendly error messages.
- Highlight invalid fields (e.g., red border).
- Use real-time validation on input events.

7 Real-time Validation

- We can validate input as the user types using the input event.
- This improves user experience.

Example :-

```
emailInput.addEventListener("input", () => {
  const email = emailInput.value.trim();
  if (email && !email.includes("@")) {
    showError("emailError", "Invalid email format");
  } else {
    clearError("emailError");
  }
});
```

8 Complete Example

- A simple registration form with validation:

Example :-

```
form.addEventListener("submit", (e) => {
  e.preventDefault();
  if (validateForm()) {
    alert("Form submitted successfully!");
    form.reset();
  }
});
```

Output (on valid input) :-

9 Best Practices

- ✓ Always validate input on both client-side (JavaScript) and server-side.
- ✓ Use meaningful error messages.
- ✓ Use built-in HTML validation (required, pattern, type).
- ✓ Avoid inline JavaScript.
- ✓ Keep the form accessible (label, aria).
- ✓ Sanitize user input to prevent security risks (e.g., XSS).
- ✓ Provide a good user experience (real-time validation, clear UI).

10 Common Interview Questions

1. How do you prevent form submission in JS?
2. How can you validate an email input?
3. What is the FormData API?
4. How do you show error messages in a form?
5. What is the difference between client-side and server-side validation?
6. How can you reset a form using JavaScript?
7. How do you handle file uploads in a form?
8. What are the best practices for form validation?

Event Handling in JavaScript (Bubbling, Capturing & Delegation)

Events allow JavaScript to respond to user interactions (like clicks, key presses, form input) or browser actions (like page load, resize, etc.). Understanding event flow is important for writing efficient and scalable code.

"Events turn static web pages into interactive experiences."

1 What is an Event?

- An event is an action or occurrence that happens in the browser.
- Examples: click, input, submit, keydown, load, resize, etc.
- We can listen for events and run a function when the event occurs.

Example :-

```
<button id="btn">Click Me</button>
<script>
  document.getElementById("btn")
    .addEventListener("click", () => {
      console.log("Button clicked!");
  });
</script>
```

Output:
(Button clicked in console when clicked)

2 Ways to Handle Events

- There are multiple ways to handle events in JavaScript:
  1. Inline event handlers
  2. Using DOM properties (on-event)
  3. Using addEventListener() (recommended)

Example :-

```
// 1. Inline event handler
<button onclick="alert('Hello!')">Click</button>
// 2. Using DOM property
const btn = document.getElementById("btn");
btn.onclick = function() {
  alert("Hello using onclick!");
};
// 3. Using addEventListener (Recommended)
btn.addEventListener("click", () => {
  alert("Hello using addEventListener!");
});
```

3 Common Mouse Events

Event	Description
click	Fires when an element is clicked
dblclick	Fires on double click
mouseduwn	When mouse button is pressed
mouseup	When mouse button is released
mouseenter	When mouse enters an element
mouseleave	When mouse leaves an element
mousenowe	Fires when mouse moves
contextmenu	Fires on right click

Tip :-

- Use mouse events for interactive UI elements like buttons, menus, and drag-drop features.
- Use preventDefault() for events like contextmenu if you want to disable default behavior.

4 Common Keyboard Events

Event	Description
keydown	Fires when a key is pressed down
keyup	Fires when a key is released
keypress	(Deprecated) Use keydown/keyup
input	Fires when input value changes
change	Fires when input loses focus (after value change)

Example :-

```
const input = document.getElementById("name");
input.addEventListener("keydown", (e) => {
  console.log("Key pressed: ", e.key);
});
```

5 Common Form Events

Event	Description
submit	Fires when a form is submitted
input	Fires when input value changes
change	Fires when input loses focus
focus	Fires when element gets focus
blur	Fires when element loses focus
reset	Fires when form is reset

Example :-

```
const form = document.getElementById("myForm");
form.addEventListener("submit", (e) => {
  e.preventDefault(); // prevent page reload
  console.log("Form submitted!");
});
```

6 Event Object

- When an event occurs, an **event object** is automatically passed to the event handler function.
- It contains useful information about the event.

Example :-

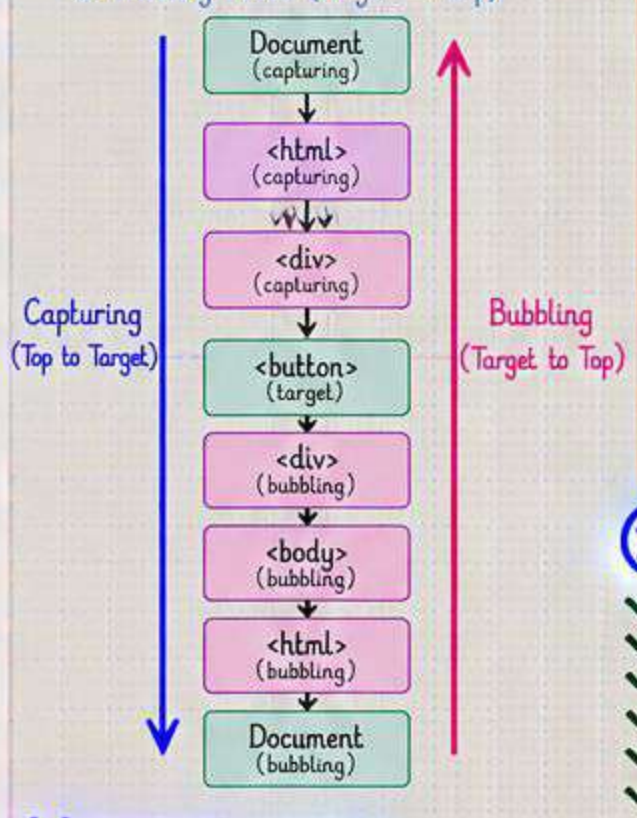
```
btn.addEventListener("click", (e) => {
  console.log(e.type);          // "click"
  console.log(e.target);        // element clicked
  console.log(e.clientX, e.clientY); // mouse position
  console.log(e.preventDefault); // method
});
```

Note :-

- Use e.preventDefault() to stop the default action (e.g., form submission, link navigation).

7 Event Propagation

- When an event occurs on an element, it can propagate (travel) through the DOM tree.
- There are three phases:
  1. Capturing Phase (top → target)
  2. Target Phase (at target element)
  3. Bubbling Phase (target → top)



8 Capturing vs Bubbling

- By default, events use bubbling (false).
- You can set the third paremeter of addEventListener to **true** to use capturing.

Example :-

```
const box = document.getElementById("box");
box.addEventListener("click", (o) => {
  console.log("Bubbling phase (default)");
});

box.addEventListener("click", () => {
  console.log("Capturing phase");
}, true); // true = capturing phase
```

Note :-

- Use capturing when you want to handle the event before it reaches the target element. Use bubbling for most cases.

9 Event Delegation

- Event delegation is a technique where we take advantage of event bubbling to handle events on multiple child elements using a single event listener.
- Useful for dynamic content and better performance.

Example :-

```
<ul id="list">
    <li>Apple</li>
    <li>Banana</li>
    <li>Orange</li>
</ul>

const list = document.getElementById("list");
list.addEventListener("click", (e) => {
  if (e.target.tagName === "LI") {
    console.log("You clicked: ", e.target.textContent);
  }
});
```

10 Key Takeaways

- ✔ Events make web pages interactive.
- ✔ Use addEventListener() (best practice).
- ✔ Understand event propagation (capturing & bubbling).
- ✔ Use event delegation for dynamic content.
- ✔ Use preventDefault() when needed.
- ✔ Access useful event data from the event object.

11 Common Interview Questions

1. What is event bubbling and event capturing?
2. How do you stop event propagation?
3. Difference between event.target and event.currentTarget?
4. What is event delegation? Why is it useful?
5. How do you prevent the default behavior of a link or form?
6. What are common mouse and keyboard events?
7. How does the event loop relate to event handling?

Karyvio
LEARN | BUILD | GROW | GET HIRED

Browser APIs in JavaScript

Browser APIs are built-in interfaces provided by the web browser that allow JavaScript to interact with the browser, the document (DOM), device features, network, storage, and more. They help us build powerful, real-world web applications.

"Browser APIs extend JavaScript beyond the language, giving it superpowers in the real world."

1 What are Browser APIs?

- Provided by the browser (not part of core JavaScript).
- Allow JavaScript to interact with browser features and the user's environment.
- Examples: DOM, BOM, Fetch, Storage, Geolocation, Timers, etc.
- Defined by web standards (like WHATWG).

Note :-

- Browser APIs are available in the browser environment, not in Node.js (which has different APIs).

2 DOM API (Document API)

- Used to access and manipulate HTML elements on the page.
- Allows dynamic updates to content, structure, and style.

Example :-

```
const title = document.getElementById("title");
title.textContent = "Welcome to Karyvio!";
title.style.color = "blue";

document.querySelector(".btn").addEventListener(
  "click", () => {
    alert("Button clicked!");
  }
);
```

3 BOM API (Browser Object Model)

- Used to interact with the browser window, tabs, navigation, and screen.
- Provides information about the browser environment.

Example :-

```
console.log(window.innerWidth); // viewport width
console.log(window.innerHeight); // viewport height
console.log(window.location.href); // current URL

window.open("https://karyvio.com", "_blank");
window.scrollTo(0, 500); // scroll down
console.log(navigator.userAgent); // browser info
```

Tip :-

- The window object is the global object in browsers and represents the browser window.

4 Fetch API (Network Requests)

- Used to make HTTP requasle in fetch data from a server.
- Returns a Promise and works with async/await.
- Replaces older methods like XMLHttpRequest (Ajax).

Example :-

```
// Using fetch with async/await
async function getData() {
  try {
    const res = await fetch("https://jsonplaceholder.typicode.com/posts");
    const data = await res.json();
    console.log(data);
  } catch (error){
    console.error("Error:", error);
  }
}
getData();
```

5 Web Storage API

- Provides a way to store data in the browser.
- Includes:
  - localStorage (persists even after browser close)
  - sessionStorage (clears when tab is closed)

Example :-

```
// localStorage
localStorage.setItem("theme", "dark");
console.log(localStorage.getItem("theme"));
localStorage.removeItem("theme");

// sessionStorage
sessionStorage.setItem("user", "Sovit");
console.log(sessionStorage.getItem("user"));
sessionStorage.clear();
```

Note :-

- Both store only string values.
- Use JSON.parse() and JSON.stringify() for objects.

6 Geolocation API

- Provides access to the user's geographical location (with permission).
- Useful for maps, location-based services, etc.

Example :-

```
if (navigator.geolocation) {
  navigator.geolocation.getCurrentPosition(
    (position) => {
      const lat = position.coords.latitude;
      const lon = position.coords.longitude;
      console.log("Latitude:", lat);
      console.log("Longitude:", lon);
    },
    (error) => {
      console.error("Error getting location:", error);
    }
  );
} else {
  console.log("Geolocation is not supported");
}
```

7 Timers API

- Used to execute code after a delay or repeatedly at intervals.
- Includes setTimeout(), setInterval(), clearTimeout(), clearInterval().

Example :-

```
// setTimeout
const timeoutId = setTimeout(() => {
  console.log("Runs after 2 seconds");
}, 2000);

// setInterval
const intervalId = setInterval(() => {
  console.log("Runs every 1 second");
}, 1000);

// Stop them
clearTimeout(timeoutId);
clearInterval(intervalId);
```

8 History API

- Allows navigation through the browser's session history.
- Useful for single-page applications (SPAs).

Example :-

```
// Add a new entry
history.pushState({ page: "home" }, "Home", "/home");

// Replace current entry
history.replaceState({ page: "about" }, "About", "/about");

// Navigate
history.back();    // go to previous page
history.forward(); // go to next page
console.log(history.state); // current state object
```

Tip :-

- History API is commonly used with the popstate event in SPAs.

9 More Useful Browser APIs

API	Purpose
Clipboard API	Read and write text to clipboard
Notification API	Show browser notifications
Local/Session Storage	Store data locally in browser
Canvas API	Draw graphics and images
Web Audio API	Work with audio (sound, music)
WebRTC API	Real-time communication (video, voice, data)
Intersection Observer	Detect when elements are visible in viewport
Match Media API	Check media queries (responsive design)
Performance API	Measure page load performance
Device Orientation API	Access device motion/orientation
Vibration API	Trigger device vibration (mobile)

10 Best Practices

- ✓ Always check for API support (feature detection).
- ✓ Handle errors (permissions, network failures, etc.).
- ✓ Use async/await for cleaner asynchronous code.
- ✓ Avoid blocking the main thread (use Web Workers for heavy tasks).
- ✓ Respect user privacy (e.g., ask permission for geolocation).
- ✓ Follow web standards and best practices.

11 Common Interview Questions

1. What are Browser APIs?
2. Difference between DOM and BOM?
3. How does the Fetch API work?
4. What is localStorage vs sessionStorage?
5. How do you get the user's location in JavaScript?
6. What is the purpose of the History API?
7. How do setTimeout and setInterval differ?
8. How do you check for API support in the browser?
9. What are some other useful Browser APIs?
10. How do you handle errors when using Fetch API?

12 Key Takeaways

- ✓ Browser APIs make JavaScript powerful and interactive.
- ✓ They allow access to the DOM, browser features, network, storage, and more.
- ✓ Always handle permissions and errors.
- ✓ Use the right API for the right use case.
- ✓ Learn and explore modern web APIs to build real-world projects.

Karyvio
LEARN | BUILD | GROW | GET HIRED

Asynchronous JavaScript (Callbacks, Promises & Async/Await)

Asynchronous JavaScript allows us to handle time-consuming tasks (like API calls, reading files, timers, etc.) without blocking the main thread. It helps in building fast and responsive applications.

"Asynchronous code doesn't wait, it keeps your application alive."

1 What is Asynchronous?

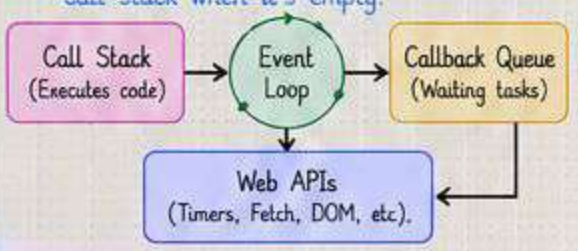
- Asynchronous code runs in the background without blocking the main thread.
- It allows other code to run while waiting for a task to complete.
- Common in web APIs, timers, file operations, network requests, etc.
- Helps improve performance and user experience.

2 Synchronous vs Asynchronous

Synchronous	Asynchronous
Executes code line by line	Does not block the main thread
Waits for each task to complete	Can run multiple tasks in parallel
Slower for I/O tasks	Faster and more efficient
Example: normal function	Example: setTimeout, fetch

3 The Event Loop (High Level)

- JavaScript uses an event loop to handle asynchronous operations.
- Tasks are offloaded to Web APIs (like timers, network requests, etc.).
- Once completed, callbacks are pushed to the callback queue.
- The event loop moves them to the call stack when it's empty.



4 Callbacks

- A callback is a function passed as an argument to another function.
- It is executed after a task is completed.
- Callbacks can lead to callback hell when nested multiple times.

Example :-

```
function fetchData(callback) {
  setTimeout(() => {
    console.log("Data loaded");
    callback();
  }, 2000);
}

fetchData(() => {
  console.log("Callback executed");
});
```

Note :-

Callbacks are useful but can make code hard to read when nested deeply (callback hell).

5 Callback Hell (Problem)

- Nested callbacks make the code hard to read and maintain.

Example :-

```
setTimeout(() => {
  console.log("Step 1");
    setTimeout(() => {
      console.log("Step 2");
        setTimeout(() => {
          console.log("Step 3");
            setTimeout(() => {
              console.log("Step 4");
            }, 1000);
        }, 1000);
    }, 1000);
}, 1000);
```

Tip :-

- Use Promises or async/await to avoid callback hell.

6 Promises

- A Promise is an object that represents the eventual completion (or failure) of an asynchronous operation.
- A Promise can be in one of three states:
  - Pending – initial state
  - Fulfilled – operation completed successfully
  - Rejected – operation failed

Example :-

```
const promise = new Promise((resolve, reject) => {
  setTimeout(() => {
    const success = true;
    if (success) {
      resolve("Operation successful!");
    } else {
      reject("Operation failed!");
    }
  }, 2000);
});

promise
  .then((result) => console.log(result))
  .catch((error) => console.log(error));
```

Note :-

A Promise cannot be reset. Once it is fulfilled or rejected, its state is final.

7 Promise Chaining

- We can chain multiple .then() calls to perform a sequence of asynchronous tasks.
- Each .then() returns a new Promise.

Example :-

```
fetch("https://jsonplaceholder.typicode.com/posts/1")
  .then(response => response.json())
  .then(data => {
    console.log("Post ID:", data.id);
    return fetch("https://jsonplaceholder.typicode.com/users/1");
  })
  .then(response => response.json())
  .then(user => {
    console.log("User Name:", user.name);
  })
  .catch(error => console.log("Error:", error));
```

8 Async / Await

- async/await is a modern way to work with Promises.
- It makes asynchronous code look and behave like synchronous code.
- It improves readability and is easier to maintain.

Example :-

```
async function getData() {
  try {
    const response = await fetch(
      "https://jsonplaceholder.typicode.com/posts/1"
    );
    const data = await response.json();
    console.log("Post:", data);
  } catch (error) {
    console.log("Error:", error);
  }
}
getData();
```

Tip :-

- Always use try...catch with async/await to handle errors.

9 Real-world Example

Fetching data from an API using async/await and displaying it:

Example :-

```
async function fetchUser() {
  try {
    const response = await fetch(
      "https://jsonplaceholder.typicode.com/users/1"
    const user = await response.json();
    console.log("User:", user);
    document.getElementById("output").innerText =
    `Name: ${user.name\nEmail: ${user.email}`;
  } catch (error) {
    console.log("Failed to fetch user:", error);
  }
}
fetchUser();
```

10 Key Takeaways

- ✓ Use asynchronous JavaScript to keep your app responsive.
- ✓ Understand the event loop and how async code is executed.
- ✓ Callbacks are useful but can lead to callback hell.
- ✓ Promises provide a cleaner way to handle async code.
- ✓ Use async/await for better readability.
- ✓ Always handle errors using .catch() or try...catch.
- ✓ Asynchronous programming is essential for real-world web development (APIs, databases, file handling, etc.).

Common Interview Questions

1. What is asynchronous JavaScript?
2. What is the difference between synchronous and asynchronous code?
3. Explain the event loop in JavaScript.
4. What is a callback? Why can it lead to callback hell?
5. What is a Promise? What are its states?
6. How does async/await work?
7. How do you handle errors in async/await?
8. Give a real-world example of using Promises or async/await.
9. What is the difference between .then() and await?
10. Why is asynchronous programming important in web development?

Karyvio
LEARN | BUILD | GROW | GET HIRED

JavaScript DOM (Document Object Model)

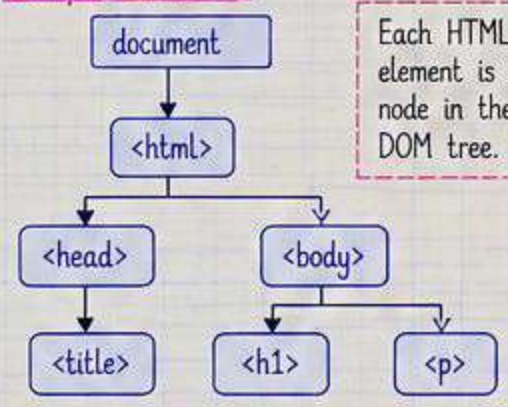
Understand how to interact with HTML using JavaScript

DOM bridges JavaScript and HTML. It allows us to read, modify and create content dynamically on a web page.

1 What is DOM ?

- DOM (Document Object Model) is a programming interface for HTML documents.
- It represents the HTML structure as a tree of objects (nodes).
- We can use JavaScript to access, modify, add or delete these nodes.

Example Structure:



Each HTML element is a node in the DOM tree.

2 Selecting Elements

- We can select elements using different methods provided by the DOM.

Example :-

```
// Select by ID
document.getElementById("title");
// Select by class name
document.getElementsByClassName("item");
// Select by tag name
document.getElementsByTagName("p");
// Select using CSS selector
document.querySelector(".item");
document.querySelectorAll(".item");
```

Note :-

querySelector() returns the first match.
querySelectorAll() returns a NodeList of all matches.

3 Changing Content

- We can change the text, HTML or attributes of an element.

Example :-

```
let heading = document.getElementById("title");
// Change text content
heading.textContent = "Welcome to Karyvio!";
// Change HTML content
heading.innerHTML = "<span>Hello JS</span>";
// Change an attribute
let img = document.querySelector("img");
img.src = "logo.png";
img.alt = "Karyvio Logo";
```

textContent is safer (prevents XSS).
Use innerHTML only when you need HTML tags.

4 Creating and Adding Elements

- We can create new elements and add them to the DOM.

Example :-

```
let li = document.createElement("li");
li.textContent = "New Item";

let ul = document.getElementById("list");
ul.appendChild(li);     // add at end
// Add at beginning
ul.prepend(li);
// Insert before a specific element
ul.insertBefore(li, ul.children[0]);
```

5 Removing Elements

- We can remove elements from the DOM using remove() method.

Example :-

```
let item = document.getElementById("item");
item.remove();

// Or using parent
let parent = document.getElementById("list");
parent.removeChild(item);
```

Tip :-

- remove() is simpler and modern.
- Always check if the element exists before removing.

6 Getting & Setting Attributes

- We can get or set HTML attributes like src, href, class, id, etc.

Example :-

```
let link = document.querySelector("a");

// Get attribute
console.log(link.getAttribute("href"));

// Set attribute
link.setAttribute("href", "https://karyvio.com");

// Check if attribute exists
console.log(link.hasAttribute("href"));   // true
```

7 Class Manipulation

- We can add, remove, toggle or check classes using classList.

Example :-

```
let box = document.getElementById("box");

box.classList.add("active");      // add class
box.classList.remove("active");   // remove class
box.classList.toggle("active");   // toggle class
console.log(box.classList.contains("active"));
```

8 Event Handling

- We can respond to user actions like clicks, input, hover, etc.

Example :-

```
let btn = document.getElementById("btn");
btn.addEventListener("click", function() {
    alert("Button clicked!");
});
// With arrow function
btn.addEventListener("mouseover", () => {
    console.log("Mouse over button");
});
```

9 Traversing the DOM

- We can navigate between parent, child and sibling elements.

Example :-

```
let item = document.getElementById("item");
console.log(item.parentElement);  // parent
console.log(item.children);       // children
console.log(item.firstElementChild);
console.log(item.lastElementChild);
console.log(item.nextElementSibling);
console.log(item.previousElementSibling);
```

10 DOM Practice Example

- Let's create a simple to-do list using DOM.

Example :-

```
let input = document.getElementById("task");
let list = document.getElementById("list");
function addTask() {
  let li = document.createElement("li");
  li.textContent = input.value;
  list.appendChild(li);
  input.value = "";
}
```

Try This :-

- Create a to-do list with add and delete.
- Change the page theme (light/dark).
- Create a simple image gallery.
- Build a counter app.

11 Best Practices

- ✓ Use querySelector instead of older methods when possible.
- ✓ Cache DOM elements in variables.
- ✓ Use textContent to prevent XSS.
- ✓ Avoid too many DOM manipulations (performance).
- ✓ Use event delegation for dynamic elements.
- ✓ Keep your code clean and modular.
- ✓ Use meaningful class and id names.
- ✓ Test your code in different browsers.

12 Common DOM Events

Event	Description
click	User clicks on an element
dblclick	User double clicks
mouseover	Mouse hovers over element
mouseout	Mouse leaves element
input	User types in input field
change	Value is changed
submit	Form is submitted
keydown	Key is pressed
keyup	Key is released
load	Page is fully loaded

JavaScript Events (In-Depth)

Learn how to handle user interactions and browser events effectively

Events make web pages interactive. They allow us to respond to user actions like clicks, key presses, form submissions, and more !!

1 What are Events ?

- Events are actions or occurrences that happen in the browser.
- They can be triggered by the user or by the browser itself.
- We can listen to these events and run code when they occur.

Examples of Events:

- click (user clicks an element)
- keydown (user presses a key)
- submit (form is submitted)
- load (page finishes loading)
- mouseover (mouse enters an element)
- resize (window is resized)
- scroll (user scrolls the page)

Events bring web pages to life!

2 Adding Event Listeners

- We use addEventListener() to listen for an event on an element.
- It allows multiple event listeners.

Example :-

```
const button = document.getElementById("btn");
button.addEventListener("click", function() {
  console.log("Button clicked!");
});
button.addEventListener("mouseover", () => {
  console.log("Mouse over button!");
});
```

Note :- We can add multiple event listeners to the same element for different events.

3 Inline Event Handling

- We can also handle events directly in HTML using attributes.
- Not recommended for large projects, but useful for quick tests.

Example :-

```
<button onclick="alert('Hello!')">
  Click Me
</button>
```

Use inline events only for simple cases.
For complex projects, prefer addEventListener().

4 Event Object

- When an event occurs, an event object is passed to the event handler.
- It contains useful information about the event.

Example :-

```
button.addEventListener("click", function(e) {
  console.log(e);          // event object
  console.log(e.type);     // "click"
  console.log(e.target);   // element clicked
  console.log(e.clientX, e.clientY); // mouse position
});
```

5 Common Mouse Events

- Mouse events are triggered by mouse actions.
- Useful for creating interactive UI.

Event	Description
click	When an element is clicked
dblclick	When double-clicked
mousedown	When mouse button is pressed
mouseup	When mouse button is released
mousemve	When mouse moves over element
mouseover	When mouse enters element
mouseout	When mouse leaves element
contextmenu	When right-clicked

Tip :- Try logging different mouse events to understand how they work.

6 Common Keyboard Events

- Keyboard events are triggered when a key is pressed, released, or held down.

Event	Description
keydown	When a key is pressed down
keyup	When a key is released
keypress	(Depercated) Use keydown/keyup

Example :-

```
document.addEventListener("keydown", (e) => {
  console.log("Key:", e.key);
  console.log("Key Code:", e.code);
});
```

e.key gives the actual key (e.g. "a")
e.code gives the physical key (e.g. "KeyA").

7 Form Events

- Form events help us handle user input and form submission.
- Common form events: submit, input, change, focus, blur.

Example :-

```
const form = document.getElementById("myForm");
form.addEventListener("submit", (e) => {
  e.preventDefault();  // prevent page reload
  console.log("Form submitted!");
});
const input = document.getElementById("name");
input.addEventListener("input", () => {
  console.log("Input value:", input.value);
});
```

Note :- Use preventDefault() to stop the default form submission behavior.

8 Event Delegation

- We can use event delegation to handle events on multiple elements using a single event listener.
- It works because of event bubbling.

Example :-

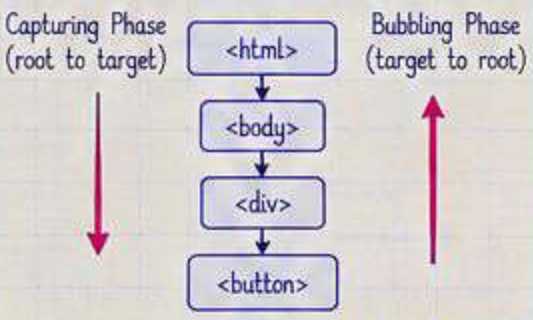
```
const list = document.getElementById("list");
list.addEventListener("click", (e) => {
  if (e.target.tagName === "LI") {
    console.log("You clicked:", e.target.textContent);
  }
});
```

Why use it ?

- Works for dynamically added elements
- Better performance
- Cleaner code

9 Event Bubbling & Capturing

- When an event occurs, it can propagate through the DOM tree.
- Two phases: Capturing (top → bottom) and Bubbling (bottom → top).



By default, event listeners run in the bubbling phase.

We can set the third parameter to true to use capturing: addEventListener("click", handler, true)

10 Removing Event Listeners

- We can remove an event listener using removeEventListener().

Example :-

```
function handleClick() {
  console.log("Button clicked!");
}
button.addEventListener("click", handleClick);
button.removeEventListener("click", handleClick);
```

Note :- The function reference must be the same when removing the listener.

11 Useful Tips

- ✓ Use descriptive function names.
- ✓ Avoid inline event handlers.
- ✓ Use event delegation for dynamic elements.
- ✓ Remove event listeners when no longer needed.
- ✓ Be careful with event propagation.
- ✓ Use preventDefault() when necessary.
- ✓ Test events on different devices and browsers.
- ✓ Combine events with DOM manipulation for interactive experiences.

12 Practice Tasks

1. Create a button that changes color on click.
2. Create a form and handle its submit event.
3. Log mouse position on mousemove.
4. Use event delegation to handle clicks on a list of items.
5. Create a simple key logger (show pressed keys).
6. Explore more events (scroll, resize, focus).

"Events turn static pages into dynamic and interactive experiences."

Karyvio
LEARN | BUILD | GROW | GET HIRED

JavaScript Asynchronous Programming

Understand and master asynchronous JavaScript with callbacks, promises, async/await and the event loop.

Asynchronous JavaScript helps us perform time-consuming tasks without blocking the main thread. It keeps our applications fast and responsive !!

1 What is Asynchronous JavaScript?

- Asynchronous code allows us to perform tasks that take time (like fetching data, reading files, etc.) without blocking the execution of other code.
- It uses the event loop, callback queue and Web APIs provided by the browser (or Node.js).
- Helps create non-blocking, responsive apps.

Examples of Async Tasks :-

- API calls (fetch data from server)
- Reading files
- Timers (setTimeout, setInterval)
- User interactions (click, input)
- Database requests (in backend)

Don't Block the UI!

2 Synchronous vs Asynchronous

- Synchronous code runs line by line and blocks the next execution.
- Asynchronous code allows other code to run while waiting for a task to complete.

Synchronous Example :-

```
console.log("Start");          // Output:
console.log("Middle");         // Start
console.log("End");            // Middle
                               // End
```

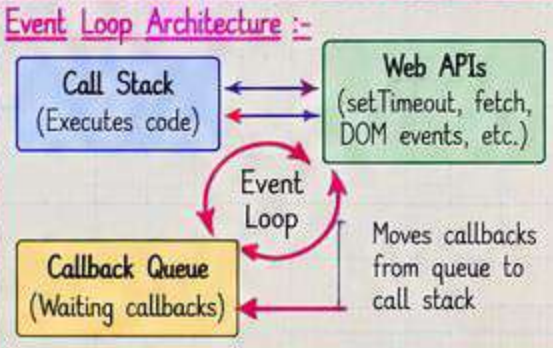
Asynchronous Example :-

```
console.log("Start");          // Output:
setTimeout(() => {             // Start
  console.log("Middle");       // End
}, 2000);                      // Middle (after 2s)
console.log("End");
```

3 The Event Loop

- The event loop is responsible for handling asynchronous operations in JavaScript.
- It checks the call stack, Web APIs and callback queue to execute code.
- Even though JavaScript is single-threaded, it can handle asynchronous tasks using the event loop.

Event Loop Architecture :-



4 setTimeout() and setInterval()

- setTimeout() executes a function after a specified delay (in milliseconds).
- setInterval() repeatedly executes a function at a fixed interval.

Example :-

```
setTimeout(() => {
  console.log("This runs after 2 seconds");
}, 2000);

let count = 0;
const id = setInterval(() => {
  console.log("Count:", ++count);
  if (count === 3) clearInterval(id);
}, 1000);  // runs every 1 second
```

5 Callbacks

- A callback is a function passed as an argument to another function.
- It is executed after the completion of an asynchronous task.
- Callbacks can lead to deeply nested code (callback hell) if not handled properly.

Example :-

```
function fetchData(callback) {
  setTimeout(() => {
    callback("Data loaded!");
  }, 2000);
}

fetchData((data) => {
  console.log(data);
});
```

Callback Hell :-

```
step1(() => {
  step2(() => {
    step3(() => {
      step4(() => {
        // ....
      });
    });
  });
});
```

Hard to read and maintain!

6 Promises

- A Promise is a cleaner way to handle asynchronous operations.
- It can be in one of three states: Pending, Fulfilled (Resolved) or Rejected.
- Promises help avoid callback hell.

Example :-

```
const promise = new Promise((resolve, reject) => {
  let success = true;
  if (success) {
    resolve("Operation successful!");
  } else {
    reject("Something went wrong!");
  }
});

promise
  .then((message) => console.log(message))
  .catch((error) => console.log(error));
```

Promise States:

- Pending
- Fulfilled
- Rejected

7 Async / Await

- async/await is a modern and cleaner way to work with promises.
- It makes asynchronous code look and behave more like synchronous code.
- It is easier to read and maintain.

Example :-

```
function fetchData() {
  return new Promise((resolve) => {
    setTimeout(() => resolve("Data from server"),
      2000);
  });
}
async function getData() {
  console.log("Fetching(data...");
  const data = await fetchData();
  console.log(data);
}
getData();
```

Note :-
We can use try...catch with async/await to handle errors properly.

8 Error Handling

- We can handle errors in async code using .catch() (for promises) or try...catch (for async/await).
- Always handle errors to prevent unexpected crashes.

Promise Error Handling :-

```
fetchData()
  .then((data) => console.log(data))
  .catch((error) => console.log("Error:", error));
```

Async/Await Error Handling :-

```
async function getData() {
  try {
    const data = await fetchData();
    console.log(data);
  } catch (error) {
    console.log("Error:", error);
  }
}
```

9 Real-World Example (Fetch API)

- We can use fetch() to make API requests and handle the response using promises or async/await.
- It returns a promise that resolves to the Response object.

Example (Using Async/Await) :-

```
async function getUsers() {
  try {
    const response = await fetch(
      "https://jsonplaceholder.typicode.com/users"
    );
    const users = await response.json();
    console.log(users);
  } catch (error) {
    console.log("Failed to fetch users:", error);
  }
}

getUsers();
```

10 Best Practices

- ✓ Use async/await for cleaner code.
- ✓ Always handle errors.
- ✓ Avoid callback hell.
- ✓ Use meaningful function and variable names.
- ✓ Don't block the main thread.
- ✓ Keep your async code modular.
- ✓ Use Promise.all() for multiple requests.
- ✓ Add loading states in UI.
- ✓ Test error cases properly.

11 Common Interview Questions

1. What is a callback in JavaScript?
2. Explain the event loop.
3. What are promises? How do they work?
4. What is the difference between async/await and promises?
5. How does setTimeout() work?
6. What happens when a promise is rejected?
7. How can you handle multiple API requests?
8. What is Promise.all()?

12 Key Takeaways

- ✓ Asynchronous JavaScript is essential for building modern web apps.
- ✓ Use callbacks (basic), promises (better), and async/await (best) based on the need.
- ✓ Understand the event loop.
- ✓ Always handle errors.
- ✓ Practice by building real projects (e.g., fetch data from an API).
- ✓ Good understanding of async code helps in interviews and real-world development.

Karyvio
LEARN | BUILD | GROW | GET HIRED

JavaScript ES6+ Features (Modern JS)

Explore modern JavaScript features that make your code cleaner, shorter and more powerful.

Modern JavaScript helps you write less code, do more, and build real-world applications faster!

1 let, const and Block Scope

- let and const are block scoped.
- let can be reassigned, const cannot.
- They solve issues with var (function scoped).

Example :-

```
let name = "Sovit";
name = "Karyvio";     // allowed

const pi = 3.14;
// pi = 3.15;     // Error: Assignment
                  // to constant variable
```

Tip :-

- ✔ Use const by default.
- ✔ Use let only when you need to reassign.
- ✔ Avoid var in modern JavaScript.

2 Arrow Functions

- Shorter syntax for writing functions.
- No own this (inherits from parent scope).
- Great for callbacks and functional code.

Example :-

```
const add = (a, b) => a + b;
console.log(add(5, 3));     // 8

const square = n => n * n;
console.log(square(4));     // 16
```

Note :-

- Arrow functions do not have their own this, arguments, super or new.target.
- Not suitable for methods in objects (where you need this).

3 Template Literals

- Use backticks (`) instead of quotes.
- Allow multi-line strings and interpolation.

Example :-

```
let name = "Karyvio";
let msg = `Hello ${name}!
Welcome to Karyvio.
Learn | Build | Grow | Get Hired.`;
console.log(msg);
```

Output :-

```
Hello Karyvio!
Welcome to Karyvio.
Learn | Build | Grow | Get Hired.
```

4 Default Parameters

- Provide default values for function parameters.
- Helps avoid undefined values.

Example :-

```
function greet(name = "Guest") {
  console.log("Hello, ${name}!");
}

greet();          // Hello, Guest!
greet("Sovit");   // Hello, Sovit!
```

5 Rest Parameters (...args)

- Allows a function to accept any number of arguments.
- Useful when you don't know how many arguments will be passed.

Example :-

```
function sum(...numbers) {
  return numbers.reduce((acc, n) => acc + n, 0);
}
console.log(sum(1, 2, 3, 4));     // 10
console.log(sum(10, 20));         // 30
```

Note :- Rest parameters collect all remaining arguments into an array.

6 Spread Operator (...)

- Expands elements of an array or object.
- Useful for copying, merging and passing elements.

Example :-

```
let arr1 = [1, 2, 3];
let arr2 = [...arr1, 4, 5];
console.log(arr2);   // [1, 2, 3, 4, 5]

let obj1 = { a: 1, b: 2 };
let obj2 = { ...obj1, c: 3};
console.log(obj2);  // {a: 1, b: 2, c: 3}
```

7 Destructuring

- Extract values from arrays or objects into variables.
- Makes your code cleaner and readable.

Example :-

```
// Array destructuring
const [x, y] = [10, 20];
console.log(x, y);    // 10 20

// Object destructuring
const { name, age } = { name: "Sovit", age: 22 };
console.log(name);    // Sovit
console.log(age);     // 22
```

8 Object Shorthand

- Shorter syntax for object properties.
- Useful when variable names and property names are the same.

Example :-

```
let name = "Sovit";
let age = 22;

// Old way
let user1 = { name: name, age: age };

// Shorthand way
let user2 = { name, age };
console.log(user2);  // { name: "Sovit", age: 22 }
```

9 Enhanced Object Literals

- You can now use methods, computed property names and concise syntax.

Example :-

```
let key = "course";
let obj = {
  key,                    // computed property
  getInfo() {             // method shorthand
    return "JavaScript";
  },
 ["full" + "Name"]: "Sovit Lenka"  // computed key
};
 console.log(obj);
 console.log(obj.getInfo());  // JavaScript
```

10 Classes (ES6)

- A cleaner syntax for creating objects and inheritance.
- Introduces class and constructor.

Example :-

```
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
  greet() {
    console.log("Hello, I'm ${this.name}");
  }
}
const p = new Person("Sovit", 22);
p.greet();    // Hello, I'm Sovit
```

11 Modules (import / export)

- Helps to split code into multiple files.
- Use export to share and import to use.

math.js (export)

```
export const add = (a, b) => a + b;
export const PI = 3.14;
```

app.js (import)

```
import { add, PI } from "./math.js";
console.log(add(5, 3)); // 8
console.log(PI);        // 3.14
```

Tip :-

- Use modules to keep your code organized and maintainable.

12 Summary & Best Practices

- ✔ Use const by default.
- ✔ Use arrow functions for cleaner code.
- ✔ Leverage template literals for readability.
- ✔ Use destructuring and spread for cleaner data handling.
- ✔ Write modular code using import/export.
- ✔ Follow modern ES6+ features in real projects.
- ✔ Keep your code simple, readable and maintainable.

"Modern JavaScript is not just about writing less code, but writing better code."

Karyvio
LEARN | BUILD | GROW | GET HIRED

JavaScript Error Handling (In-Depth)

Learn how to handle errors gracefully and write more reliable JavaScript code.

"Good error handling turns a crash into a better user experience."

1 What is an Error?

- An error is an unexpected problem that occurs during program execution.
- It stops the normal flow of code.
- JavaScript provides error objects to describe what went wrong.

Example :-

```
console.log(x);
    // ReferenceError
    x is not defined
```

2 Types of Errors

- ReferenceError – using an undeclared variable.
- TypeError – using a value in an invalid way.
- SyntaxError – invalid syntax in code.
- RangeError – value out of allowed range.
- URIError – invalid URI.
- Custom Errors – user-defined errors.

Common Error Types :-

- ReferenceError
- TypeError
- SyntaxError
- RangeError
- URIError
- EvalError (rarely used)

} Built-in Error Types in JS

3 try...catch...finally

- Used to catch and handle errors without stopping the entire program.
- finally always runs (cleanup code).
- Useful for graceful error handling.

Example :-

```
try {
   let result = riskyOperation();
   console.log(result);
} catch (error) {
   console.log("Error:", error.message);
} finally {
   console.log("This always runs");
}
```

Keeps your app stable

4 The Error Object

- When an error occurs, JavaScript creates an Error object.
- It contains useful information like name, message, and stack trace.

Example :-

```
try {
   throw new Error("Something went wrong");
} catch (error) {
   console.log(error.name);    // Error
   console.log(error.message); // Something went wrong
   console.log(error.stack);   // stack trace
}
```

5 Throwing Errors

- We can manually throw errors using the throw keyword.
- Useful for custom validation and handling invalid input.

Example :-

```
function divide(a, b) {
   if (b === 0) {
      throw new Error("Cannot divide
      by zero!");
   }
   return a / b;
}
try {
   console.log(divide(10, 2));  // 5
   console.log(divide(10, 0));  // Error
} catch (e) {
   console.log(e.message);
}
```

6 Custom Error Classes

- We can create our own error types by extending the Error class.
- Useful for application-specific errors.

Example :-

```
class ValidationError extends Error {
   constructor(message) {
      super(message);
      this.name = "ValidationError";
   }
}
function validateAge(age) {
   if (age < 18) {
      throw new ValidationError(
      "Age must be 18 or above!")
   }
   return "Valid age";
}
try {
   validateAge(16);
} catch (e) {
   console.log(e.name);    // ValidationError
   console.log(e.message); // Age must be 18 or above!
}
```

7 Promise Error Handling

- Use .catch() to handle errors in promises.
- Errors inside .then() are also caught.

Example :-

```
fetch("https://jsonplaceholder.typicode.com/invalid")
   .then(response => {
      if (!response.ok) {
         throw new Error("Network error!");
      }
      return response.json();
   })
   .then(data => console.log(data))
   .catch(error => console.log("Fetch Error:", error.message));
```

8 Async/Await Error Handling

- Use try...catch with async/await.
- Makes async code easier to read and handle errors.

Example :-

```
async function getData() {
   try {
      const res = await fetch(
        "https://jsonplaceholder.typicode.com/posts/1"
      );
      if (!res.ok) throw new Error("Failed to fetch");
      const data = await res.json();
      console.log(data);
   } catch (error) {
      console.log("Async Error:", error.message);
} }
getData();
```

9 Common Real-World Scenarios

- Handling API failures
- Validating user input
- Handling file operations
- Managing database errors (in backend)
- Showing user-friendly error messages

Example (User Input Validation) :-

```
function register(username) {
   if (!username) {
      throw new Error("Username is required!");
   }
   return "User registered: " + username;
}
try {
   console.log(registeer(""));
} catch (e) {
   console.log(e.message); // Username is required!
}
```

10 Best Practices

- ✓ Always handle errors (don't ignore them).
- ✓ Use meaningful error messages.
- ✓ Use specific error types.
- ✓ Avoid empty catch blocks.
- ✓ Log errors for debugging.
- ✓ Show user-friendly messages (don't expose details to end users).
- ✓ Use finally for cleanup.
- ✓ Handle errors in async code properly.

11 Common Mistakes

- ✗ Ignoring errors completely.
- ✗ Using generic error messages.
- ✗ Not handling promise rejections.
- ✗ Leaving catch blocks empty.
- ✗ Exposing sensitive information to users.
- ✗ Not validating user input.
- ✗ Forgetting to handle async errors.
- ✗ Using try...catch for control flow (instead of proper checks).

12 Key Takeaways

- ✓ Errors are a natural part of coding.
- ✓ Handle them gracefully.
- ✓ Use try...catch...finally.
- ✓ Create custom errors when needed.
- ✓ Always validate input.
- ✓ Use .catch() or try...catch for async code.
- ✓ Keep error messages clear and helpful.

Karyvio
LEARN | BUILD | GROW | GET HIRED

this, call(), apply(), bind() in JavaScript

"this is not about where a function is defined, but how it is called."

In JavaScript, `this` refers to the context in which a function is executed. The methods call(), apply(), and bind() allow us to explicitly set the value of `this` for a function. They are powerful tools for function borrowing, method sharing, and controlling context.

1 What is `this`?

- `this` refers to the object that is currently executing the function.
- Its value depends on how the function is called (not where it is defined).
- It behaves differently in different contexts (global, object, function, arrow function, etc.).
- Understanding `this` is crucial for using call(), apply(), and bind().

2 Values of `this` in Different Contexts

Context	Value of `this`
Global context	window (in browser) or undefined (in strict mode)
Object method	the object itself
Function (regular)	window or undefined (in strict mode)
Arrow function	inherits `this` from its lexical scope
Event handler	the element that triggered the event
Constructor function	the new instance (using `new`)
call(), apply(), bind()	the value explicitly provided
Class method	the instance (similar to object method)

3 Why Use call(), apply(), bind()?

- To explicitly set the value of `this`.
- To borrow functions from one object and use them in another.
- To avoid code repetition.
- Useful in inheritance, functional programming, and working with array-like objects (e.g., arguments, NodeLists).
- Helps in creating reusable and flexible code.

Tip :-

- Use call() when you have individual arguments.
- Use apply() when you have an array of arguments.
- Use bind() when you want a new function with a fixed `this`.

4 Using call()

- The call() method calls a function with a given `this` value and arguments passed individually.
- Syntax: function.call(thisArg, arg1, arg2, ...);

Example :-

```
function greet(city, country) {
  console.log(`Hello, I am ${this.name} from ${city}, ${country}`);
}

const person1 = { name: "Sovit" };

greet.call(person1, "Bhubaneswar", "India");
// Output: Hello, I am Sovit from Bhubaneswar, India
```

5 Using apply()

- The apply() method is similar to call(), but it takes arguments as an array (or array-like object).
- Syntax: function.apply(thisArg, [argsArray]);

Example :-

```
function greet(city, country) {
  console.log(`Hello, I am ${this.name} from ${city}, ${country}`);
}
const person2 = { name: "Smruti" };
greet.apply(person2, ["Cuttack", "India"]);
// Output: Hello, I am Smruti from
// Cuttack, India
```

Note :-

- call() and apply() both immediately invoke the function.

6 Using bind()

- The bind() method returns a new function with `this` permanently set to the provided value.
- It does not immediately invoke the function.
- Syntax: function.bind(thisArg, arg1, arg2, ...);

Example :-

```
function greet(city, country) {
  console.log(`Hello, I am ${this.name} from ${city}, ${country}`);
}
const person3 = { name: "Karyvio" };
const newGreet = greet.bind(person3, "Puri");
newGreet("India");
// Output: Hello, I am Karyvio from Puri, India
```

Note :-

- bind() is useful when you want to reuse the function later with the same `this`.

7 Function Borrowing (Real Example)

- We can use call() or apply() to borrow methods from one object and use them in another.

Example :-

```
const person = { name: "Sovit", age: 22 };
const student = { name: "Rohan", age: 20 };

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

introduce.call(person);  // Hi, I am Sovit and I am 22...
introduce.call(student); // Hi, I am Rohan and I am 20...
```

8 Working with Array-like Objects

- We can use call() or apply() to use array methods on array-like objects (like arguments, NodeList, etc.).

Example :-

```
function sum() {
  return Array.prototype.reduce.call(
    arguments,
    (acc, curr) => acc + curr,
    0
  );
}
console.log(sum(1, 2, 3, 4));  // 10
```

Note :-

- `arguments` is an array-like object, not a real array. We used call() to borrow the reduce() method.

9 call() vs apply() vs bind()

Method	Arguments	Invokes Immediately?	Returns
call()	Individual arguments	Yes	Result of the function
apply()	Array of arguments	Yes	Result of the function
bind()	Individual arguments	No	A new function

10 Common Use Cases

- ✓ Function borrowing.
- ✓ Working with array-like objects.
- ✓ Setting a specific context for event handlers.
- ✓ Partial application (using bind).
- ✓ Maintaining `this` in callbacks.
- ✓ Useful in OOP, inheritance, and utility functions.

11 Key Takeaways

- ✓ `this` depends on how a function is called.
- ✓ call() passes arguments individually.
- ✓ apply() passes arguments as an array.
- ✓ bind() returns a new function with fixed `this`.
- ✓ Use them to write flexible, reusable, and powerful code.
- ✓ They are widely used in real-world JavaScript development.

Common Interview Questions

1. What is `this` in JavaScript?
2. What is the difference between call(), apply() and bind()?
3. When would you use bind() instead of call()?
4. How can you borrow a method from one object to another?
5. What will be the output of a function using call() / apply() / bind()?
6. Can you use bind() to set `this` for an arrow function?
7. How would you use call() to find the maximum number in an array?
8. What are some real-world use cases of bind()?

Closures & Higher-Order Functions

Closures and Higher-Order Functions are powerful concepts in JavaScript. They help us write more flexible, reusable and maintainable code.

"Functions are first-class citizens in JavaScript, and closures make them remember."

1 What is a Closure?

- A closure is a function that remembers the variables from its outer (lexical) scope even after the outer function has finished execution.
- It has access to its own scope, the outer scope and global scope.
- Closures are created every time a function is created, at function creation time.

Example :-

```
function outer() {
  let count = 0;
  return function inner() {
    count++;
    return count;
  };
}
const counter = outer();
console.log(counter());  // 1
console.log(counter());  // 2
console.log(counter());  // 3
```

The inner function "remembers" the count variable even though outer() has already finished execution.

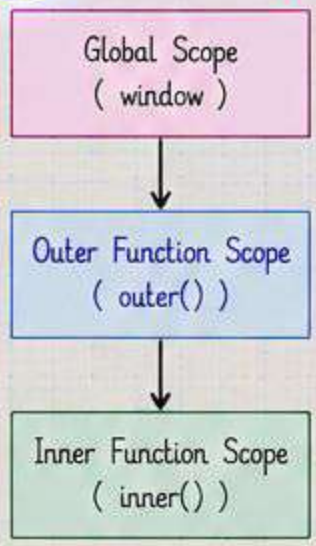
2 How Closures Work?

1. A function is created.
2. It forms a closure with the variables in its outer scope.
3. Even after the outer function finishes, the inner function still has access to those variables.
4. The variables are not garbage collected because they are still being used.

Real-world Analogy :-

Think of a closure like a backpack 🎒 The inner function carries the variables from the outer function with it, wherever it goes.

3 Closure Scope Chain



4 Practical Uses of Closures

- ✔ Data privacy (encapsulation)
- ✔ Maintaining state in functions
- ✔ Function factories
- ✔ Event handling
- ✔ Module patterns
- ✔ Callbacks and asynchronous code

5 Example: Private Variable

```
function createUser(name) {
  let score = 0;  // private variable
  return {
    getName: () => name,
    getScore: () => score,
    increaseScore: () => score++
  };
}

const user = createUser("Sovit");
console.log(user.getName());  // Sovit
console.log(user.getScore());  // 0
user.increaseScore();
console.log(user.getScore());  // 1
```

The score variable cannot be accessed directly from outside. This is data privacy using closures.

6 What are Higher-Order Functions?

- A higher-order function (HOF) is a function that either:
- Takes one or more functions as arguments, or
- Returns a function as its result.
- In JavaScript, functions are first-class citizens, so they can be passed around like variables.

"A function that works with another function."

7 Examples of Higher-Order Functions

a) Function as an Argument

```
function greet(name) {
  console.log("Hello " + name);
}

function processUser(name, callback) {
  console.log("Processing...");
  callback(name);  // calling the function passed
}

processUser("Sovit", greet);
// Processing...
// Hello Sovit
```

b) Function Returning a Function

```
function multiplier(factor) {
  return function(number) {
    return number * factor;
  };
}

const double = multiplier(2);
const triple = multiplier(3);

console.log(double(5));  // 10
console.log(triple(5));  // 15
```

8 Built-in Higher-Order Functions

Method	Description
map()	Transforms each element
filter()	Filters elements based on condition
reduce()	Reduces array to a single value
forEach()	Executes a function for each element
sort()	Sorts elements using a compare function
some()	Checks if at least one element matches
every()	Checks if all elements match
find()	Returns the first matching element
findIndex()	Returns the index of the first match

9 Combining Closures with HOFs

```
function createMultiplier(factor) {
  return function(num) {
    return num * factor;
  };
}

const multiplyBy5 = createMultiplier(5);
const numbers = [1, 2, 3];
const result = numbers.map(multiplyBy5);

console.log(result);  // [5, 10, 15]
```

Here we use a closure (multiplyBy5) inside a higher-order function (map).

10 Key Takeaways

- ✔ A closure remembers variables from its outer scope.
- ✔ Closures are created at function creation time.
- ✔ Higher-order functions take or return functions.
- ✔ Functions enable powerful patterns like modular code, callbacks, and functional programming.
- ✔ Closures help in data privacy and maintaining state.
- ✔ Built-in HOFs (map, filter, reduce, etc.) are widely used.
- ✔ Understanding closures and HOFs is important for interviews and real-world JavaScript development.

Karyvio
LEARN | BUILD | GROW | GET HIRED

Classes & Object-Oriented JavaScript

JavaScript is a prototype-based language, but with ES6 we got a cleaner syntax using classes to write object-oriented code. Classes are basically syntactic sugar over prototypes.

"Classes in JavaScript make it easier to write and understand object-oriented code."

1 What is a Class?

- A class is a blueprint for creating objects.
- It allows you to define properties and methods in a structured way.
- Introduced in ES6 (ES2015).
- Classes are not hoisted.
- Internally, JavaScript still uses prototypes.

2 Creating a Class

Example :-

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

  greet() {
    console.log(`Hi, I'm ${this.name}`);
  }
}

const p1 = new Person("Sovit", 22);
p1.greet();    // Hi, I'm Sovit
```

3 Constructor Method

- The constructor() method is called automatically when a new object is created.
- It is used to initialize properties.
- You can only have one constructor per class.

Note :-

- You don't need to manually return anything from the constructor.

4 Class with Multiple Methods

Example :-

```
class Car {
  constructor(brand, model) {
    this.brand = brand;
    this.model = model;
  }

  start() {
    console.log(`${this.brand} ${this.model} started`);
  }

  stop() {
    console.log(`${this.brand} ${this.model} stopped`);
  }
}

const car1 = new Car("Toyota", "Corolla");
car1.start();    // Toyota Corolla started
car1.stop();     // Toyota Corolla stopped
```

5 Inheritance with extends

Example :-

```
class Animal {
  constructor(name) {
    this.name = name;
  }

  speak() {
    console.log(`${this.name} makes a sound.`);
  }
}

class Dog extends Animal {
  constructor(name, breed) {
    super(name); // call parent constructor
    this.breed = breed;
  }

  speak() {
    console.log(`${this.name} barks.`);
  }
}

const dog = new Dog("Buddy", "Labrador");
dog.speak();    // Buddy barks.
```

6 super() Keyword

- The super() keyword is used to call the parent class constructor.
- It must be called before using this in a child constructor.
- It can also be used to call parent class methods.

Example :-

```
class Parent {
  hello() {
    console.log("Hello from Parent");
  }
}
class Child extends Parent {
  hello() {
    super.hello();
    console.log("Hello from Child");
  }
}
const c = new Child();
c.hello();          // Hello from Parent
                    // Hello from Child
```

7 Static Methods and Properties

- Static methods belong to the class itself, not to instances.
- You can call them using the class name.

Example :-

```
class MathUtil {
  static add(a, b) {
    return a + b;
  }

  static PI = 3.1416;    // static property
}

console.log(MathUtil.add(5, 3));    // 8
console.log(MathUtil.PI);           // 3.1416

const m = new MathUtil();
// m.add(1,2);  ✗ Error (not accessible)
```

8 Getters and Setters

- Used to control access to properties.
- Useful for validation and encapsulation.

Example :-

```
class User {
  constructor(name, age) {
    this.name = name;
    this._age = age;
  }
  get age() {
    return this._age;
  }
  set age(value) {
    if (value < 0) {
      console.log("Age cannot be negative");
    } else {
      this._age = value;
    }
  }
}
const u = new User("Sovit", 22);
console.log(u.age);  // 22
u.age = 25;
console.log(u.age);  // 25
u.age = -5;          // Age cannot be negative
```

9 Class vs Prototype (Quick Comparison)

Feature	Class (ES6)	Prototype (Old Way)
Introduced	ES6 (2015)	From the beginning
Syntax	Cleaner, modern	More verbose
Ease of use	Easier to understand	Complex
Underlying mechanism	Still uses prototypes	Directly uses prototypes
Use case	Preferred for modern code	Useful for deeper understanding

10 Real-world Use Cases

- ✓ Creating reusable UI components.
- ✓ Models in backend applications (e.g., User, Product).
- ✓ Game development (e.g., Player, Enemy classes).
- ✓ Building libraries and frameworks.
- ✓ Organizing large codebases with OOP principles.

11 Common Interview Questions

1. What is a class in JavaScript?
2. How is a class different from a constructor function?
3. What is the purpose of the constructor method?
4. Explain the super() keyword.
5. What are static methods and properties?
6. What are getters and setters?
7. How does inheritance work in JavaScript?
8. Are classes hoisted in JavaScript?

Key Takeaways :-

- ✓ Classes are syntactic sugar over prototypes.
- ✓ Use constructor() to initialize properties.
- ✓ Use extends for inheritance.
- ✓ Use super() to call parent constructor/methods.
- ✓ Static methods belong to the class.
- ✓ Getters and setters help with validation.
- ✓ Use classes for cleaner and maintainable code.

Karyvio.com

"Learn Today. Build Tomorrow."

Karyvio
LEARN | BUILD | GROW | GET HIRED

Modules in JavaScript (ES6)

Modules allow us to split our code into multiple files, making it more organized, reusable and maintainable. ES6 introduced a native module system using **import** and **export**.

"Good code is modular code."

1 What are Modules?

- A module is a separate file containing JavaScript code.
- Each module has its own scope.
- Variables, functions, classes inside a module are not accessible outside unless exported.
- Helps in better code organization, reusability and avoiding global scope pollution.

2 Why Use Modules?

- ✓ Keeps code organized in separate files.
- ✓ Avoids global scope pollution.
- ✓ Makes code reusable.
- ✓ Easier to maintain and scale.
- ✓ Helps in collaborative development.
- ✓ Works well with modern frameworks (React, Vue, etc.).

Real-world Analogy:
Think of modules like separate subject notebooks. Each notebook has its own notes, and you only open the ones you need instead of keeping everything in one big notebook.

3 Exporting from a Module

There are two types of exports:

a) Named Export (export multiple values)

```
// math.js
export const PI = 3.1416;
export function add(a, b) {
    return a + b;
}
export class Calculator {
    multiply(a, b) {
        return a * b;
    }
}
```

You can export multiple values using named exports.

b) Default Export (export a single value)

```
// message.js
export default function greet(name) {
    return `Hello, ${name}!`;
}
```

Only one default export is allowed per module.

4 Importing in a Module

There are two types of imports:

a) Named Import

```
// app.js
import { PI, add, Calculator } from './math.js';
console.log(PI);
console.log(add(5, 3));
const calc = new Calculator();
console.log(calc.multiply(4, 2));
```

b) Default Import

```
// app.js
import greet from './message.js';
console.log(greet("Sovit"));
```

c) Rename Import (alias)

```
import { add as sum } from './math.js';
console.log(sum(10, 5));   // 15
```

d) Import Everything as an Object

```
import * as MathUtil from './math.js';
console.log(MathUtil.PI);
console.log(MathUtil.add(2, 3));
```

5 Module File Structure

```
project/
├── index.html
├── app.js        // main file
├── math.js       // named exports
└── message.js    // default export
```

Use .js extension and relative path (./) to import local modules.

6 Using Modules in HTML

```
<!-- index.html -->
<!DOCTYPE html>
<html>
 <body>
   <h1>JavaScript Modules</h1>
   <script type="module" src="app.js"></script>
 </body>
</html>
```

Always use type="module" in the script tag to enable ES6 modules.

7 Important Points

- Modules are loaded in strict mode by default.
- Each module has its own scope.
- Use relative (./) or absolute paths.
- Browser requires type="module" in <script>.
- You can't use require() (that's for CommonJS in Node.js).
- Modules can also be dynamically imported.

8 Dynamic Import (Advanced)

```
// Load module only when needed
import('./math.js').then(module => {
    console.log(module.add(5, 3));
});
// or using async/await
async function load() {
    const module = await import('./math.js');
    console.log(module.PI);
}
load();
```

9 Common Interview Questions

1. What are modules in JavaScript?
2. Difference between named and default export?
3. Can we export multiple values using default export?
4. What is the purpose of type="module"?
5. How does dynamic import work?
6. Can we use modules in older browsers?
7. How are modules different from CommonJS (require)?

Karyvio
LEARN | BUILD | GROW | GET HIRED

Promises in JavaScript

Promises help us handle asynchronous operations in a cleaner way. They represent a value that may be available now, in the future, or never.

"A Promise is a commitment to a future value."

1 What is a Promise?

- A Promise is an object that represents the eventual completion (or failure) of an asynchronous operation.
- It can be in one of three states:
- Pending – initial state (not fulfilled or rejected).
- Fulfilled – operation completed successfully.
- Rejected – operation failed.
- A Promise can only be settled once (either fulfilled or rejected).

Think of it as:

A food order at a restaurant.
You place the order (pending), you either get the food (fulfilled) or they say it's not available (rejected).

2 Creating a Promise

Example :-

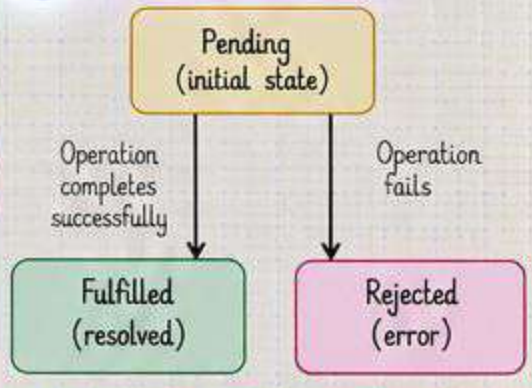
```
const promise = new Promise((resolve, reject) => {
  const success = true;

  setTimeout(() => {
    if (success) {
      resolve("Operation successful!");
    } else {
      reject("Something went wrong!");
    }
  }, 2000);
});

console.log(promise); // Promise { <pending> }
```

Here, resolve() is called when the operation succeeds, and reject() is called when it fails.

3 Promise States



Note :-

A Promise can only change from
Pending → Fulfilled or
Pending → Rejected.
It cannot go back to Pending again.

4 Consuming a Promise (then, catch, finally)

Example :-

```
const promise = new Promise((resolve, reject) => {
  let succees = true;
  setTimeout(() => {
    if (success) {
      resolve("Data loaded successfully!");
    } else {
      reject("Failed to load data!");
  }, 1000);
});

promise
  .then((result) => {
    console.log("Then:", result);   // runs if resolved
  })
  .catch((error) => {
    console.log("Catch:", error);   // runs if rejected
  })
  .finally(() => {
    console.log("Finally: Operation completed."); // always runs
});
```

Output (if success = true):

Then: Data loaded successfully!
Finally: Operation completed.

Output (if success = false):

Catch: Failed to load data!
Finally: Operation completed.

finally() runs regardless of success or failure.

5 Chaining Promises

Example :-

```
function step1() {
  return new Promise((resolve) => {
    setTimeout(() => {
      console.log("Step 1 complete");
      resolve("Data from step 1");
    }, 1000);
  });
}
function step2(data) {
  return new Promise((resolve) => {
    setTimeout(() => {
      console.log("Step 2 complete");
      resolve(data + " → processed");
    }, 1000);
  });
}
step1()
  .then(step2)
  .then((result) => {
    console.log("Final Result:", result);
  })
  .catch((error) => console.log("Error:", error));
```

Promises can be chained to perform multiple asynchronous tasks in sequence.

6 Promise Combinators

JavaScript provides some built-in methods to work with multiple promises.

Method	Description	Example
Promise.all()	Waits for all promises to fulfill (rejects if any fails).	Promise.all([p1, p2])
Promise.allSettled()	Waits for all promises to settle (never rejects).	Promise.allSettled([p1, p2])
Promise.race()	Returns the first settled promise (fulfilled or rejected).	Promise.race([p1, p2])
Promise.any()	Returns the first fulfilled promise (ignores rejections).	Promise.any([p1, p2])

7 Example: Promise.all()

```
const p1 = Promise.resolve("User data");
const p2 = new Promise((resolve) =>
  setTimeout(() => resolve("Posts data"), 1000)
);

Promise.all([p1, p2])
  .then((results) => {
  console.log(results); // ['User data', 'Posts data']
})
  .catch((error) => console.log("Error:", error));
```

Useful when you need to run multiple async tasks in parallel and wait for all of them.

8 Error Handling Best Practices

- ✓ Always use .catch() to handle errors.
- ✓ Use meaningful error messages.
- ✓ Avoid nesting .then(), prefer async/await for complex flows.
- ✓ Use .finally() for cleanup (e.g., hide loading spinner).
- ✓ Handle errors at the right level (don't swallow errors).
- ✓ Use Promise.allSettled() when you don't want one failure to stop everything.

9 Common Interview Questions

1. What is a Promise in JavaScript?
2. What are the states of a Promise?
3. What is the difference between Promise.all() and Promise.allSettled()?
4. What does Promise.race() do?
5. What is the purpose of Promise.any()?
6. How does error handling work in Promises?
7. How can you chain multiple promises?
8. Why is finally() useful?

10 Key Takeaways

- ✓ Promises make asynchronous code easier to read and maintain.
- ✓ They can be pending, fulfilled, or rejected.
- ✓ Use .then(), .catch(), and .finally().
- ✓ Use combinators for multiple promises.
- ✓ Prefer async/await for cleaner syntax.

Karyvio
LEARN | BUILD | GROW | GET HIRED

Async/Await and Fetch API in JavaScript

Async/Await is a modern way to work with asynchronous code, built on top of Promises. The Fetch API is a modern browser API used to make HTTP requests and work with web data (like from an API).

"Async/Await makes asynchronous code look synchronous, which is easier to read and maintain."

1 Why Async/Await?

- Makes asynchronous code cleaner and more readable.
- Avoids callback hell and complex .then() chains.
- Works with Promises (not a replacement, just syntactic sugar).
- Helps us write code that looks synchronous but runs asynchronously.

2 Basic Syntax

Example :-

```
async function greet() {
  return "Hello, Karyvio!";
}

greet().then((msg) => console.log(msg));
// Hello, Karyvio!
```

An async function always returns a Promise, even if you return a normal value.

3 Using await

Example :-

```
function delay(ms) {
  return new Promise((resolve) =>
    setTimeout(() => resolve("Done!"), ms)
  );
}

async function test() {
  console.log("Start");
  const result = await delay(2000);
  console.log(result);
  console.log("End");
}                    //  Start
                     //  Done!
test();              //  End
```

The await keyword pauses the execution inside an async function until the Promise is settled.

4 Error Handling with try...catch

Example :-

```
async function fetchData() {
  try {
    const res = await fetch("https://invalid-api.com");
    const data = await res.json();
    console.log(data);
  } catch (error) {
    console.log("Error:", error.message);
  }
}
fetchData();
```

Use try...catch to handle errors in async/await code.

5 The Fetch API Basics

- Fetch API is used to make HTTP requests (GET, POST, etc.).
- It returns a Promise that resolves to a Response object.
- We usually convert the response to JSON using .json().
- It replaces older methods like XMLHttpRequest.

Fetch works with Promises!

6 Making a GET Request

Example :-

```
async function getPosts() {
  try {
    const res = await fetch("https://jsonplaceholder.typicode.com/posts");
    const data = await res.json();
    console.log(data.slice(0, 3));   // first 3 posts
  } catch (error) {
    console.log("Failed to fetch posts:", error.message);
  }
}                                    // Array of post objects
getPosts();
```

7 Handling Response

- The fetch() function does not reject the Promise for HTTP error status codes (like 404 or 500).
- You need to check response.ok or response.status manually.

res.ok is true for status in the range 200-299.

Example :-

```
const res = await fetch("https://jsonplaceholder.typicode.com/invalid");
if (!res.ok) {
  throw new Error(`HTTP error! Status: ${res.status}`);
}
const data = await res.json();
```

8 Making a POST Request

Example :-

```
async function createPost() {
  const res = await fetch("https://jsonplaceholder.typicode.com/posts", {
    method: "POST",
    headers: {
      "Content-Type": "application/json"
    },
    body: JSON.stringify({
      title: "Learning JS",
      body: "Async/Await and Fetch API",
      userId: 1
    })
  });
  const data = await res.json();
  console.log(data);
}
createPost();
```

Use method, headers, and body for POST requests.

9 Real-world Example (API with loading & error)

Example :-

```
async function loadUsers() {
  try {
    console.log("Loading users...");
    const res = await fetch("https://jsonplaceholder.typicode.com/users");
    if (!res.ok) throw new Error("Failed to fetch users");
    const users = await res.json();
    console.log("Users:", users.slice(0, 2));
  } catch (error) {
    console.log("Error:", error.message);
  } finally {
    console.log("Done! ");
  }
}
loadUsers();
```

Use try...catch... finally for better control flow (loading, error, cleanup).

10 Key Takeaways

- ✓ Async/Await makes async code easier to read and write.
- ✓ Always use try...catch for error handling.
- ✓ Fetch API is powerful and widely used for working with real APIs.
- ✓ Check response.ok to handle HTTP errors.
- ✓ You can use fetch for GET, POST, PUT, DELETE requests.
- ✓ Combine async/await with modern JavaScript features for clean and efficient code.

Karyvio
LEARN | BUILD | GROW | GET HIRED

The Event Loop in JavaScript

The Event Loop is a core part of JavaScript's runtime environment. It handles asynchronous operations, making non-blocking behavior possible even though JavaScript is single-threaded.

"The Event Loop turns waiting time into productive time."

1 What is the Event Loop?

- The Event Loop is a mechanism that checks the Call Stack and the Callback Queues, and moves callbacks to the Call Stack when it's empty.
- It allows JavaScript to perform non-blocking operations.
- It is part of the JavaScript runtime environment (like browsers or Node.js), not the JavaScript language itself.

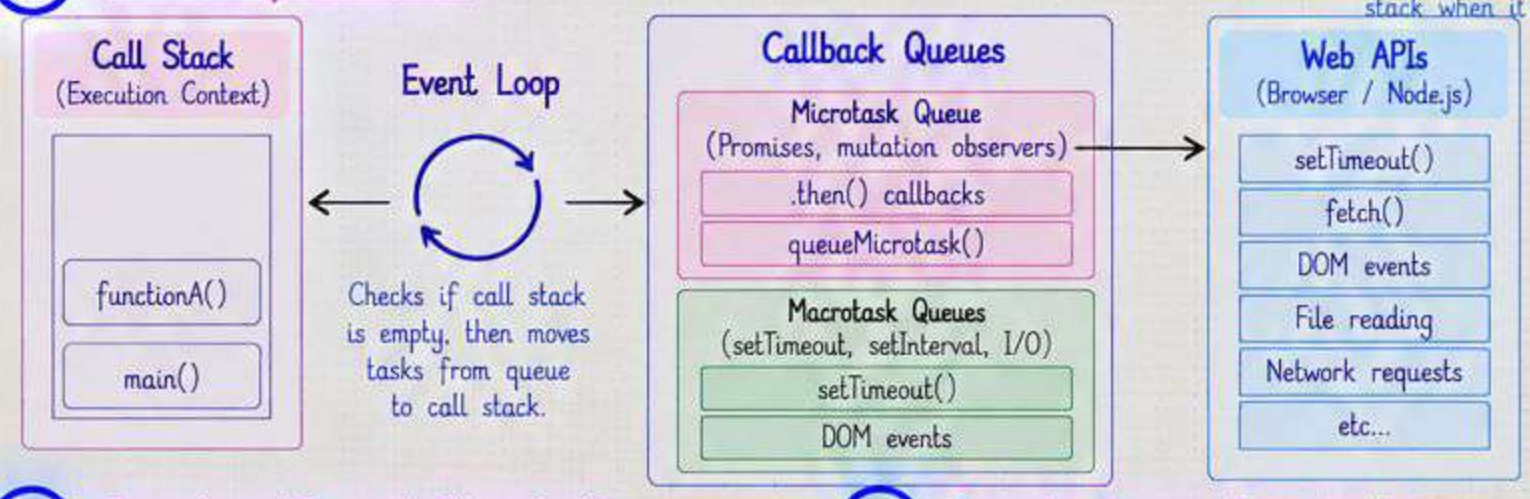
2 Why do we need it?

- JavaScript is single-threaded (one thing at a time).
- Without the Event Loop, long-running tasks (like network requests, timers, file operations) would block the entire program.
- The Event Loop allows asynchronous tasks to be handled efficiently without blocking the main thread.

3 Components Involved

1. **Call Stack**
   Keeps track of function execution (Last In, First Out).
2. **Web APIs**
   Provided by the browser (or Node.js) to handle asynchronous tasks (e.g., setTimeout, fetch, DOM events).
3. **Callback Queues**
   Queues where asynchronous callbacks are placed (e.g., Microtask Queue, Macrotask Queue).
4. **Event Loop**
   Monitors the call stack and moves callbacks from the queues to the call stack when it's empty.

4 Visual Representation



5 Example: setTimeout (Macrotask)

```
console.log("Start");
setTimeout(() => {
    console.log("Inside setTimeout");
}, 0);
console.log("End");
```

Output:

```
Start
End
Inside setTimeout
```

6 Example: Promise (Microtask)

```
console.log("Start");
Promise.resolve().then(() => {
    console.log("Inside Promise");
});
console.log("End");
```

Output:

```
Start
End
Inside Promise
```

Note :-

- Promise callbacks (Microtasks) run before setTimeout (Macrotasks), even if the timeout is 0ms.

7 Example: Combined (Microtask vs Macrotask)

```
console.log("1");
setTimeout(() => console.log("2"), 0);
Promise.resolve().then(() => console.log("3"));
console.log("4");
```

Output:

```
1
4
3   (Microtask)
2   (Macrotask)
```

8 Order of Execution

1. Execute all synchronous code (in call stack).
2. Execute all microtasks (e.g., Promise .then, queueMicrotask).
3. Execute one macrotask (e.g., setTimeout, setInterval, DOM events).
4. Repeat the process.

9 Microtasks vs Macrotasks

Microtasks	Macrotasks
Higher priority (runs first)	Runs after microtasks
Includes Promise .then(), queueMicrotask(), MutationObserver	Includes setTimeout(), setInterval(), DOM events, I/O, setImmediate (Node.js)
Runs immediately after the current task	Runs in the next event loop cycle
Can starve macrotasks if added continuously	Used for scheduled or external asynchronous tasks

10 Real-world Analogy

Think of a café:

- Call Stack → The chef (handles one order at a time).
- Web APIs → The kitchen (prepares food in background).
- Callback Queue → Completed orders waiting to be served (Microtasks = priority orders, Macrotasks = regular orders).
- Event Loop → The waiter (keeps checking and brings orders to the chef when he is free).

"The Event Loop ensures nobody waits unnecessarily, even in a single-threaded world."

11 Key Takeaways

- ✓ JavaScript is single-threaded, but the Event Loop enables asynchronous behavior.
- ✓ Microtasks (Promises) run before Macrotasks (setTimeout, DOM events).
- ✓ The Event Loop continuously checks the call stack and moves tasks from queues.
- ✓ Understanding the Event Loop is crucial for performance, debugging, and writing better async code.

12 Common Interview Questions

1. What is the Event Loop in JavaScript?
2. What is the difference between Microtasks and Macrotasks?
3. Why do Promises run before setTimeout?
4. How does the Event Loop work in the browser?
5. How does the Event Loop work in Node.js?
6. Can the Event Loop be blocked?
7. What happens if you keep adding microtasks?

Karyvio
LEARN | BUILD | GROW | GET HIRED

Advanced JavaScript Concepts

These advanced concepts help you write more powerful, efficient, and maintainable code. They are commonly used in real-world projects and often asked in interviews.

"Master the fundamentals, explore the advanced, and build without limits."

1 IIFE (Immediately Invoked Function Expression)

- A function that runs immediately after definition.
- Creates its own scope and avoids polluting the global scope.
- Useful for initialization code.

Example :-

```
(function() {
  const message = "IIFE is running!";
  console.log(message);
})(); // IIFE
```

Output :

```
IIFE is running!
```

Use Cases :-

- Avoid global scope pollution.
- Initialize configuration.

2 Higher-Order Functions

- Functions that take another function as an argument or return a function.
- Common examples: map, filter, reduce.

Example :-

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

Output :-

```
[ 2, 4, 6, 8 ]
```

Built-in higher-order functions: map(), filter(), reduce(), forEach(), some(), every(), find(), sort()

3 Function Currying

- Transform a function with multiple arguments into a sequence of functions, each taking a single argument.
- Helps in code reusability and partial application.

Example :-

```
function multiply(a) {
  return function(b) {
    return a * b;
  }
}
const double = multiply(2);
console.log(double(5));    // 10
```

4 Composition vs Inheritance

- Composition: Build complex functionality by combining smaller functions or objects.
- Inheritance: Create new objects based on existing ones.

Example (Composition) :-

```
const logger = (fn) => (...args) => {
  console.log("Calling function...");
  return fn(...args);
}

const add = (a, b) => a + b;
const loggedAdd = logger(add);
console.log(loggedAdd(2, 3));

// Output:
// Calling function...
// 5
```

Why Composition?

- ✔ More flexible than inheritance.
- ✔ Avoids tight coupling.
- ✔ Easier to test and maintain.

5 Memoization

- Optimization technique to cache expensive function results.
- Returns cached result if the same input is provided again.

Example :-

```
function memoize(fn) {
  const cache = {};
  return function(...args) {
    const key = JSON.stringify(args);
    if (key in cache) return cache[key];
    const result = fn(...args);
    cache[key] = result;
    return result;
  }
}
const factorial = memoize(n =>
  n <= 1 ? 1 : n * factorial(n - 1));
console.log(factorial(5));   // 120
console.log(factorial(5));   // from cache
```

6 Debouncing and Throttling

- Debouncing: Limits a function call until a certain time has passed since the last call.
- Throttling: Ensures a function is called at most once in a given time interval.

Debounce Example :-

```
function debounce(fn, delay) {
  let timer;
  return (...args) => {
    clearTimeout(timer);
    timer = setTimeout(() => {
      fn(...args);
    }, delay);
  }
}
// Useful in search boxes, resize events etc.
```

7 Deep Copy vs Shallow Copy

- Shallow Copy: Copies only top-level properties (nested objects are still referenced).
- Deep Copy: Creates a completely independent copy.

Example :-

```
const original = { name: "Karyvio",
  details: { age: 22 };

const shallow = { ...original };
const deep = JSON.parse(JSON.stringify(original));

shallow.details.age = 25;
console.log(original.details.age); // 25

deep.details.age = 30;
console.log(original.details.age); // 25
```

Deep Copy (modern way):

```
const deepCopy =
  structuredClone(original);
```

Note :-

Use structuredClone() (in modern browsers) for deep copying complex objects (includes Date, Map, Set, etc.).

8 Symbols (Advanced Use Cases)

- Symbols are unique and immutable values.
- Often used for private object properties to avoid name conflicts.

Example :-

```
const id = Symbol("id");
const user = {
  name: "Sovit",
  [id]: 12345
};
console.log(user[id]); // 12345
console.log(Object.keys(user));
// [ 'name' ]  (symbol is not enumerable)
```

Symbols are often used in libraries and frameworks to define internal properties.

9 Iterators and Generators

- Iterators: Objects that define a sequence and how to access elements.
- Generators: Functions that can pause and resume execution using yield.

Example (Generator) :-

```
function* count() {
  yield 1;
  yield 2;
  yield 3;
}

const gen = count();
console.log(gen.next().value);  // 1
console.log(gen.next().value);  // 2
console.log(gen.next().value);  // 3
console.log(gen.next().done);   // true
```

10 Proxies (Meta Programming)

- Proxies allow you to intercept and customize operations on objects (like get, set, delete, etc.).
- Useful for validation, logging, or creating custom behavior.

Example :-

```
const user = { name: "Karyvio", age: 22 };
const proxy = new Proxy(user, {
  get(target, prop) {
    console.log(`Accessing ${prop}`);
    return target[prop];
  }
});
console.log(proxy.name);      // Accessing name
                              // Karyvio
```

Key Takeaways :-

- ✔ Advanced JS concepts help you write cleaner, smarter, and more efficient code.
- ✔ Use higher-order functions and composition for better code structure.
- ✔ Memoization, debouncing, and throttling improve performance.
- ✔ Understand deep vs shallow copy to avoid unexpected bugs.
- ✔ Generators, Symbols, and Proxies are powerful tools used in real-world libraries.
- ✔ These concepts are commonly asked in interviews and used in modern frameworks.

10 Common Interview Questions

1. What is an IIFE? Why is it used?
2. Difference between composition and inheritance?
3. How does memoization work?
4. What is the difference between debouncing and throttling?
5. How can you create a deep copy of an object?
6. What are Symbols? Give a real-world use case.
7. What are generators and how do they work?
8. What are Proxies? Give an example.

Karyvio
LEARN | BUILD | GROW | GET HIRED

Best Practices, Performance & Security in JavaScript

Writing JavaScript is not just about making it work, but also about writing clean, maintainable, secure and performant code. These best practices will help you become a better developer and build real-world applications.

"Good developers write code that works. Great developers write code that lasts."

1 Write Clean and Readable Code

- ✔ Use meaningful variable and function names.
- ✔ Keep functions small and focused (single responsibility).
- ✔ Use consistent indentation and formatting.
- ✔ Add comments where necessary (but don't overdo it).
- ✔ Follow a coding style guide (e.g., Airbnb, StandardJS).
- ✔ Use linters (ESLint) and formatters (Prettier).

Example :-

✘ Bad Practice

```
function c(d){
  return d*2
}
```

✔ Good Practice

```
function calculateDouble(number) {
  return number * 2;
}
```

2 Avoid Common Mistakes

- Avoid using var (use let or const).
- Don't use == (use ===).
- Avoid global variables.
- Don't modify built-in prototypes.
- Handle errors properly.
- Avoid deeply nested code (callback hell).
- Don't ignore returned Promises.
- Be careful with this keyword.
- Avoid memory leaks (remove unused event listeners).
- Don't use eval() (security risk).

3 Code Formatting Tools

- ESLint → Helps find and fix problems in your code.
- Prettier → Automatically formats your code.
- EditorConfig → Keeps consistent settings across different editors.
- Use these tools in your project for a professional codebase.

.eslintrc example :-

```
{
  "env": {"browser": true, "es2021": true},
  "extends": "eslint:recommended",
  "rules": {
    "no-unused-vars": "warn",
    "no-console": "off"
  }
}
```

4 Performance Optimization

- ✔ Minimize DOM manipulation (it's slow).
- ✔ Use document fragments for multiple DOM updates.
- ✔ Debounce or throttle event handlers.
- ✔ Lazy load images and modules.
- ✔ Use efficient data structures (Map, Set).
- ✔ Avoid heavy computations in the main thread.
- ✔ Use Web Workers for CPU intensive tasks.
- ✔ Optimize loops and avoid unnecessary iterations.

Example: Debounce Function

```
function debounce(fn, delay) {
  let timer;
  return function (...args) {
    clearTimeout(timer);
    timer = setTimeout(() => fn(...args), delay);
  };
}
```

5 Memory Management

- Avoid memory leaks by cleaning up event listeners, timers, and subscriptions.
- Remove unused DOM elements.
- Use WeakMap and WeakSet for objects that should be garbage collected.
- Be careful with closures (they can keep references alive).

Example: Removing Event Listener

```
function handleClick() {
  console.log("Button clicked");
}
button.addEventListener("click", handleClick);

// Later, when not needed
button.romoveEventListener("click", handleClick);
```

6 Security Best Practices

- ✔ Avoid eval() and Function() constructor.
- ✔ Sanitize user input (prevent XSS attacks).
- ✔ Use HTTPS for all API requests.
- ✔ Store sensitive data securely (e.g., environment variables).
- ✔ Validate data on both client and server.
- ✔ Be careful with third-party libraries (check for vulnerabilities).
- ✔ Use Content Security Policy (CSP).
- ✔ Keep dependencies up to date.

Example: Preventing XSS

```
const userInput = "<script>alert('Hacked')</script>";

// Sanitize input (simple example)
const safeInput = userInput.replace(/</g, "&lt;").
                            .replace(/>/g, "&gt;");
console.log(safeInput);
// &lt;script&gt;alert('Hacked')&lt;/script&gt;
```

7 Use Modern JavaScript Features

- Use let, const, arrow functions, template literals.
- Use destructuring and spread/rest operators.
- Use modules (import/export).
- Use async/await instead of callbacks.
- Use optional chaining (?.) and nullish coalescing (??).
- These features make your code cleaner and safer.

Example:

```
const user = {
  name: "Sovit",
  address: { city: "Bhubaneswar" }
};
console.log(user.address?.city ?? "Unknown");
// Bhubaneswar
```

8 Testing Your Code

- ✔ Write unit tests for critical functions.
- ✔ Use testing frameworks (Jest, Mocha).
- ✔ Test edge cases (null, undefined, invalid input).
- ✔ Use integration and end-to-end tests for larger projects.

Example: Jest Test

```
function add(a, b) {
  return a + b;
}
test("adds two numbers", () => {
  expect(add(2, 3)).toBe(5);
});
```

9 Useful Developer Tips

- ✔ Use browser DevTools effectively.
- ✔ Use console.table() for better debugging.
- ✔ Log meaningful messages.
- ✔ Use keyboard shortcuts (productivity boost).
- ✔ Learn and use useful npm packages.
- ✔ Read and understand error messages.
- ✔ Keep learning and practice regularly.
- ✔ Build real projects (that's the best way to learn).

Remember :-

- Practice more, copy less.
- Build projects, not just notes.
- Clean code today, better opportunities tomorrow.
- Keep learning, the JavaScript ecosystem is always evolving!

10 Final Takeaways

- ✔ JavaScript is powerful, but with great power comes great responsibility.
- ✔ Follow best practices to write clean, efficient, and secure code.
- ✔ Optimize performance and avoid common pitfalls.
- ✔ Use modern features and tools to improve productivity.
- ✔ Keep practicing, building projects, and stay updated with new features.
- ✔ With the right mindset and discipline, you can become a great JavaScript developer!

You've completed 40 pages of JavaScript notes!
Now it's time to apply your knowledge.
build real projects, and create the future you want.

Keep Going!