Javascript

Html dom methods:-

getElementById()

const elements = document.getElementsByClassName('myClass');

const elements = document.getElementsByTagName('div');

const element = document.querySelector('.myClass');

const elements = document.querySelectorAll('.myClass');

** JavaScript can change HTML content. **

<button type="button" onclick='document.getElementById("demo").innerHTML = "Hello JavaScript!"'>Click Me!</button>

<button onclick="document.getElementById('myImage').src='pic_bulbon.gif'">Turn on the light</button>

<img id="myImage" src="pic_bulboff.gif" style="width:100px">

<button onclick="document.getElementById('myImage').src='pic_bulboff.gif'">Turn off the light</button>

** JavaScript can change the style of an HTML element. **

<button type="button" onclick="document.getElementById('demo').style.backgroundColor='red'">Click Me!</button>

Script tag

Just before the closing </body> tag: Placing <script> tags just before the closing </body> tag allows the page to load and render first, then execute the scripts. This can lead to better performance and user experience since the page content is visible to the user while the scripts load. This placement is generally

preferred for most scripts, especially if they involve heavy computations or network requests.

Javascript output

Writing into an HTML element, using innerHTML.

Writing into the HTML output using document.write().

Writing into an alert box, using window.alert().

Writing into the browser console, using console.log().

Using document.write() after an HTML document is loaded, will delete all existing HTML:

Const

A variable defined with the const keyword cannot be reassigned:

JavaScript has 8 Datatypes

1. String

2. Number

3. Bigint

4. Boolean

5. Undefined

6. Null

7. Symbol

8. Object

The Object Datatype

The object data type can contain:

1. An object

2. An array

3. A date

Reference Types:

In JavaScript, reference types refer to data types that are complex data structures stored in memory by reference rather than by value. This means when you work with a reference type, you are dealing with a reference (or pointer) to the data in memory rather than the data itself.

Extracting String Parts

There are 3 methods for extracting a part of a string:

slice(start, end)

substring(start, end)

substr(start, length)

The difference is that start and end values less than 0 are treated

as 0 in substring().

A string is converted to upper case with toUpperCase():

A string is converted to lower case with toLowerCase():

concat() joins two or more strings:

let text1 = "Hello";

let text2 = "World";

let text3 = text1.concat(" ", text2);

The trim() method removes whitespace from both sides of a string:

let text1 = " Hello World! ";

let text2 = text1.trim();

The trimStart() method works like trim(), but removes whitespace only from the start of a string.

The trimEnd() method works like trim(), but removes whitespace only from the end of a string.

The **padStart() method** pads a string from the start.

It pads a string with another string (multiple times) until it reaches a given length.

let text = "5";

let padded = text.padStart(4,"0");

The padEnd() method pads a string from the end.

It pads a string with another string (multiple times) until it reaches a given length.

let text = "5";

let padded = text.padEnd(4,"0");

The **repeat() method** returns a string with a number of copies of a string.

The **repeat() method** returns a new string.

let text = "Hello world!";

let result = text.repeat(2);

The replace() method replaces a specified value with another value in a string:

let text = "Please visit Microsoft!";

let newText = text.replace("Microsoft", "W3Schools");

----

let text = "Please visit Microsoft!";

let newText = text**.replace(/**MICROSOFT/i, "W3Schools");

---

let text = "Please visit Microsoft and Microsoft!";

let newText = text**.replace(/**Microsoft/g, "W3Schools");

---

text = text**.replaceAll("**Cats","Dogs");

text = text**.replaceAll("**cats","dogs");

---

text = text**.replaceAll(/**Cats/g,"Dogs");

text = text**.replaceAll(/**cats/g,"dogs");

---

text**.split**(",") // Split on commas

text**.split**(" ") // Split on spaces

text**.split("|")** // Split on pipe

---

let text = "Please locate where 'locate' occurs!";

let index = text**.indexOf**("locate");

---

let text = "Please locate where 'locate' occurs!";

let index = text**.lastIndexOf("**locate");

Both indexOf(), and lastIndexOf() return -1 if the text is not found

The lastIndexOf() methods searches backwards (from the end to the beginning), meaning: if the second parameter is 15, the search starts at position 15, and searches to the beginning of the string.

let text = "Please locate where 'locate' occurs!";

text**.lastIndexOf("**locate", 15);

The search() method searches a string for a string (or a regular expression) and returns the position of the match:

let text = "Please locate where 'locate' occurs!";

text.**search**("locate");

let text = "The rain in SPAIN stays mainly in the plain";

text**.match**(/ain/);

let text = "The rain in SPAIN stays mainly in the plain";

text**.match**(/ain/g);

let text = "The rain in SPAIN stays mainly in the plain";

text**.match**(/ain/gi);

const iterator = text**.matchAll**("Cats");

const iterator = text.**matchAll(/**Cats/g);

```
const iterator = text.matchAll(/Cats/gi);

let text = "Hello world, welcome to the universe.";

text.includes("world");

let text = "Hello world, welcome to the universe.";

text.includes("world", 12);

let text = "Hello world, welcome to the universe.";

text.startsWith("Hello");

let text = "Hello world, welcome to the universe.";

text.startsWith("world", 5)

let text = "John Doe";

text.endsWith("Doe");

let text = "Hello world, welcome to the universe.";

text.endsWith("world", 11);
```

string interpolation:-

```
const cars = new Array("Saab", "Volvo", "BMW");

cars.length   // Returns the number of elements

cars.sort()   // Sorts the array

fruits.push("Lemon");  // Adds a new element (Lemon) to fruits

fruits[fruits.length] = "Lemon";

Array.isArray(fruits);

const fruits = ["Banana", "Orange", "Apple"];

fruits instanceof Array;
```

JavaScript Array Methods

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
```

let size = fruits.**length**;

const fruits = ["Banana", "Orange", "Apple", "Mango"];

document.getElementById("demo").innerHTML = fruits.**toString**();

const fruits = ["Banana", "Orange", "Apple", "Mango"];

let fruit = fruits.**at**(2);

The **join**() method also joins all array elements into a string.

The **pop() method** removes the last element from an array:

The **pop**() method returns the value that was "popped out":

The **push() method** adds a new element to an array (at the end):

The **push() method** returns the new array length:

The **shift()** method removes the first array element and "shifts" all other elements to a lower index.

The **shift() method** returns the value that was "shifted out":

The **unshift() method** adds a new element to an array (at the beginning), and "unshifts" older elements:

The **unshift() method** returns the new array length:

Changing Elements

const fruits = ["Banana", "Orange", "Apple", "Mango"];

fruits[0] = "Kiwi";

const fruits = ["Banana", "Orange", "Apple", "Mango"];

fruits[fruits.length] = "Kiwi";

Using **delete()** leaves undefined holes in the array.

Use pop() or shift() instead.

const fruits = ["Banana", "Orange", "Apple", "Mango"];

delete fruits[0];

The **concat() method** creates a new array by merging (concatenating) existing arrays:

const myGirls = ["Cecilie", "Lone"];

const myBoys = ["Emil", "Tobias", "Linus"];

const myChildren = myGirls.**concat**(myBoys);

The **concat()** method does not change the existing arrays. It always returns a new array.

The **concat() method** can take any number of array arguments.

The **copyWithin() method copies** array elements to another position in an array:

const fruits = ["Banana", "Orange", "Apple", "Mango", "Kiwi"];

fruits.**copyWithin**(2, 0, 2);

The **copyWithin() method** overwrites the existing values.

The **copyWithin() method** does not add items to the array.

The **copyWithin() method** does not change the length of the array.

Flattening an array is the process of reducing the dimensionality of an array.

Flattening is useful when you want to convert a multi-dimensional array into a one-dimensional array.

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

const newArr = **myArr.flat();**

The splice() method can be used to add new items to an array:

const fruits = ["Banana", "Orange", "Apple", "Mango"];

fruits.**splice**(2, 0, "Lemon", "Kiwi");

you can use **splice()** to remove elements without leaving "holes" in the array:

const fruits = ["Banana", "Orange", "Apple", "Mango"];

fruits.**splice**(0, 1);

The difference between the new **toSpliced() method** and the old **splice() method** is that the new method creates a new array, keeping the original array unchanged, while the old method altered the original array.

const months = ["Jan", "Feb", "Mar", "Apr"];

const spliced = months.**toSpliced**(0, 1);

The **slice() method** slices out a piece of an array into a new array:

The **indexOf() method** searches an array for an element value and returns its position.

```
const fruits = ["Apple", "Orange", "Apple", "Mango"];
let position = fruits.indexOf("Apple")
```

returns -1 if the item is not found.

```
const fruits = ["Apple", "Orange", "Apple", "Mango"];
let position = fruits.lastIndexOf("Apple")
```

returns -1 if the item is not found.

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.includes("Mango"); // is true
```

return false if not found

The **find() method** returns the value of the first array element

that passes a test function.

```
const numbers = [4, 9, 16, 25, 29];
let first = numbers.find(myFunction);
function myFunction(value, index, array) {
  return value > 18;
}
```

The **findIndex() method** returns the index of the first array element that passes a test function.

```
const numbers = [4, 9, 16, 25, 29];
let first = numbers.findIndex(myFunction);
function myFunction(value, index, array) {
  return value > 18;
}
```

The findLastIndex() method finds the index of the last element

that satisfies a condition.

The **sort() method** sorts an array alphabetically:

const fruits = ["Banana", "Orange", "Apple", "Mango"];

fruits.sort();

The **reverse() method** reverses the elements in an array:

const fruits = ["Banana", "Orange", "Apple", "Mango"];

fruits.reverse();

The difference between **toSorted**() and **sort**() is that the first method creates a new array, keeping the original array unchanged, while the last method alters the original array.

const months = ["Jan", "Feb", "Mar", "Apr"];

const sorted = months.**toSorted**();

The difference between **toReversed**() and **reverse**() is that the first method creates a new array, keeping the original array unchanged, while the last method alters the original array.

const months = ["Jan", "Feb", "Mar", "Apr"];

const reversed = months.**toReversed**();

points.**sort**(function(a, b){return a - b});

to sort numbers

Math.min Math.min.apply(null, arr);

Math.max Math.max.apply(null, arr);

Array Iteration

The **forEach() method** calls a function (a callback function) once for each array element.

The **map() method** creates a new array by performing a function on each array element.

The **map() method** does not execute the function for array elements without values.

The **map() method** does not change the original array.

The **flatMap() method** first maps all elements of an array and then creates a new array by flattening the array.

The **filter() method** creates a new array with array elements that pass a test.

The **reduce() method** runs a function on each array element to produce (reduce it to) a single value.

The **reduce() method** works from left-to-right in the array.

The **reduce() method** can accept an initial value:

```
const numbers = [45, 4, 9, 16, 25];

let sum = numbers.reduce(myFunction, 100);

function myFunction(total, value) {
  return total + value;
}
```

The **reduceRight() method** runs a function on each array element to produce (reduce it to) a single value.

The **reduceRight**() works from right-to-left in the array

The **every() method checks** if all array values pass a test.

```
const numbers = [45, 4, 9, 16, 25];

let allOver18 = numbers.every(myFunction);

function myFunction(value, index, array) {
  return value > 18;
}
```

The **some() method** checks if some array values pass a test.

```
const numbers = [45, 4, 9, 16, 25];

let someOver18 = numbers.some(myFunction);

function myFunction(value, index, array) {
  return value > 18;
}
```

The Array.**from() method** returns an Array object from any object with a length property or any iterable object.

```
Array.from("ABCDEFG");
```

A,B,C,D,E,F,G

The **Array.keys() method** returns an Array Iterator object with the keys of an array.

const fruits = ["Banana", "Orange", "Apple", "Mango"];

const keys = fruits.**keys**();

Array entries()

Create an Array Iterator, and then iterate over the key/value pairs:

const fruits = ["Banana", "Orange", "Apple", "Mango"];

const f = fruits.**entries**();

0,Banana

1,Orange

2,Apple

3,Mango

with() method as a safe way to update elements in an array without altering the original array.

const months = ["Januar", "Februar", "Mar", "April"];

const myMonths = months.**with**(2, "March");

Array Spread (...)

The ... operator expands an iterable (like an array) into more elements:

JavaScript Date Objects

JavaScript Math Object

Math.round(x) Returns x rounded to its nearest integer

Math.ceil(x) Returns x rounded up to its nearest integer

Math.floor(x) Returns x rounded down to its nearest integer

Math.trunc(x) Returns the integer part of x (new in ES6)

Math.sign(x) returns if x is negative, null or positive:

Math.pow(x, y) returns the value of x to the power of y:

Math.sqrt(x) returns the square root of x:

Math.abs(x) returns the absolute (positive) value of x:

Math.min() and Math.max() can be used to find the lowest or highest value in a list of arguments:

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

JavaScript Hoisting

Hoisting is JavaScript's default behavior of moving declarations to the top.

In JavaScript, a variable can be declared after it has been used.

In other words; a variable can be used before it has been declared.

x = 5; // Assign 5 to x

elem = document.getElementById("demo"); // Find an element

elem.innerHTML = x; // Display x in the element

var x; // Declare x

Hoisting is JavaScript's default behavior of moving all declarations to the top of the current scope

Variables defined with let and const are hoisted to the top of the block, but not initialized.

Using a let variable before it is declared will result in a ReferenceError.

The variable is in a "temporal dead zone" from the start of the block until it is declared:

Using a const variable before it is declared, is a syntax error, so the code will simply not run.

JavaScript Use Strict

"use strict"; Defines that JavaScript code should be executed in "strict mode".

"use strict";

x = 3.14; // This will cause an error because x is not declared

Strict mode makes it easier to write "secure" JavaScript.

Strict mode changes previously accepted "bad syntax" into real errors.

The JavaScript this Keyword

In JavaScript, the this keyword refers to an object.

Which object depends on how this is being invoked (used or called).

JavaScript Callbacks

Var , let and const

//scope

var-having global scope

let block scoped

const block scoped

//shadowing

//illegal shadowing

let replaced by var

//hoisting

js code execution context - creation phase and execution phase

in creation phase it will create a window object and secondly it creates a memory heap and initalizes functions and variables and value as undefiend

let are hoisted at temporal deadzone - they are in the scope but thery are not being declared yet

Polyfill

A polyfill is a piece of code (usually JavaScript on the Web) used to provide modern functionality on older browsers that do not natively support it.

In older browsers, the following features require polyfill support by explicitly defining the functions:

Promises, Array.from, Array.includes, Array.of, Map, Set, Symbol, object.values, etc

IIFE

immediately invoked function expression

Closure

the ability of a funtion to access the values that are lexically out of its scope

Callback function

a function passed into another function as an argument

(Event Propagation) - Bubbling, Capturing, and Deligation

Debouncing and throttling

Promises

promises is an asynchronous code

js is a single threaded language,

promise basically represents the upcoming completion or failure of an asynchronous event

const sub= new Promise(resolve,reject)=>{}

using .then and .catch

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

if multiple promises, then run with then .then and finally catch for catching err

promise combinator:-

promise .all()', if one of these promises fail all of these fails

Prmisie.race()

promise.allSettled()

prmoise.any()

modern way of approaching promises async and await

Promises in JavaScript:

A Promise in JavaScript is an object representing the eventual completion (or failure) of an asynchronous operation and its resulting value. Promises provide a cleaner, more predictable way to handle asynchronous tasks compared to traditional callback-based approaches.

Pending: The Promise is waiting for the asynchronous operation to complete.

Fulfilled: The Promise's asynchronous operation has completed successfully, and it now holds a resolved value.

Rejected: The Promise's asynchronous operation failed, and it now holds a reason for rejection (an error).

```
function fetchData(url) {
   return new Promise((resolve, reject) => {
      // Simulate an asynchronous operation
      setTimeout(() => {
         // Simulating a successful operation
         if (url) {
            resolve(`Data from ${url}`);
         } else {
            reject(new Error('URL not provided'));
         }
      }, 1000);
   });
}


fetchData('https://api.example.com/data')
```

```
    .then((data) => {
        console.log('Success:', data);
    })
    .catch((error) => {
        console.error('Error:', error);
    });
==========================
function examplePromise() {
    return new Promise((resolve, reject) => {
        // Simulate an asynchronous operation
        setTimeout(() => {
            const success = Math.random() > 0.5; // Randomly decide if operation succeeds or fails
            if (success) {
                resolve('Operation succeeded');
            } else {
                reject(new Error('Operation failed'));
            }
        }, 1000);
    });
}

examplePromise()
    .then((value) => {
        console.log('Fulfilled:', value);
    })
```

```
  .catch((error) => {
    console.log('Rejected:', error);
  })
  .finally(() => {
    console.log('Promise completed');
  });
```

all dom access methods in js:-

In JavaScript, there are various methods available for accessing and manipulating elements in the DOM

document.getElementById(id):-

event bubbling and event capturing:-

explaining event bubbling and event capturing requires a clear understanding of the event propagation process within the DOM

Event Propagation in JavaScript:

When an event occurs in the DOM, such as a click or keypress, it can propagate through the DOM in two ways: **event bubbling and event capturing.** Both describe how an event travels through the DOM tree.

Event Bubbling:

Event bubbling is the process by which an event starts from the target element where the event occurred and then propagates upward through its parent elements in the DOM tree.

In event bubbling, an event handler attached to a parent element can capture events that occur on its children.

By default, events bubble up unless explicitly stopped. You can stop the event from bubbling further using **event.stopPropagation()** in the event handler.

Event Capturing:

Event capturing (also called event trickling or event delegation) is the process by which an event starts from the root of the DOM tree and travels downward to the target element.

In event capturing, an event handler attached to an ancestor element can capture events as they travel down to the target element.

To use event capturing, you specify true as the third argument in addEventListener(). This tells the event

listener to capture events during the capturing phase.

The **event.stopImmediatePropagation()** method is used in JavaScript event handling to stop the further propagation of an event in the DOM, just like e**vent.stopPropagation().** However, it also prevents any other event listeners on the same event from being called.

```
// Add multiple click event listeners to the same element

document.querySelector('#myButton').addEventListener('click', function(event) {

  console.log('First event handler');

  // Stop further propagation and prevent other event listeners from executing

  event.stopImmediatePropagation();

});

document.querySelector('#myButton').addEventListener('click', function() {

  console.log('Second event handler');

});

document.querySelector('#parentElement').addEventListener('click', function() {

  console.log('Parent element clicked');

});

// If you click on the button element with ID 'myButton', only 'First event handler' will log

// because stopImmediatePropagation prevents further event propagation and other event listeners on the button and its parent.
```

event.preventDefault():-

In JavaScript, the event.preventDefault() method is used to cancel the default action associated with an event. This allows you to prevent the browser from performing its default behavior when an event occurs.

```
const form = document.querySelector('#myForm');

form.addEventListener('submit', function(event) {

  event.preventDefault(); // Prevent default form submission

  // Handle form submission with JavaScript (e.g., make an AJAX request)
```

});

why do we need react:-

Why We Need React:

Component-Based Architecture:

React uses a component-based architecture that allows developers to create reusable, self-contained components.

This approach simplifies the process of building complex UIs by breaking them down into smaller, manageable components.

Declarative Programming:

React uses a declarative programming model, which means you define what you want the UI to look like based on the current state and react takes care of rendering the UI.

This makes the code more predictable and easier to understand.

Virtual DOM:

React uses a Virtual DOM to optimize updates to the UI.

When the state changes, react calculates the minimal set of changes needed to update the actual DOM, which improves performance.

One-Way Data Flow:

React promotes a one-way data flow architecture, which makes state management more predictable and easier to debug.

This approach can help prevent bugs and simplify the development of complex applications.

Ease of Maintenance and Scalability:

React’s component-based architecture, along with its state management patterns, makes applications easier to maintain and scale as they grow.

Cross-Platform Development:

React's capabilities are not limited to web development. Frameworks like React Native allow developers to create cross-platform mobile applications using React.

Why Can't Everything Be Done in JavaScript Alone:

JavaScript's Limitations:

While JavaScript is versatile and capable of creating dynamic UIs, it doesn't provide built-in solutions for many complex UI challenges such as efficient re-rendering, state management, and data flow patterns.

Using just JavaScript, developers would need to implement many features from scratch, increasing complexity and potential for errors.

Development Speed and Efficiency:

React's abstractions and patterns simplify development, allowing developers to focus on the business logic and UI design rather than on lower-level DOM manipulations.

This can result in faster development and easier maintenance.

Modern UI Development:

Modern web applications often require dynamic, responsive, and interactive user interfaces.

Handling complex UIs with JavaScript alone can become cumbersome and error-prone.

While creating custom components in traditional JavaScript is possible, it requires careful planning and can become complex quickly as application complexity grows. Modern frameworks and libraries like React provide built-in support and abstractions for creating custom components, making development faster, more maintainable, and efficient.

what are the features of es6:-

when discussing the features of ECMAScript 2015 (commonly referred to as ES6)

let and const:

let: Introduces block-scoped variables, allowing you to declare variables that are accessible only within the block in which they are defined.

const: Declares block-scoped constants, which cannot be reassigned after declaration

Arrow Functions:

Template Literals:

Destructuring Assignment:

```
const [a, b] = [1, 2];

const { x, y } = { x: 10, y: 20 };
```

Default Parameters:

```
function greet(name = 'Guest') {
```

```
    console.log(`Hello, ${name}!`);
}
```

Rest and Spread Operators:

```
function sum(...args) {
    return args.reduce((a, b) => a + b, 0);
}
const arr = [1, 2, 3];
console.log(...arr); // Outputs: 1 2 3
```

Modules:

```
// Exporting a function
export function add(a, b) {
    return a + b;
}
// Importing a function
import { add } from './math.js';
```

Classes:

```
class Person {
    constructor(name) {
        this.name = name;
    }
    greet() {
        console.log(`Hello, ${this.name}!`);
    }
}
```

Promises:

Generators:

Functions that can be paused and resumed, allowing you to generate a sequence of values over time.

Useful for lazy evaluation and working with asynchronous data streams.

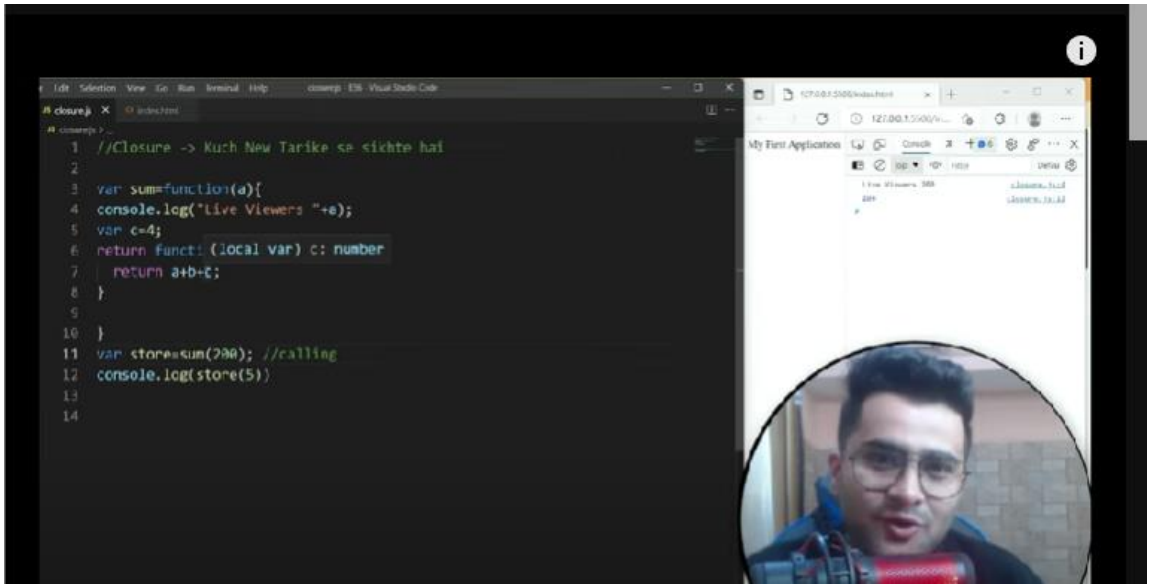
```
function* counter() {
   yield 1;
   yield 2;
   yield 3;
}
```

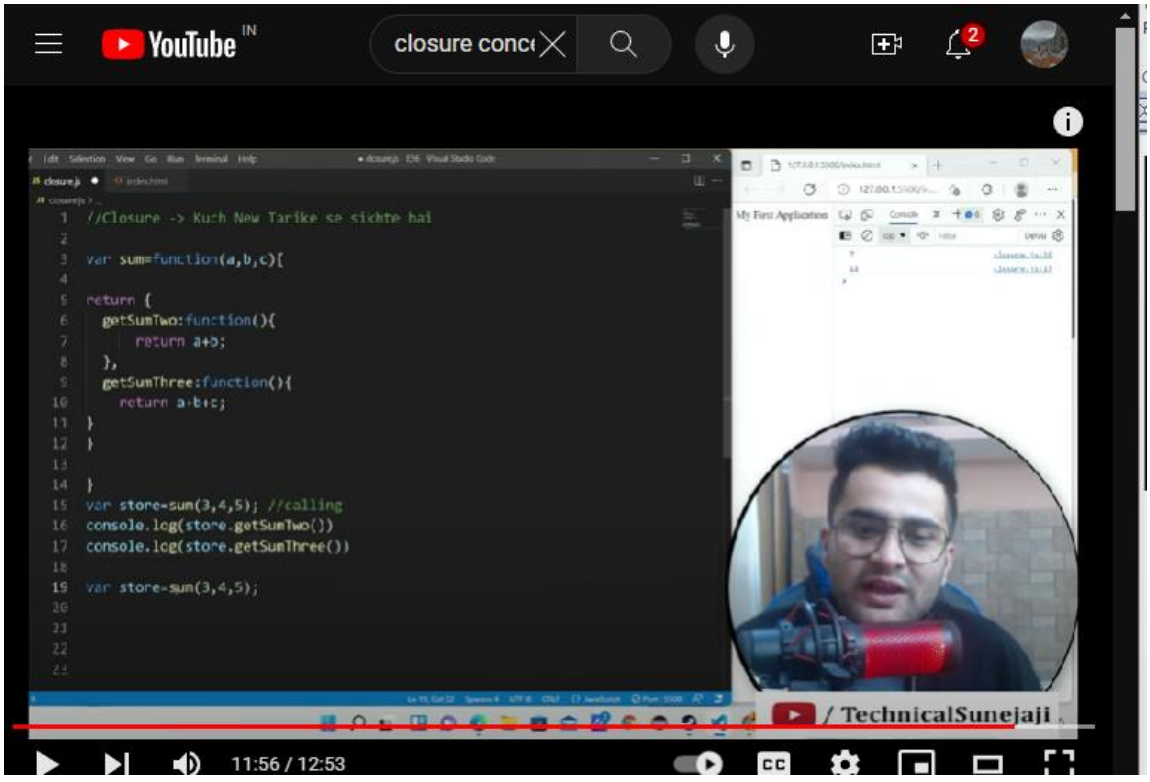
swap array values using array destructuring:-

```
const arr = [1, 2];
console.log('Before swap:', arr); // Outputs: [1, 2]

// Destructure the array and swap the values
[arr[0], arr[1]] = [arr[1], arr[0]];

console.log('After swap:', arr); // Outputs: [2, 1]
```





Shallow Copy:

A shallow copy creates a new object or array, but it only copies the top-level properties.

Nested objects or arrays are not fully copied; instead, references to the nested structures are maintained.

Using Spread Operator

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

const shallowCopyArray = [...originalArray];

// Both originalArray and shallowCopyArray reference the same nested array

shallowCopyArray[2][0] = 99;

console.log(originalArray[2][0]); // Output: 99
```

Using Object.assign()

```
const originalObject = { a: 1, b: { c: 2 } };

const shallowCopyObject = Object.assign({}, originalObject);


// Both originalObject and shallowCopyObject reference the same nested object

shallowCopyObject.b.c = 99;

console.log(originalObject.b.c); // Output: 99
```

Deep Copy:

A deep copy creates a new object or array with a completely new structure, including all nested objects and arrays.

Nested structures are copied recursively, so changes to the nested structures in the copy do not affect the original.

Using JSON methods

```
const originalObject = { a: 1, b: { c: 2 } };

const deepCopyObject = JSON.parse(JSON.stringify(originalObject));

// Nested objects are completely separate

deepCopyObject.b.c = 99;

console.log(originalObject.b.c); // Output: 2
```

Using libraries

Libraries like Lodash provide utility functions

(_.cloneDeep) to create deep copies:

```
const _ = require('lodash');
const originalObject = { a: 1, b: { c: 2 } };
const deepCopyObject = _.cloneDeep(originalObject);
deepCopyObject.b.c = 99;
console.log(originalObject.b.c); // Output: 2
```

Shallow copies are faster to create because they only copy top-level properties.

Deep copies take more time and memory because they duplicate the entire structure.

Obj.freeze

```
const nestedObj = {
    outer: { inner: 42 },
};

// Freeze the top-level object
Object.freeze(nestedObj);

// The nested object is still mutable
nestedObj.outer.inner = 99; // This will work

// To prevent modifications to nested objects, freeze them as well
Object.freeze(nestedObj.outer);

// Now this will fail
```

nestedObj.outer.inner = 100; // Throws a TypeError in strict mode or silently fails in non-strict mode

const with primitive and non primitive values:-

const with Primitive Values:-

When you declare a variable using `const` with a primitive value, the value itself cannot be changed.

const with Non-Primitive Values:-

When you declare a variable using `const` with a non-primitive value (such as an object or array), the variable cannot be reassigned to a new value. However, you can still modify the properties or elements of the object or array.

```
const person = { name: 'John' };

person.name = 'Jane'; // This is allowed

person.age = 30; // This is also allowed
```

Reason:-

Memory Address vs. Value

When you declare a variable with `const`, you are essentially creating a constant memory address that points to the value.

For primitive values, the value itself is stored directly in memory, so it cannot be changed.

For non-primitive values, the constant memory address still points to the same object or array, so you can modify the properties or elements of the object or array without changing the memory address itself.

Immutability vs. Reassignment:

- `const` provides immutability for primitive values (the value cannot be changed).
- For non-primitive values, `const` prevents reassignment of the variable itself but does not prevent changes to the properties or elements of the value it points to.

map vs filter:-

`map()`:

`map()` is used to transform each element of an array into a new value based on a provided function.

It creates a new array with the same length as the original array, where each element is the result of applying the provided function to the corresponding element of the original array.

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

const doubledNumbers = numbers.map(num => num * 2);

// doubledNumbers: [2, 4, 6, 8]

`filter()`:

`filter()` is used to create a new array with only the elements that pass a certain condition specified by a provided function.

It returns a new array containing only the elements for which the provided function returns `true`.

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

const evenNumbers = numbers.filter(num => num % 2 === 0);

// evenNumbers: [2, 4]

Arrow functions:-

Arrow functions are a concise syntax for writing JavaScript functions, introduced in ECMAScript 6 (ES6). They provide several advantages over traditional function expressions. Here's an overview of arrow functions and their advantages:

Differences between Spread and Rest Operator

Spread Operator expands an iterable into its individual elements.
Rest Operator collects multiple elements and condenses them into a single array.

```
const array1 = [1, 2, 3];

const array2 = [...array1, 4, 5]; // [1, 2, 3, 4, 5]
```

```
function sum(...numbers) {
  return numbers.reduce((acc, number) => acc + number, 0);
console.log(sum(1, 2, 3, 4, 5)); // 15
```

type coercion:-

Type coercion is the automatic conversion of one data type to another data type in JavaScript. This conversion happens implicitly when operators or functions expect operands of a certain type.

Implicit vs. Explicit Coercion:

Explicit Coercion

Developers can explicitly convert values from one type to another using built-in functions like `Number()`, `String()`, `Boolean()`, etc.

Explicit coercion is done intentionally by the developer to ensure correct behavior in certain situations.

Developers can explicitly convert values from one type to another using built-in functions like `Number()`, `String()`, `Boolean()`, etc.

Implicit Coercion

JavaScript automatically converts values from one type to another when needed.

It occurs during operations where operands are of different types, such as addition (`+`), comparison (`==`), or logical operations.

```
const num = 10;
const str = "Number: " + num; // Implicitly converts num to a string
```

```
const num = 10;
const str = "10";
```

```
console.log(num == str); // true (implicit conversion of str to a number)


console.log(1 == true); // true (implicit conversion of true to 1)
console.log(0 == false); // true (implicit conversion of false to 0)
```

Higher-Order Functions:

Higher-order functions are functions that can take other functions as arguments or return functions as results.

```
// Higher-order function
function map(array, transformFn) {
  const result = [];
  for (const element of array) {
    result.push(transformFn(element));
  }
  return result;
}
// Usage
const numbers = [1, 2, 3, 4, 5];
const squaredNumbers = map(numbers, num => num * num);
console.log(squaredNumbers); // Output: [1, 4, 9, 16, 25]
```

global execution:-

Global execution refers to the initial phase of JavaScript code execution when the script starts running.

Creation of Global Execution Context

When a JavaScript file or script is loaded, a global execution context is created. This context

represents the global scope of the script.

Variable and Function Declarations

During global execution, JavaScript hoists variable declarations (`var`) and function declarations to the top of their respective scopes.

This means that variables and functions can be accessed and used anywhere within their scope, even before they are declared in the code.

Execution of Code

After hoisting, the JavaScript engine starts executing the code line by line, following the order of statements in the script.

It assigns values to variables, executes function calls, and performs other operations as instructed by the code.

Creation of Execution Contexts for Functions

When a function is called during global execution, a new execution context is created for that function.

This process includes creating a local scope for the function, initializing parameters and local variables, and executing the function's code.

Completion of Execution

Global execution continues until all statements in the script have been executed or until an error occurs.

Once the script finishes executing, the global execution context is destroyed, and the script's execution completes.

```
console.log("Start of script");

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

var message = "Welcome to JavaScript";

greet("Alice");

console.log(message);
```

lexical scoping:-

Lexical scoping is a fundamental concept in JavaScript that determines the visibility and accessibility of variables and functions based on their location within the code.

diff bt asyncawait and promises:-

Async/await and promises are both mechanisms for handling asynchronous operations in JavaScript, but they have different syntax and behavioral characteristics.

Promises

Promises are objects representing the eventual completion or failure of an asynchronous operation.

They have a `then()` method that allows you to handle success and a `catch()` method for handling errors.

```
function fetchData() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve("Data fetched successfully");
    }, 2000);
  });
}
fetchData()
  .then(data => {
    console.log(data);
  })
  .catch(error => {
    console.error(error);
  });
```

Async/await:

Async/await is a syntactic sugar built on top of promises to make asynchronous code more readable and easier to write.

The `async` keyword is used to define a function that returns a promise, and the `await` keyword is used to pause execution until a promise is settled (resolved or rejected).

```
async function fetchData() {
   return new Promise((resolve, reject) => {
      setTimeout(() => {
         resolve("Data fetched successfully");
      }, 2000);
   });
}

async function fetchDataAndLog() {
   try {
      const data = await fetchData();
      console.log(data);
   } catch (error) {
      console.error(error);
   }
}
fetchDataAndLog();
```

Differences

Syntax:

- Promises use `then()` and `catch()` methods for chaining asynchronous operations and handling errors.

- Async/await uses `async` and `await` keywords to write asynchronous code in a synchronous-like manner, making it more readable and concise.

Error Handling

- With promises, error handling is typically done using the `catch()` method or by chaining a second `then()` with an error callback.
- With async/await, error handling is done using `try...catch` blocks, making error handling more similar to synchronous code.

3 stages of event propagations:-

Event propagation in JavaScript occurs in three stages: **capturing phase, target phase, and bubbling phase.**

1. Capturing Phase:

During the capturing phase, the event starts from the top of the DOM hierarchy (the root of the document) and propagates downward to the target element.

In this phase, the event moves through each ancestor element of the target element, from the outermost ancestor down to the target element itself

2. Target Phase:

In the target phase, the event reaches the target element for which the event was originally triggered.

At this stage, event listeners attached directly to the target element are invoked to handle the event.

3. Bubbling Phase:

After the target phase, the event enters the bubbling phase, where it propagates back up through the DOM hierarchy, starting from the target element.

In this phase, the event moves through each ancestor element of the target element, from the target element up to the outermost ancestor (the root of the document).

```
<div id="outer">
  <div id="middle">
    <div id="inner">Click me!</div>
  </div>
```

```
</div>
const outer = document.getElementById("outer");
const middle = document.getElementById("middle");
const inner = document.getElementById("inner");


outer.addEventListener("click", () => console.log("Outer"));
middle.addEventListener("click", () => console.log("Middle"));
inner.addEventListener("click", () => console.log("Inner"));
```

If you click on the innermost element ("inner"), the sequence of logged messages will be: "Inner", "Middle", "Outer".

This sequence reflects the bubbling phase, where the event starts from the target element ("inner") and propagates up through its ancestor elements.

debouncing and throtling in js:-

Debouncing and throttling are two techniques used to improve performance and optimize event handling in JavaScript, especially for tasks that involve frequent or repetitive triggering of events such as scrolling, resizing, or typing.

Debouncing:

Debouncing is a technique used to ensure that a function is not executed until after a certain amount of time has passed since the last time it was invoked.

- It postpones the execution of a function until the input event (such as scrolling or typing) has stopped occurring for a specified interval.

Debouncing is useful for scenarios where you want to wait for a pause in events before performing an action, such as auto-saving data while typing in a text field or performing a search after the user has finished typing.

```
Const getData=()=>{
console.log(“fetching data”);
}
```

```
function myDeebounce(call,d){
let timer;
return function(...args){
if(timer)cleartiemout(timer)
setTiemout(()=>{
call();
,d);
}}
const betterFunction(getData,1000);
```

Throttling:

Throttling is a technique used to limit the rate at which a function is invoked.

- It ensures that the function is not called more than once within a specified interval, regardless of how many times it is triggered.

- Throttling is useful for scenarios where you want to limit the frequency of function calls to prevent performance issues or to control the rate of user interactions, such as scrolling or resizing events.

```
Const mythrottle=(fn,d)->{
return function(...args){
setTimeout(()=>{
fn()
document.getElementByid(“myid”).disabled=true;
,d);
}}))
const newFun=myThrottle(()=>{
console.log(“user clicked”)
},5000)
```

swap values of two varibles whout using a third variable:-

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

console.log("Before swapping:");
console.log("a:", a); // Output: 5
console.log("b:", b); // Output: 10

// Swap values using destructuring assignment
[b, a] = [a, b];

console.log("After swapping:");
console.log("a:", a); // Output: 10
console.log("b:", b); // Output: 5
```

single threaded or multithreaded:-

JavaScript is primarily single-threaded, meaning it has only one call stack and one thread of execution. This means that at any given time, JavaScript code is executed sequentially, one statement at a time, and cannot perform multiple tasks concurrently in parallel.

JavaScript is primarily single-threaded, meaning it executes code sequentially on a single main thread. However, it supports concurrency through asynchronous programming techniques, allowing it to handle multiple tasks efficiently without blocking the main thread.

settimeout and below there is promise, which will run first:-

```
setTimeout(() => {
    console.log("setTimeout callback");
}, 0);

const promise = new Promise((resolve, reject) => {
    console.log("Promise executor");
    resolve("Promise resolved");
```

```
});


promise.then((result) => {
  console.log(result);
});
```

In this code, `setTimeout` is scheduled to execute its callback function after a delay of 0 milliseconds.

- The Promise is immediately constructed, and its executor function (the function passed to the `Promise` constructor) is executed synchronously.
- After the Promise executor function is executed, the Promise's `then` method is chained to handle the fulfillment of the Promise.
- Since `setTimeout` schedules its callback to be executed asynchronously after a minimum delay, it will run after the current synchronous code block has finished executing.
- Therefore, in this specific scenario, the Promise executor function will run first, followed by the `setTimeout` callback.

query selector:-

querySelector is a method in JavaScript used to select and retrieve elements from the DOM (Document Object Model) using CSS selectors.

```
<!DOCTYPE html>
<html>
<head>
  <title>querySelector Example</title>
</head>
<body>
  <div id="container">
    <h1 class="title">Hello, World!</h1>
    <p>This is a paragraph.</p>
    <ul>
      <li>Item 1</li>
```

```
        <li>Item 2</li>
    </ul>
</div>

<script>
    // Select elements using querySelector
    const titleElement = document.querySelector('.title');
    const paragraphElement = document.querySelector('p');
    const listItemElement = document.querySelector('ul li');

    // Manipulate selected elements
    titleElement.textContent = 'Updated Title';
    paragraphElement.style.color = 'blue';
    listItemElement.textContent = 'Updated Item';
</script>
</body>
</html>
```

strict mode:-

Strict mode is a feature in JavaScript that allows you to place your code into a stricter operating context. When you enable strict mode, certain actions that are considered as errors in JavaScript are treated as actual errors, helping you write more robust and secure code.

`'use strict'`

```
'use strict';

function foo() {
    x = 10; // Throws a ReferenceError in strict mode
    console.log(this); // Outputs 'undefined' in strict mode
}
```

```
foo();
```

function and variable with same name:-

In JavaScript, it is possible to have a function and a variable with the same name. However, this is not generally recommended, as it can lead to confusion and errors.

When a function and a variable have the same name, the function will take precedence. This means that if you try to access the variable, you will instead get the function.

```
function foo() {
  return "bar";
}

var foo = "baz";

console.log(foo); // Error: foo is not a function
```

In this example, the variable foo is declared after the function foo. This means that the function foo will be hoisted to the top of the scope, and the variable foo will be hidden.

To avoid this error, you should always use different names for functions and variables. If you need to use the same name for both, you can use the window object to access the global variable.

```
function foo() {
  return "bar";
}
var foo = "baz";
console.log(window.foo); // "baz"
```

Settimeout:-

```
function greet() {
```

```
  console.log('Hello, world!');
}
// Schedule the greet function to be called after 2 seconds
setTimeout(greet, 2000);
// Schedule an anonymous function to be executed after 3 seconds
setTimeout(() => {
  console.log('Delayed message');
}, 3000);
```

setinterval:-

```
function logTime() {
  console.log(new Date().toLocaleTimeString());
}

// Execute logTime every 2 seconds
const intervalId = setInterval(logTime, 2000);

// Stop the interval after 10 seconds
setTimeout(() => {
  clearInterval(intervalId);
  console.log('Interval stopped after 10 seconds');
}, 10000);
```

setTimeout executes a function once after a delay.

setInterval executes a function repeatedly at specified intervals until cleared.

Use setTimeout when you want to execute a function once after a delay, such as showing a message or performing an action after a certain time has passed.

Use setInterval when you want to execute a function repeatedly at fixed intervals, such as polling for data updates, running animations, or updating UI elements periodically.

Both setTimeout and setInterval return a timer ID, which can be used to cancel the scheduled execution using clearTimeout and clearInterval respectively.

It's important to clear timers when they are no longer needed to prevent memory leaks and unnecessary resource consumption

```
// Example using setTimeout with clearTimeout

// Define a function to be executed after 3 seconds
function greet() {
  console.log("Hello, world!");
}

// Schedule the execution of the greet function after 3 seconds
const timeoutId = setTimeout(greet, 3000);

// Cancel the scheduled execution of greet function after 2 seconds
setTimeout(() => {
  clearTimeout(timeoutId);
  console.log("Execution of greet function canceled");
}, 2000);
// Example using setInterval with clearInterval

let count = 0;

// Define a function to be executed every second
function incrementAndLog() {
  count++;
  console.log("Count:", count);
}
```

```
// Schedule the execution of incrementAndLog function every second
const intervalId = setInterval(incrementAndLog, 1000);

// Stop the execution after 5 seconds
setTimeout(() => {
  clearInterval(intervalId);
  console.log("Execution stopped after 5 seconds");
}, 5000);
```

structuredclone:-

The "structured clone" algorithm is a method used in JavaScript to create a deep copy of objects

```
// Create an object with nested properties
const originalObject = {
  name: 'John',
  age: 30,
  address: {
    city: 'New York',
    country: 'USA'
  }
};
// Use structured clone to create a deep copy
const clonedObject = structuredClone(originalObject);
console.log(clonedObject);
// Output: { name: 'John', age: 30, address: { city: 'New York', country: 'USA' } }
```

```
// Original array
const originalArray = [1, 2, 3, [4, 5]];

// Using structured clone to copy the array
const clonedArray = structuredClone(originalArray);

console.log(clonedArray); // Output: [1, 2, 3, [4, 5]]
```