JavaScript Notes

1. JavaScript

JavaScript is a high-level, interpreted scripting language primarily used to make web pages interactive. It allows you to implement complex features like real-time updates, interactive forms, animations, and more. JavaScript is an essential part of web development along with HTML and CSS.

- JavaScript is case-sensitive.
- It runs on the client-side in the browser, and with Node.js, it can also run on the server.
- Created by Brendan Eich in 1995.
- It follows ECMAScript standards.

---

2. Using the Console

The console is used in JavaScript for debugging and displaying output.

```
console.log("Hello, JavaScript"); // Prints message to the console
console.error("This is an error"); // Shows error message
console.warn("This is a warning"); // Shows warning message
```

The console is accessible via browser developer tools (usually opened with F12).

---

3. What is a Variable

Variables are containers for storing data values. In JavaScript, you can declare variables using var, let, or const.

- var is function-scoped
- let and const are block-scoped
- Use let for values that may change and const for values that should remain constant

```
var name = "John";
let age = 25;
```

```
const pi = 3.1415;
```

---

4. Data Types in JavaScript

JavaScript supports the following data types.

Primitive Data Types:

- Number
- String
- Boolean
- Undefined
- Null
- Symbol
- BigInt

Non-Primitive Data Types:

- Object
- Array
- Function

```
let num = 10;              // Number
let str = "JavaScript";    // String
let isOpen = true;         // Boolean
let x;                     // Undefined
let y = null;              // Null
let person = { name: "John", age: 30 }; // Object
let arr = [1, 2, 3];       // Array
```

---

5. Numbers in JavaScript

All numeric values (integers and floats) fall under the number data type in JavaScript.

```
let a = 10;
let b = 3.14;
```

JavaScript provides built-in functions to convert and manipulate numbers, such as:

```
parseInt("100");   // 100
parseFloat("12.34"); // 12.34
Number("123");     // 123
```

---

6. Operations in JavaScript

Arithmetic operations:

```
let a = 10 + 5;  // Addition
let b = 10 - 5;  // Subtraction
let c = 10 * 2;  // Multiplication
let d = 10 / 2;  // Division
let e = 10 % 3;  // Modulus
```

Comparison operations:

```
console.log(10 > 5);   // true
console.log(10 === "10"); // false
console.log(10 == "10");  // true (loose equality)
```

Logical operations:

```
true && false;  // false
true || false;  // true
!true;          // false
```

---

7. NaN in JavaScript

NaN stands for "Not a Number". It is returned when a mathematical operation fails or cannot be performed.

Examples:

```
console.log(0 / 0);          // NaN
console.log("abc" - 1);      // NaN
console.log(Number("abc"));  // NaN
```

You can check for NaN using isNaN() function:

```
isNaN("abc");  // true
```

---

8. Operator Precedence

Operator precedence determines the order in which operations are performed in an expression.

Example:

```
let result = 5 + 3 * 2;  // result = 11, because multiplication has higher precedence
```

Use parentheses to control precedence:

```
let result = (5 + 3) * 2; // result = 16
```

---

9. let, const and var Keywords

Comparison:

Keyword	Scope	Reassignment	Redeclaration	Hoisted
var	Function	Yes	Yes	Yes
let	Block	Yes	No	No
const	Block	No	No	No

Examples:

```
var x = 10;
let y = 20;
const z = 30;
```

Note:

- const must be initialized during declaration.
- let and const prevent redeclaration in the same scope.

---

10. Assignment Operators

Assignment operators are used to assign values to variables.

```
x = 10;     // Simple assignment
x += 5;     // x = x + 5
x -= 3;     // x = x - 3
x *= 2;     // x = x * 2
x /= 2;     // x = x / 2
x %= 2;     // x = x % 2
```

---

11. Unary Operators

Unary operators operate on a single operand.

```
let x = 5;
console.log(-x);    // -5
console.log(+x);    // 5

x++;   // Post-increment (increment after use)
++x;   // Pre-increment (increment before use)

x--;   // Post-decrement
--x;   // Pre-decrement
```

---

12. Identifier Rules

Identifiers are names used for variables, functions, etc.

Rules for valid identifiers:

- Must begin with a letter, underscore _, or dollar sign $
- Cannot start with a number
- Cannot use reserved JavaScript keywords (like var, let, if, else, etc.)
- Case-sensitive

Examples:

```
let _name = "John";
```

```
let $salary = 5000;
let student1 = "Alice";
```

---

13. Boolean in JavaScript

Booleans represent logical entities and can have two values: true or false.

```
let isOnline = true;
let isLoggedIn = false;
```

Falsy values in JavaScript:

- false
- 0
- ""
- null
- undefined
- NaN

Any value not falsy is considered truthy.

---

14. What is TypeScript

TypeScript is a strongly typed, object-oriented, compiled language. It is a superset of JavaScript that adds optional static typing.

- Developed by Microsoft.
- Code is written in .ts files and compiled to JavaScript.
- Helps in large-scale applications and catching errors early.

Example:

```
let age: number = 25;
let name: string = "Prathamesh";
```

TypeScript enforces types during compilation, whereas JavaScript is dynamically typed.

---

15. String in JavaScript

Strings are used to store text. You can define strings using single quotes, double quotes, or backticks.

```
let str1 = "Hello";
let str2 = 'World';
let str3 = `JavaScript`;
```

Common string methods:

```
let name = "JavaScript";
name.length;             // 10
name.toUpperCase();      // "JAVASCRIPT"
name.toLowerCase();      // "javascript"
name.includes("Script"); // true
name.indexOf("a");       // 1
```

---

16. String Indices

Strings are indexed, meaning each character in a string has a numeric index starting from 0.

```
let str = "Hello";
console.log(str[0]);     // 'H'
console.log(str.charAt(1)); // 'e'
```

Strings are immutable. You cannot change a character using index assignment.

---

17. null and undefined in JavaScript

- undefined: A variable that has been declared but not assigned a value.
- null: An assignment value that represents no value or no object.

```
let a;
console.log(a);   // undefined

let b = null;
```

```
console.log(b);   // null
```

You can check types using typeof:

```
typeof undefined; // "undefined"
typeof null;      // "object" (this is a known quirk in JavaScript)
```

1. console.log()

Purpose
Used to print messages to the browser's console.

Syntax

```
console.log("Hello World");
```

Use Cases

- Debugging JavaScript code
- Viewing variable values and outputs

---

2. Linking JavaScript File

External JS Linking
Link a JavaScript file using the <script> tag.

Syntax

```
<script src="script.js"></script>
```

Where to Place
Usually placed before the closing </body> tag so HTML loads first.

---

3. Template Literals

Used For

- Embedding variables directly into strings
- Multiline strings

Syntax
Use backticks `` and ${} for variables.

```
let name = "Prathamesh";
console.log(`Hello, my name is ${name}`);
```

---

4. Operators in JavaScript

Types

- Arithmetic: +, -, *, /, %, **
- Assignment: =, +=, -=, etc.
- Increment and Decrement: ++, --

Example

```
let a = 10;
let b = 3;
console.log(a + b);
```

---

5. Comparison Operators

Used To
Compare values.

Types

- == (equal to – loose)
- === (strict equal)
- !=, !==, >, <, >=, <=

Example

```
console.log(5 == "5");     // true
console.log(5 === "5");    // false
```

---

6. Comparison for Non-Numbers

String Comparison
Alphabetical order is used.

"apple" > "banana" // false

null and undefined

null == undefined // true
null === undefined // false

---

7. Conditional Statements

Purpose
To execute different blocks of code based on conditions.

Syntax

```
if (condition) {
  // code
}
```

---

8. if Statement

Purpose
Executes code only if the condition is true.

Example

```
let age = 18;
if (age >= 18) {
  console.log("You are eligible");
}
```

---

9. else if Statement

Purpose
Provides multiple conditions.

Example

```
let marks = 75;
if (marks > 90) {
  console.log("A Grade");
} else if (marks > 70) {
  console.log("B Grade");
}
```

---

10. else Statement

Purpose
Runs when all other conditions are false.

Example

```
let temp = 15;
if (temp > 30) {
  console.log("Hot");
} else {
  console.log("Normal");
}
```

---

11. Nested if-else

Purpose
Using if inside another if.

Example

```
let score = 80;
if (score >= 50) {
  if (score >= 75) {
    console.log("Great job");
  } else {
    console.log("Passed");
  }
}
```

---

12. Logical Operators

Used To
Combine conditions.

Types

- && (AND)
- || (OR)
- ! (NOT)

Example

```
let age = 20;
if (age > 18 && age < 60) {
  console.log("Eligible");
}
```

---

13. Truthy and Falsy Values

Falsy Values in JavaScript

- false
- 0
- "" (empty string)
- null
- undefined
- NaN

Example

```
if ("") {
  console.log("Truthy");
} else {
  console.log("Falsy");
}
```

---

14. Switch Statement

Purpose
Clean alternative to many if-else conditions.

Syntax

```
let day = 2;
switch (day) {
  case 1: console.log("Monday"); break;
  case 2: console.log("Tuesday"); break;
  default: console.log("Other day");
}
```

---

15. Alerts and Prompts

alert()
Shows a popup message.

```
alert("Welcome");
```

prompt()
Takes user input and returns it.

```
let name = prompt("Enter your name");
alert("Hello " + name);
```

String Concepts in JavaScript

String Methods

Definition:
JavaScript strings come with many built-in methods for processing text.

Examples:

```
let name = "Prathamesh";
console.log(name.length);      // 10
console.log(name.charAt(0));   // P
```

Trim Method

Definition:
Removes leading and trailing whitespaces.

Example:

```
let str = "   Hello JS!   ";
console.log(str.trim()); // "Hello JS!"
```

Strings are Immutable in JS

Definition:
Strings cannot be changed after creation. Operations return a **new string**.

Example:

```
let str = "hello";
str[0] = 'H';
console.log(str); // "hello"
```

ToUpperCase and ToLowerCase

Definition:
Convert string characters to upper or lower case.

Example:

```
let greet = "Hello";
console.log(greet.toUpperCase()); // "HELLO"
console.log(greet.toLowerCase()); // "hello"
```

Methods with Arguments – indexOf

Definition:
Finds the **first occurrence** of a substring.

Example:

```
let text = "JavaScript is awesome";
console.log(text.indexOf("is")); // 11
```

Method Chaining

Definition:
Calling multiple methods one after another.

Example:

```
let msg = "   Hello   ";
console.log(msg.trim().toUpperCase().slice(0, 5)); // "HELLO"
```

Slice Method

Definition:
Extracts a section of a string.

Example:

```
let lang = "JavaScript";
console.log(lang.slice(4));      // "Script"
console.log(lang.slice(0, 4));   // "Java"
```

Replace & Repeat Method

- **replace():** Replaces substring.
- **repeat():** Repeats string multiple times.

Example:

```
let str = "I like Java";
console.log(str.replace("Java", "JavaScript")); // "I like JavaScript"
console.log("Ha ".repeat(3)); // "Ha Ha Ha "
```

Array Concepts in JavaScript

Array (Data Structure)

Definition:
Used to store **multiple values** in a single variable.

Example:

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

Visualizing Arrays

Each element has an index starting from 0.

Example:

```
let nums = [10, 20, 30];
console.log(nums[1]); // 20
```

Creating Arrays

You can create arrays using:

- **Array literals**
- **Array constructor**

Example:

```
let a = [1, 2, 3];
let b = new Array(4, 5, 6);
```

Arrays are Mutable

You can change elements of an array even if it's declared using const.

Example:

```
const fruits = ["apple", "banana"];
fruits[0] = "mango";
console.log(fruits); // ["mango", "banana"]
```

Array Methods

Common methods:

- push() – add to end
- pop() – remove from end
- shift() – remove from start
- unshift() – add to start

---

indexOf & includes Method

- indexOf() – returns index of first match
- includes() – returns true if found

Example:

```
let arr = [1, 2, 3];
console.log(arr.indexOf(2)); // 1
console.log(arr.includes(5)); // false
```

Concatenation & Reverse

- concat() – combines arrays
- reverse() – reverses array elements

Example:

```
let a = [1, 2];
let b = [3, 4];
console.log(a.concat(b)); // [1, 2, 3, 4]
console.log(a.reverse()); // [2, 1]
```

Slice in Arrays

Returns a shallow copy of part of an array.

Example:

```
let arr = [10, 20, 30, 40];
console.log(arr.slice(1, 3)); // [20, 30]
```

Splice in Arrays

Changes content by removing/replacing elements.

Syntax:

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

Example:

```
let arr = ["a", "b", "c"];
arr.splice(1, 1, "x"); // ["a", "x", "c"]
```

Sort in Arrays

- sort() sorts alphabetically
- For numbers, use custom compare function

Example:

```
let nums = [40, 10, 30];
nums.sort(); // Wrong: [10, 30, 40] lexically
nums.sort((a, b) => a - b); // Correct numerically
```

Arrays References

Arrays are stored as references.

Example:

```
let a = [1, 2];
let b = a;
b[0] = 9;
console.log(a); // [9, 2]
```

Constant Arrays

You can change elements of a const array, but not reassign it.

Example:

```
const a = [1, 2];
a.push(3); // Allowed
// a = [4, 5]; // Error
```

Nested Arrays

Arrays within arrays (2D arrays).

Example:

```
let matrix = [
  [1, 2],
  [3, 4]
];
console.log(matrix[1][0]); // 3
```

JavaScript Looping Concepts –

1.Loops (For Loop)

Definition: A loop is used to repeat a block of code a certain number of times.

Syntax:

```
for (initialization; condition; increment) {
```

```
  // code to be executed
}
```

Example:

```
for (let i = 1; i <= 5; i++) {
   console.log(i);
}
```

Purpose: Runs from 1 to 5 and logs each number.

2.Infinite Loops

Definition: A loop that never ends due to a condition that always evaluates to true.

Example:

```
// while (true) {
//     console.log("Infinite loop");
// }
```

 Note: Infinite loops can freeze or crash your browser. Use break to avoid them intentionally.

3.Nested For Loops

Definition: A loop inside another loop. Commonly used for multidimensional data structures.

Example:

```
for (let i = 1; i <= 3; i++) {
   for (let j = 1; j <= 2; j++) {
      console.log(`i=${i}, j=${j}`);
   }
}
```

Use Cases: Patterns, matrices, grids.

4.While Loops

Definition: Executes a block of code **as long as** the specified condition is true.

Syntax:

```
while (condition) {
   // code block
}
```

Example:

```
let i = 1;
while (i <= 5) {
   console.log(i);
   i++;
}
```

When to Use: When the number of iterations is unknown.

5.Break Keyword

Definition: Used to **exit the loop prematurely**.

Example:

```
for (let i = 1; i <= 10; i++) {
   if (i === 5) break;
   console.log(i);
}
```

Output: 1 2 3 4

Use Case: Exit when a condition is met (e.g., stop searching).

6.Loops with Arrays

Definition: Iterating through array elements using loops like for, while, or for-of.

Example (for loop):

```
const names = ["Alice", "Bob", "Charlie"];
for (let i = 0; i < names.length; i++) {
   console.log(names[i]);
}
```

Key Point: Access elements using index.

7.Loops with Nested Arrays

Definition: Looping through arrays inside arrays (2D arrays or matrices).

Example:

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

for (let i = 0; i < matrix.length; i++) {
   for (let j = 0; j < matrix[i].length; j++) {
      console.log(matrix[i][j]);
   }
}
```

Use Case: Handling tabular data like grids, tables, or matrices.

8.For-of Loops

Definition: A simpler way to iterate over **iterable objects** (arrays, strings, etc.).

Syntax:

```
for (const element of iterable) {
   // code block
}
```

Example:

```
const fruits = ["Apple", "Banana"];
for (const fruit of fruits) {
   console.log(fruit);
}
```

Advantage: Cleaner syntax, avoids index management.

9.Nested For-of Loops

Definition: For-of loops inside each other to handle nested structures like 2D arrays.

Example:

```
const teams = [
   ["Alice", "Bob"],
   ["Charlie", "Dave"]
];

for (const team of teams) {
   for (const member of team) {
      console.log(member);
   }
}
```

Use Case: Cleaner and readable way to handle nested arrays.

01. Object Literals

□ What is an Object Literal?

An object literal is a comma-separated list of **name-value pairs** inside {}.

```
const student = {
  name: "Prathamesh",
  age: 21,
  branch: "AIDS"
};
```

□ Key Points:

- No class needed.
- Can store various types (strings, numbers, arrays, objects).
- Each key is a string (implicitly).

---

02. Creating Object Literals & Get Values

□ Accessing Object Properties:

```
objectName.key            // Dot notation
objectName["key"]          // Bracket notation (useful for dynamic keys)

const car = {
  brand: "Toyota",
  model: "Fortuner"
};

console.log(car.brand);       // Toyota
console.log(car["model"]);    // Fortuner
```

---

03. Conversion in Get Values

□ Type Conversion when Getting Values:

```
const person = { age: "25" };
```

```
let ageNum = Number(person.age);    // "25" → 25 (string to number)
let ageStr = String(ageNum);        // 25 → "25" (number to string)

console.log(typeof ageNum);         // number
console.log(typeof ageStr);         // string
```

---

04. Add / Update Values

Add New Key-Value:

```
employee.salary = 50000;
```

Update Existing Key:

```
employee.department = "HR";
```

Example:

```
const employee = {
  name: "Ravi",
  department: "IT"
};

employee.salary = 50000;
employee.department = "HR";
console.log(employee);
```

---

05. Nested Objects

Object Inside Object:

```
const user = {
  name: "John",
  address: {
    city: "Pune",
    pin: 411001
  }
};

console.log(user.address.city);     // Pune
console.log(user["address"]["pin"]); // 411001
```

---

06. Array of Objects

 Define:

```
const products = [
  { id: 1, name: "Laptop", price: 45000 },
  { id: 2, name: "Phone", price: 25000 }
];
```

 Loop through Array:

```
products.forEach(product => {
  console.log(product.name, product.price);
});
```

---

07. Math Object

 Overview:

JavaScript Math is a built-in object for performing mathematical tasks.

 Common Methods:

Method	Description	Example
Math.PI	Value of PI	3.14159...
Math.round()	Round to nearest	Math.round(4.5) → 5
Math.ceil()	Round up	Math.ceil(4.2) → 5
Math.floor()	Round down	Math.floor(4.9) → 4
Math.sqrt()	Square root	Math.sqrt(25) → 5
Math.pow()	Exponentiation	Math.pow(2, 3) → 8
Math.min()	Smallest number	Math.min(3, 1, 5) → 1
Math.max()	Largest number	Math.max(3, 1, 5) → 5

08. Random Integers

Math.random() – Generates decimal between 0 and 1

```
Math.random(); // e.g., 0.462
```

Random Integer (1 to 10):

```
Math.floor(Math.random() * 10) + 1;
```

Random Integer (between min and max):

```
function getRandomInt(min, max) {
  return Math.floor(Math.random() * (max - min + 1)) + min;
}

getRandomInt(50, 100); // random between 50 and 100
```

1. What Are Functions?

Definition:
A function is a block of code designed to perform a particular task. It runs when something calls it.

Syntax:

```
function functionName() {
   // code to be executed
}
```

Example:

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

2. Functions with Arguments

Arguments:
You can pass values (arguments) to functions to make them reusable with different inputs.

Syntax:

```
function add(a, b) {
   console.log(a + b);
}
add(5, 10);
```

---

3. return Keyword

Definition:
return ends function execution and specifies the value to be returned.

Use Case:

- Useful when you want the function to **give a value back** to the caller.

Example:

```
function multiply(x, y) {
   return x * y;
}
let result = multiply(4, 5); // 20
```

---

4. What is Scope?

Scope defines **where variables can be accessed** in your code.

 Types of Scope:

- **Global Scope**: Outside all functions/blocks.
- **Local Scope**: Inside functions (also known as function scope).

Example:

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

function testScope() {
   let localVar = "I am local";
   console.log(globalVar); // Accessible
   console.log(localVar);  // Accessible
}
```

---

5. Block Scope

Definition:
Variables declared with let and const inside {} have block-level scope.

Example:

```
if (true) {
   let blockVar = "inside block";
   console.log(blockVar); // works
}
// console.log(blockVar);  Error: not accessible outside
```

---

6. Lexical Scope

Definition:
A function has access to the variables in its **outer scope**, even after the outer function has finished executing.

Example:

```
function outer() {
   let a = "outer";
   function inner() {
      console.log(a); // 'outer'
   }
   inner();
}
outer();
```

---

7. Function Expressions

Definition:
A function assigned to a variable. Can be named or anonymous.

Syntax:

```
const greet = function() {
   console.log("Hello!");
};
greet();
```

Note: Function expressions are **not hoisted** like function declarations.

---

8. Higher Order Functions

Definition:
A function that takes another function as an argument.

Use Case:
Used for abstracting actions like looping, filtering, mapping.

Example:

```
function callTwice(func) {
   func();
   func();
}

callTwice(() => console.log("Hi"));
```

---

9. Higher Order Functions (Returns)

Definition:
Functions that **return another function**.

Example:

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

const double = multiplier(2);
console.log(double(5)); // 10
```

---

10. Methods

Definition:
A method is a **function inside an object**.

Syntax:

```
const person = {
   name: "John",
   greet: function() {
      console.log("Hello, I'm " + this.name);
   }
};
person.greet();
```

1. try & catch

 Purpose:

Used to **handle errors gracefully** without breaking the program.

 Syntax:

```
try {
   // Code that may throw error
} catch (error) {
   // Handle the error
} finally {
   // (Optional) Runs always
}
```

 Example:

```
try {
   let result = 10 / 0;
   throw new Error("Custom error thrown");
} catch (e) {
   console.log("Error:", e.message);
} finally {
   console.log("Always runs");
}
```

---

 2. Arrow Functions =>

 Purpose:

A shorter syntax for writing functions.

 Syntax:

```
const add = (a, b) => {
   return a + b;
};
```

 Differences from regular functions:

- Doesn't have its own this
- Can have **implicit return**
- Can't be used as constructors

---

3. Implicit Return in Arrow Functions

 Purpose:

Return a value **without using the return keyword or curly braces {}**

 Syntax:

```
const multiply = (x, y) => x * y;
const greet = name => `Hello, ${name}!`;
```

 Note:

Implicit return only works if there's **no block body {}**.

---

4. setTimeout Function

 Purpose:

Execute a function **once after a specified delay** (in milliseconds).

 Syntax:

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

---

5. setInterval Function

 Purpose:

Execute a function **repeatedly at specified intervals**.

 Syntax:

```
let count = 0;
const id = setInterval(() => {
   console.log("Running:", ++count);
   if (count >= 5) clearInterval(id); // stop after 5 runs
}, 1000);
```

---

6. this Keyword

 Purpose:

this refers to the **current execution context**.

 Context Variations:

Context	this refers to
Global (non-strict)	window (in browser)
Regular function (non-method)	window or undefined
Inside method	The calling object
Arrow function	Lexical (outer) this
Constructor function	The newly created object

 Examples:

```
function normal() {
   console.log(this); // global
}

const obj = {
   name: "JS",
   regular: function () {
      console.log(this.name); // JS
   },
   arrow: () => {
      console.log(this.name); // undefined (inherits outer scope)
   }
};

obj.regular();
obj.arrow();
```

---

7. this with Arrow Functions

 Behavior:

Arrow functions **don’t have their own this**. They use the this from the **surrounding lexical context**.

 Use Case Example:

```
const person = {
   name: "Prathamesh",
   show: function () {
      setTimeout(() => {
         console.log("Arrow this:", this.name); // works
      }, 1000);
   }
};
person.show();
```

1. Array Methods

 Common Methods:

- push() – Add to end
- pop() – Remove from end
- shift() – Remove from start
- unshift() – Add to start
- includes() – Checks if element exists
- indexOf() – Finds index
- join() – Combines array to string
- slice(start, end) – Extracts section
- splice(start, deleteCount, ...items) – Changes array

 Example:

```
const fruits = ["apple", "banana"];
fruits.push("mango"); // ["apple", "banana", "mango"]
```

---

2. Map & Filter

 map() – Transforms each element

 filter() – Filters elements by condition

 Syntax:

```
arr.map(callback)
```

arr.filter(callback)

 Example:

```
const nums = [1, 2, 3];
const squares = nums.map(n => n * n);      // [1, 4, 9]
const evens = nums.filter(n => n % 2 === 0); // [2]
```

---

3. Every & Some

 every() – Returns true if all elements pass

 some() – Returns true if at least one passes

 Example:

```
const allEven = [2, 4, 6].every(n => n % 2 === 0); // true
const hasOdd = [2, 4, 5].some(n => n % 2 !== 0);   // true
```

---

4. Reduce Method

 Combines array values into single result

 Syntax:

```
arr.reduce((accumulator, current) => ..., initialValue)
```

 Example:

```
const sum = [1, 2, 3].reduce((acc, curr) => acc + curr, 0); // 6
```

---

5. Maximum in Array

 Use spread with Math.max

 Example:

```
const arr = [10, 20, 50];
const max = Math.max(...arr); // 50
```

---

6. Default Parameters

 Assign default value if no argument is passed

 Example:

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

greet(); // Hello, Guest
```

---

7. Spread Operator (...)

 Spreads elements of array/object

---

a) Spread in Arrays

```
const a = [1, 2];
const b = [...a, 3, 4]; // [1, 2, 3, 4]
```

b) Spread in Objects

```
const obj1 = { a: 1 };
const obj2 = { ...obj1, b: 2 }; // { a: 1, b: 2 }
```

---

8. Rest Parameters

 Collects all remaining args into an array

 Example:

```
function sum(...nums) {
  return nums.reduce((a, b) => a + b);
}

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

---

9. Destructuring

 Extract values from arrays or objects

---

a) Array Destructuring

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

b) Object Destructuring

```
const user = { name: "Alice", age: 25 };
const { name, age } = user;
console.log(name); // Alice
```

JavaScript DOM Manipulation – Full Notes

---

1. What is DOM?

- **DOM** stands for **Document Object Model**.
- It is a **tree-like structure** representing the HTML elements of a webpage.
- The browser creates a **DOM object** of the page, which JavaScript can interact with.
- Every HTML element becomes a **node** in this tree.

Example:

```
<html>
  └── <body>
      ├── <h1>
      └── <p>
```

With the DOM, JavaScript can **read**, **modify**, **add**, and **delete** HTML elements and their attributes.

---

2. Selecting Elements by ID

- Syntax: document.getElementById("id")
- Returns **one** element (the first with the matching ID).
- Useful for directly targeting a unique element.

```
const title = document.getElementById("main-title");
title.style.color = "blue";
```

---

3. Selecting Elements by Class Name

- Syntax: document.getElementsByClassName("className")
- Returns a **live HTMLCollection** of all matching elements.
- Need to loop over them to manipulate.

```
const items = document.getElementsByClassName("card");
for (let item of items) {
  item.style.border = "1px solid red";
}
```

---

4. Selecting Elements by Tag Name

- Syntax: document.getElementsByTagName("tagName")
- Returns a **live HTMLCollection** of all matching tag elements.

```
const paras = document.getElementsByTagName("p");
paras[0].textContent = "Changed paragraph!";
```

---

5. Query Selectors

- **querySelector()** – returns the **first** matching element (like CSS selectors).
- **querySelectorAll()** – returns a **static NodeList** of all matching elements.

```
const firstBox = document.querySelector(".box");
const allBoxes = document.querySelectorAll(".box");
```

- You can use any CSS selector: #id, .class, tag, etc.

---

6. Setting Content in Objects

- **.textContent** – plain text (ignores HTML tags).
- **.innerHTML** – sets HTML inside the element.

```
element.textContent = "New Text";       // Replaces text only
```

element.innerHTML = "Bold Text"; // Adds tags and renders them

Use textContent for security and performance, unless HTML is required.

---

7. Manipulating Attributes

- Use .getAttribute(), .setAttribute(), .removeAttribute().

```
const link = document.getElementById("myLink");
link.setAttribute("href", "https://openai.com");
link.getAttribute("href"); // returns current href
link.removeAttribute("target");
```

---

8. Manipulating Style (Inline via JS)

- Use the .style property to change individual CSS properties.

```
const div = document.getElementById("box");
div.style.backgroundColor = "lightblue";
div.style.borderRadius = "10px";
```

Use camelCase for multi-word CSS (e.g., backgroundColor, not background-color).

---

9. classList Property

- element.classList gives access to the class list of an element.
- Methods:
    - add(), remove(), toggle(), contains()

```
div.classList.add("active");
div.classList.remove("highlight");
div.classList.toggle("show");
if (div.classList.contains("error")) {
  alert("Has error class");
}
```

---

10. Navigation on Page (DOM Tree Navigation)

- Navigate between elements using:

Method	Description
parentElement	Get parent node
children	Get all child elements
firstElementChild	First child
lastElementChild	Last child
nextElementSibling	Next sibling node
previousElementSibling	Previous sibling node

```
const para = document.getElementById("para1");
console.log(para.parentElement);
console.log(para.nextElementSibling);
```

---

11. Adding Elements on the Page

Steps:

1. createElement(tagName)
2. Modify the element (content, classes, styles)
3. appendChild() or insertBefore()

```
const newDiv = document.createElement("div");
newDiv.textContent = "Hello";
newDiv.className = "box";
document.body.appendChild(newDiv);
```

Use append() for modern usage (can insert multiple nodes or strings).

---

12. Removing Elements from the Page

- Use removeChild() on parent
- Or element.remove() for modern browsers

```
const container = document.getElementById("container");
const last = container.lastElementChild;
if (last) {
  container.removeChild(last);
}

// Or simply:
document.getElementById("myDiv").remove();
```

□ Bonus Tips

- DOM changes are reflected **live** (except in querySelectorAll, which is static).
- Prefer querySelector for flexibility.
- Minimize innerHTML for security (avoid XSS).
- Use classList over .className += for cleaner code.

1. DOM Events

What are DOM Events?

DOM events are actions that occur as a result of user interaction or browser activity (e.g., clicks, form submissions, page load, keyboard presses, etc.).

Examples of Events:

- click – Mouse click
- submit – Form submission
- keydown, keyup – Keyboard keys
- load – Page load
- change, input, focus, blur – Form events

Event Syntax (Inline)

<button onclick="sayHello()">Click</button>

---

2. Mouse / Pointer Events

Mouse Events:

- click – Mouse click
- dblclick – Double-click
- mousedown / mouseup – Press / release
- mouseover / mouseout – Hover in / out
- mousemove – Moving the mouse

Pointer Events (work with mouse, touch, pen):

- pointerdown, pointerup, pointermove, pointerenter, pointerleave

Example:

```
element.addEventListener("mouseover", () => {
  console.log("Mouse is over the element");
});
```

---

3. Event Listeners

What are Event Listeners?
An event listener waits for a specific event to occur and then runs a callback function.

Syntax:

```
element.addEventListener("event", callback);
```

Example:

```
document.getElementById("btn").addEventListener("click", function() {
  alert("Button Clicked!");
});
```

 Use removeEventListener() to detach an event.

---

4. Event Listeners for Multiple Elements

You can attach the same listener to multiple elements using querySelectorAll().

Example:

```
const buttons = document.querySelectorAll('.btn');
buttons.forEach(btn => {
  btn.addEventListener('click', () => {
    console.log(btn.textContent + ' clicked');
  });
});
```

 Useful in list rendering or handling multiple buttons.

---

5. this in Event Listeners

- In a regular function, this refers to the element that fired the event.
- In arrow functions, this is lexically bound (from parent scope) — so avoid using arrow functions when you need this to refer to the element.

Example:

```
document.querySelector('.btn').addEventListener('click', function() {
  this.style.backgroundColor = 'yellow';
});
```

 Don’t use arrow functions if you want to use this as the event target.

---

6. Keyboard Events

Types:

- keydown: Fires when key is pressed down

- keypress: Deprecated (similar to keydown but only for characters)
- keyup: Fires when key is released

Example:

```
document.addEventListener('keydown', function(event) {
  console.log("Key Pressed: " + event.key);
});
```

 event.key gives the actual character (a, Enter, etc.)

---

7. Form Events

Common Form Events:

- submit: When a form is submitted
- change: When input value changes (used with <select> and radio buttons)
- focus: When an input is focused
- blur: When an input loses focus

Example:

```
document.getElementById('form').addEventListener('submit', function(e) {
  e.preventDefault(); // prevent page reload
  alert("Form submitted!");
});
```

---

8. Extracting Form Data

Using FormData API:

```
const form = document.querySelector('form');
form.addEventListener('submit', function(e) {
  e.preventDefault();
  const formData = new FormData(form);
  const username = formData.get('username');
  console.log("Username:", username);
});
```

 You can also use form.elements to access inputs by name or ID.

---

9. More Events

Other Events Worth Knowing:

Event	Description
input	Fires whenever user types or edits
focus	Fires when an input is focused
blur	Fires when focus is lost
change	Fires when input value changes (on blur)
scroll	Fires when user scrolls the page
resize	Fires when window is resized
contextmenu	Right-click event

Example:

```
input.addEventListener('input', () => {
  console.log("Input changed to: " + input.value);
});
```

JavaScript Asynchronous Concepts & Debugging – Notes

---

1. JS Call Stack

What is it?

- A **data structure** that tracks function calls.
- Follows **LIFO** (Last In, First Out) order.
- When a function is invoked, it's **pushed** onto the stack. Once completed, it's **popped** off.

Example:

```
function a() { b(); }
function b() { c(); }
function c() { console.log("Hello"); }
a();
```

Output:

Hello

Visualization:

Call Stack:
-> a()
-> b()
-> c()
-> console.log()

---

2. Visualizing the Call Stack

- Use **Chrome DevTools** or **Firefox Debugger**.
- When an error occurs, the console shows the **stack trace**.
- Helps understand **function execution order**.

Example:

```
function x() {
  y();
}
function y() {
  z();
}
function z() {
  throw new Error("Something went wrong!");
}
x();
```

 View the stack trace in console to debug deeply nested errors.

---

3. Breakpoints

What is it?

- Used to **pause execution** at a specific line.
- Helpful in step-by-step debugging.
- Set using **browser DevTools**.

How to use?

1. Open browser DevTools (F12).
2. Go to **Sources** tab.
3. Click line number to add a **blue breakpoint dot**.
4. Reload and watch execution pause.

---

4. JS is Single Threaded

- JavaScript executes code **line by line** (one thread).
- There is **only one call stack**.
- Cannot execute multiple scripts **in parallel**.

But how does it handle async operations?

Via:

- **Web APIs** (like setTimeout)
- **Callback Queue**
- **Event Loop**

---

5. Callback Hell

What is it?

- When callbacks are nested inside other callbacks, making code **hard to read** and **maintain**.

Example:

```
setTimeout(() => {
  console.log("1");
```

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

 Problem: Increases complexity, hard to debug.

---

6. Setting up for Promises

What is a Promise?

- An object representing the **eventual result** of an async operation.
- States:
  - **Pending**
  - **Fulfilled**
  - **Rejected**

Syntax:

```
const myPromise = new Promise((resolve, reject) => {
   if (true) resolve("Success");
   else reject("Error");
});
```

---

7. Refactoring with Promises

Why?

To avoid callback hell by using **flat** and **chained** structure.

Example:

```
function wait(msg, delay) {
  return new Promise(resolve => {
    setTimeout(() => resolve(msg), delay);
  });
```

```
}

wait("Step 1", 1000)
  .then(result => {
    console.log(result);
    return wait("Step 2", 1000);
  })
  .then(result => console.log(result));
```

---

8. then() and catch() Methods

.then()

- Executes on **resolve**
- Can be chained.

.catch()

- Executes on **reject**
- Catches any error in the promise chain.

Example:

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

---

9. Promise Chaining

- Sequence of .then() calls.
- Each .then() returns a new promise.

Example:

```
doTask()
  .then(res1 => {
    console.log(res1);
    return nextTask();
  })
  .then(res2 => console.log(res2))
```

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

 Allows better **asynchronous flow control**.

---

10. Results & Errors in Promises

Handling:

- Use .catch() for promise rejections.
- Avoid unhandled rejections (can crash apps).
- You can use try-catch in async/await too.

Example:

```
const riskyPromise = new Promise((resolve, reject) => {
   const success = Math.random() > 0.5;
   success ? resolve("Yay!") : reject("Oops!");
});

riskyPromise
   .then(result => console.log("Success:", result))
   .catch(error => console.error("Failed:", error));
```

Bonus: Async/Await (Modern Way)

```
async function fetchData() {
   try {
      const data = await wait("Data loaded", 1000);
      console.log(data);
   } catch (err) {
      console.error("Error:", err);
   }
}
```

JavaScript Asynchronous & API Concepts –

---

1. Async Functions

Definition:
async functions are special functions that return a Promise. They allow us to write asynchronous code in a more synchronous-like fashion.

Syntax:

```
async function myFunction() {
   return "Hello";
}
```

Key Points:

- Always returns a Promise.
- If you return a value, JavaScript automatically wraps it in a Promise.
- You can use await inside async functions only.

---

2. Await Keyword

Definition:
await pauses the execution of the async function until the Promise is resolved or rejected.

Syntax:

```
async function fetchData() {
   let response = await fetch('https://api.example.com/data');
   let data = await response.json();
   console.log(data);
}
```

Key Points:

- Can only be used inside async functions.

- It makes asynchronous code look synchronous and easier to read.

---

3. Handling Rejections (Errors)

How to handle? Use try...catch inside async functions or .catch() with Promises.

```
async function fetchData() {
   try {
      const res = await fetch('wrong_url');
      const data = await res.json();
      console.log(data);
   } catch (err) {
      console.error("Error occurred:", err);
   }
}
```

Alternative:

```
fetch('wrong_url')
  .then(res => res.json())
  .catch(err => console.log("Error:", err));
```

---

4. What is an API?

Definition:
API = **Application Programming Interface**
It's a bridge that allows two software programs to communicate with each other.

Example:
Google Maps API, Weather API, OpenAI API.

Types:

- REST API (most common)
- GraphQL
- SOAP

---

5. Accessing Some APIs

Using fetch() method:

```
fetch('https://api.agify.io/?name=prathamesh')
  .then(response => response.json())
  .then(data => console.log(data));
```

Note: Always check documentation for endpoint format.

---

6. What is JSON?

Definition:
JSON = JavaScript Object Notation
It’s a lightweight data format used for storing and exchanging data.

```
{
  "name": "John",
  "age": 25,
  "skills": ["HTML", "CSS", "JS"]
}
```

Conversion in JS:

```
const obj = JSON.parse(jsonData); // Convert JSON to Object
const json = JSON.stringify(obj); // Convert Object to JSON
```

---

7. Accessing JSON Data

Given:

```
const data = {
  user: {
    name: "Alice",
    age: 22
  }
};
```

Access:

```
console.log(data.user.name); // Alice
```

Loop Through Array:

```
const users = [{name: "A"}, {name: "B"}];
users.forEach(user => console.log(user.name));
```

---

8. API Testing Tools

Tool	Description
Postman	GUI tool to test APIs easily
Insomnia	Lightweight API testing tool
curl	Command-line tool for HTTP requests
Thunder Client	VS Code extension for API testing

---

9. What is AJAX?

AJAX = Asynchronous JavaScript and XML
Used to fetch or send data to a server **without refreshing the page**.

Old way (XMLHttpRequest):

```
const xhr = new XMLHttpRequest();
xhr.open("GET", "https://api.example.com");
xhr.onload = () => console.log(xhr.responseText);
xhr.send();
```

Modern way (fetch):

```
fetch('https://api.example.com')
  .then(res => res.json())
  .then(data => console.log(data));
```

---

10. HTTP Verbs

Verb	Purpose
GET	Retrieve data
POST	Create data
PUT	Update data
DELETE	Delete data

---

11. Status Codes

Code	Meaning
200	OK
201	Created
400	Bad Request
401	Unauthorized
403	Forbidden
404	Not Found
500	Internal Server Error

---

12. Adding Information in URLs

Using Query Parameters:

```
const name = "michael";
fetch(`https://api.agify.io/?name=${name}`)
  .then(res => res.json())
```

```
  .then(data => console.log(data));
```

---

13. HTTP Headers

Headers provide additional information to server/client:

Examples:

```
headers: {
  "Content-Type": "application/json",
  "Authorization": "Bearer token"
}
```

---

14. Our First API Request

```
fetch("https://api.coindesk.com/v1/bpi/currentprice.json")
  .then(res => res.json())
  .then(data => console.log("Bitcoin Price:", data.bpi.USD.rate));
```

---

15. Using Fetch with Async-Await

```
async function getPost() {
  try {
    const res = await fetch("https://jsonplaceholder.typicode.com/posts/1");
    const data = await res.json();
    console.log("Post Title:", data.title);
  } catch (err) {
    console.error("Fetch error:", err);
  }
}

getPost();
```