JavaScript

JavaScript is used as front-end as well as the back-end

Front-end: React JS

Back-end: Node JS and Express JS

The Netscape hired the Developer Brenden Eich and said to develop a scripting language for their browser. So, Brenden Eich developed a Scripting language called as Mocha in 1995 with 10days, later it is named as Live Script but as a marketing strategy the company named it as JavaScript. Later, the Netscape company handled JavaScript to ECMA (European Computer Manufacturer Association) company then it named as ECMA Script. The ECMA company released the new version of JavaScript every year in the month of June. But now the trademark of JS is owned by ORACLE.

- It is a scripting Language designed for web pages.
- It is a client-side Scripting language.
- An Interpreted, event based language.
- High level Language.
- Weekly typed and case-sensitive
- Dynamically typed.
- It is embedded directly into HTML pages.

JavaScript is one of the sub applications of the Browser.
Brenden Eich invented JS because to avoid to get the data from the server side.
Since, JS is client side scripting language it will Read, Analyze and Execute the code in the Browser itself.
JavaScript is also a Programming Language.
Weekly typed specifies need not declare the datatype.
Dynamically typed because we can reuse the code.
It is an High Level Language because it is the human readable format.

JavaScript is an Interpreted Language

Q. What id Interpreted language?

Interpreted language is the codes are read line-by-line and executed. If at all it finds an error, it will not move to the next line of the code, it rather displays runtime error.

<u>Characteristics of JavaScript</u>

- It is an Interpreted language.
- It is a Synchronous language.
- It is purely Object Based Programming Language.

Everly Browser has a Java Script Engine which is responsible to execute JavaScript code.

Tokens : Tokens are the smallest unit of any programming language.

In tokens we have:

- Keywords
- Identifiers
- Literals

Keywords	Identifiers	Literals
• Pre-defined /reserved words. • Should be lowercase.	• Can't starts with numbers (but it can have number). • Except dollar($) and underscore(_) no other special character is allowed. • Keywords can't be used as identifiers	The Data which is used in JS programs. • Number • Boolean • Null • Undefined • Object • Bigint • Symbol.

Example: var, let, const, if, while,…	Example: name, age, skills,…	Example: name="Jhon";
		age=25;

Output methods of JavaScript

document.write(): Prints the output in the browser window (without space)
document.writeln(): Prints the output in the browser window (with space)
console.log(): Prints the output in the developers window (also called developer output)
prompt(): It is a pop-up message, takes the input from the user.
alert(): It is a pop-up message, gives the alert message to the user.
confirm(): It is a pop-up message, gives the confirm messages.

Containers of JavaScript

var	let	const
var is a global scope.	let is a script scope.	const is a script scope.
We can declare multiple variable with same name(The most recently created variable will be used)	We cannot declare two variables with the same name within a block.	We cannot declare two variables with the same name within a block.
The value of the variable can be modified.	The value of the variable can be modified.	The value of the variable cannot be modified.

var a=10;
a=20;
console.log(a);
	let a=10;
	a=20;
	console.log(a);
		const a=10;
		a=20;

//20	//20	**console.log(a);** //we cannot modify the data with new value. //type error
We can declare var without initialization. **Ex: var x;**	We can declare let without initialization. **Ex: let y;**	We cannot declare const without initialization. **Ex: const z;** //syntax error

The variable declared using var Belongs to global scope(window), we can access them using window object.
 	
Ex: var a=10;
console.log(window.a);
//10
	The variable declare using let does not belongs to global scope we cannot use them with the help of window. (script/block)
	Ex: let b=20;
	Console.log(window.b);
	//undefined
		The variable declare using const does not belongs to global scope we cannot use them with the help of window. (script/block)
		Ex: const c=30;
		Console.log(window.c);
		//undefined

Window object

When a JS file is given to browser by default a global window object is created and the reference is stored in window variable.
The global object consist of pre-defined members (functions and variables) which belong to browser window.
Any member we declare var in JS file is added inside global window object by JS engine. So we can use member with the help of window.
Any members(functions and variable) created in global scope will be added into the window object implicitly by JS Engine

```
var a=10; //var a is added into the global window object.
console.log(window.a); //10
```

It present in global execution context.
Window is a variable, store the address of object and its point global context which present in the heap memory.
We can access window object using window variable.
Anything inside window we can access using dot operator. Ex: window.a
Whenever we create variable using const does not belong to global object and are not added to window object.
The variable declared using var belongs to global object(window), We can access them using window Object.
The variable declared using let and const does not belong to global object and are not added to window object (cannot use them with the help of window).

Hoisting: Utilizing the variable before declaration and initialization is called as Hoisting.

Hoisting can be achieved by var, because var is a global scope or global variable.
Hoisting cannot be achieved by let and const, because let and const are script scope.
Whenever we hoist var the result is undefined.

Whenever we try to hoist let and const the result is Uncaught ReferrenceError.

<u>Temporal Dead Zone (TDZ)</u>: In the process of hoisting the time taken between Utilization of the variable before declaration and initialization.
TDZ is achieved only in let and const.
Because, whenever we try to hoist let and const the result is Uncaught ReferrenceError.
TDZ cannot be achieved in var.
Because, whenever we hoist var the result is undefined.

DATATYPES IN JAVASCRIPT:

1.PRIMITIVE DATATYPE : Which is immutable

number, string, boolean, null, undefined ,bigInt ,Symbol.

2.NON-PRIMITIVE DATATYPE: which is mutable.

functions, arrays,

objects =>date object ,math object.

<u>Operators</u>:

Operators are the symbol which performs specific operations on the operators.

Types of operators

Arithmetic Operator
Assignment Operator
Relational / Compressional Operator
Logical Operator
Ternary / Conditional Operator.

Arithmetic Operators

+ - * / %

Assignment Operators

= += -= *= /= %= **= //=

Relational / Compressional Operator

< > <= >= == != === !==

Logical Operator

&& || !

Ternary / Conditional Operator

(condition) ? expression1 : expression2

Functions

Functions are the block of statements which will get executed whenever it is called or invoked.

Types of Functions

- Anonymous Function
- Named Function
- Function with expression
- Nested Function
- Arrow Function
- Higher Order Function
- Callback Function
- Immediate Invoke Function
- Generator Function

Anonymous Function: The function which does not have function name is called as Anonymous Function.

```
Syntax:         function    ( ) {
                        Statements;
                }( );
```

Named Function: The function which has function name is called as Named Function.

```
Syntax:         function    function_name ( ) {
                        Statements;
                }
                function_name( );
```

Function with Expression: The function which is assigned as a value to a variable then the expression is called as Function with Expression.

```
Syntax:         var  variable_name   =   function    function_name ( ) {
                                          Statements;
                                          }
                                          variable_name( );
```

Arrow Function: A function having a fat arrow is called as Arrow function. To reduce the function syntax we will go for Arrow function.
this keyword doesn’t work in Arrow Function.

```
Syntax:         variable_name = ( ) = > {
                        Statements;
                }
variable_name( );
```

Nested Function: A function inside another function is called as Nested Function.
Note: One inside another function. Not one inside many function.

```
Syntax:         function    function_name1 ( ) {
                        Statements;
                              function    function_name2( ) {
                                 Statements;
```

```
                                    function    function_name3( ) {
                                        Statements;
                                    }
                                return function_name3( );
                                }
                    return function_name2( );
                    }
                    function_name1( )( )( );
```

In Nested Function we can achieve Lexical Scope / Scope chain

Lexical Scope: The ability of the JavaScript engine to search of the variable in the global scope when it is not available in the local scope. It is also known as Scope chain.

Closure: Closure holds the data of the parent function which is required by the child function
Not all the data of the parent function.
Closure object address will be stored in the child function for further/future utility.

Immediate Invoke Function Execution: IIFE is a function executes a function which is enclosed within the parenthesis ( ).

```
Syntax:         (function function_name( ){
                        Statements;
                })
                ( );
```

IIFE should be called immediately right after the declaration.
IIFE cannot be reused, it is only one time usable function.
IIFE can have function name but it cannot be called with function name.
Anonymous function, Named function, Nested function, Arrow function can be IIFE.

Higher order function: The function which takes another function as an argument is called as Higher order function.
Callback function: The function which is passed as an argument to the function / Higher order function is called as Higher order function.

```
Syntax:         function function_name(para1,para2)
                {
                        Para2( );
                        Return;
                } fuction_name(argu1,function ( ) {  } );
```

NOTE:
Higher order function accepts callback function as an argument.
Higher order function calls the call back function.

Array Methods

push( ), pop( ), unshift( ), shift( ), toString( ), slice( ), splice( ), concat( ),fill( ), sort( ), includes( ), indexOf( ), lastindexOf( ), some( ), every( ).

Array Methods

map()
reduce()
filter()

map()- The map() method is one of the array method.
A map() is also a Higher order function which accepts the callback functions, where that callback function accepts three arguments.

element
index
array

Syntax: array.map((element,index,array)=> { })
Example:

```
arr=[15,25,35,45,55,65]
arr.map((x,y,z)=>{
console.log(x,y,z)
})
```

Output:

```
15 0 (6) [15, 25, 35, 45, 55, 65]
25 1 (6) [15, 25, 35, 45, 55, 65]
35 2 (6) [15, 25, 35, 45, 55, 65]
45 3 (6) [15, 25, 35, 45, 55, 65]
55 4 (6) [15, 25, 35, 45, 55, 65]
65 5 (6) [15, 25, 35, 45, 55, 65]
```

In the callback, only the array element is required. Usually some action is performed on the value and then a new value is returned.
The map() method acts similar to the for loop which fetch each elements of an array and perform the specified task.
The map() returns the result in the form of array.

Example1:

```
var add=arr.map((x)=>{return x+25});
console.log(add)
```

Output: (6) [40, 50, 60, 70, 80, 90]

Example2:

```
var add=arr.map((x)=>{return x>25});
console.log(add)
```

Output: [false, false, true, true, true, true]

filter()- The filter() method is one of the array method.
The filter() is also a Higher order function which accepts the callback functions.

The filter() method call the predicate function one time for each element in the array.
The filter() methos helps to filter individual element of an array based on the condition, we pass in filter method as a callback function and it returns the value in the form of array.

Example:

```
arr=[15,25,35,45,55,65];
var a=arr.filter((y)=>{return y>25});
console.log(a)
```

```
//[35, 45, 55, 65]
```

Note: (Example for thisarg in filter)

```
  const musics = [
   {
     name: 'stinks like unwashed adolescent',
     tags: ['teen', 'energetic', 'excelent']
   },
   {
     name: 'After beethovens 4th',
     tags: ['older', 'moving', 'excelent']
   },
   {
     name: 'wap',
     tags: ['hip-hop', 'pounding', 'excelent']
   }
 ]

 const filterTags = function(element, index, arr) {
   console.log(this.tag)
   return element.tags.includes(this.tag)
 }

 console.log(musics.filter(filterTags, {tag:'teen'}))
```

reduce()- The reduce() method is one of the array method.
The reduce() is also a Higher order function which accepts the callback functions.
That call back functions accepts two parameters, that is: (accumulator, value)
accumulator fetch the first element of an array and stores the final result.
value fetch the second element of an array and the perform the specified operation.

Example:

```
var arr=[15,25,35,45,55,65];
var result=arr.reduce((accu, value)=>{
        return accu+value;
})
console.log(result);
```

Output: 240

Example:

```
var arr=[12, 15, 10, 30];
var result=arr.reduce((accu, value)=>{
return ans=accu+value;
},10)
console.log(ans)
```

Output: 77

String Methods

length, repeat, toUppercase( ), toLowercase( ), indexOf( ), includes( ), replace( ), replaceAll( ), substring( ), slice( ), substr( ), concat( ), trim( ), trimStart( ), trimEnd( ), padStart( ), padEnd( ), charAt( ), charCodeAt( ), endswith( ), startswith( ),reverse().

Object: Object is associated with properties separated with commas and every property is in the form of key and value pair.

Syntax:
```
let object_name={
        Key1: value1,
        Key2: value2,
        .
        .
}
```

Every property of an is Object is displayed in the form of array but cannot be accessed through index values.
The JavaScript Objects keys accepts values of all datatype.
Object is mutable mean Object address cannot be changed.
We can fetch the key and value.

Value can be fetched using one these notations

.	dot notation
[ ]	Square notation

We can reinitialize the value but key cannot be reinitialized.
We can remove the value but key cannot be removed.
Nested Object can be created.

The function can also be given as a value to a variable.

JavaScript Objects can be created using these below mentioned ways:

Object Literals

Syntax:
```
let object_name = {
                Key1: value1,
                Key2: value2,
                .
                .
        }
```

new keyword

Syntax: let object_name = new Object( );

Constructor function

Advantage: Using Constructor function multiple objects can be created at a time.

Syntax:

```
function  function_name(key1, key2, …){
        this.key1=key1;
        this.key2=key2;
        .
        .
}
let object_name1= new  function_name(value1, value2, …);
let object_name2= new  function_name(value1, value2, …);
```

DATE OBJECT:

In JavaScript, the Date object is used for working with dates and times. It allows you to create, manipulate, and format dates and times. The Date object is built into the JavaScript language and provides several methods and properties for working with dates and times.

const date = new Date();

The Date object in JavaScript comes with a variety of built-in methods to perform various operations on dates and times. Here are some of the commonly used built-in methods for the Date object:

getFullYear(): Returns the year (4 digits) of the date.

getMonth(): Returns the month (0-11) of the date.

getDate(): Returns the day of the month (1-31) of the date.

getDay(): Returns the day of the week (0-6) of the date (0 is Sunday, 6 is Saturday).

getHours(): Returns the hours (0-23) of the time.

getMinutes(): Returns the minutes (0-59) of the time.

getSeconds(): Returns the seconds (0-59) of the time.

getMilliseconds(): Returns the milliseconds (0-999) of the time.

getTime(): Returns the number of milliseconds since January 1, 1970 (Unix timestamp).

MATH OBJECT:

The Math object in JavaScript provides a collection of mathematical constants and functions for performing various mathematical operations. It is not a constructor like the Date object and does not have properties that you can modify. Instead, it's a static object that contains only methods and constants. Here are some of the commonly used methods and constants provided by the Math object in JavaScript:

Math.pow(x, y): Returns x raised to the power of y.

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

Math.cbrt(x): Returns the cube root of x.

Math.round(x): Returns the nearest integer to x.

Math.floor(x): Returns the largest integer less than or equal to x.

Math.ceil(x): Returns the smallest integer greater than or equal to x.

Math.trunc(x): Returns the integer part of x by removing the fractional part.

Math.random(): Returns a pseudo-random number between 0 (inclusive) and 1 (exclusive).

Math.random() * (max - min) + min: Generates a random number between min (inclusive) and max (exclusive).

Math.min(x, y, ...args): Returns the smallest of the provided values.

Math.max(x, y, ...args): Returns the largest of the provided values.

Synchronous Code:

In synchronous code execution, tasks are performed one at a time, in a sequential and blocking manner. Each task must complete before the next one can begin. If a task takes a long time to execute, it can block the entire program, making it unresponsive.

Asynchronous Code:

In asynchronous code execution, tasks are initiated and then allowed to run independently without waiting for their completion. This means that multiple tasks can be initiated and run concurrently. Asynchronous code allows a program to remain responsive even when performing tasks that take time to complete.

In JavaScript, setTimeout and setInterval are two functions that allow you to execute code at specified intervals. They are often used for creating timers and scheduling functions to run asynchronously.

setTimeout Function:

setTimeout is used to execute a specified function or code snippet after a specified delay (in milliseconds). It schedules a single execution of the specified function.

Syntax: setTimeout(callback, delay);

callback: A function to be executed after the delay.

delay: The time (in milliseconds) to wait before executing the callback.

EXAMPLE:

```
setTimeout(function () {
  console.log("This code will run after 1000 milliseconds (1 second).");
}, 1000);
```

setInterval Function:

setInterval is used to repeatedly execute a specified function or code snippet at a defined time interval. It keeps executing the function at regular intervals until it's canceled.

Syntax: setInterval(callback, delay);

callback: A function to be executed at each interval.

delay: The time (in milliseconds) between each execution of the callback.

Example:

```
const intervalId = setInterval(function () {
  console.log("This code will run every 2000 milliseconds (2 seconds).");
}, 2000);
```

setInterval returns a unique interval identifier that can be used to cancel the repeated function execution using clearInterval.

EXAMPLE:

```
const intervalId = setInterval(function () {
  console.log("This code will run indefinitely.");
}, 1000);
// Cancel the repeated function execution after 5 seconds
setTimeout(function () {
  clearInterval(intervalId);
}, 5000);
```

Window Object:

In JavaScript, the window object is a global object that represents the browser window or the global environment in a web browser. It serves as the top-level object in the browser's Document Object Model (DOM) hierarchy and provides access to various properties and methods that allow you to interact with the browser and control the web page.

this keyword in js:

In JavaScript, the this keyword is a special keyword that refers to the current context or object within which it is used. The behaviour of this depends on how and where it is used, and it can have different values in different contexts. Understanding how this behaves is crucial for working with JavaScript objects and functions.

Here are some common scenarios in which the this keyword behaves differently:

Global Context:

In the global scope (outside of any function or object), this refers to the global object, which in a browser environment is the window object.

Function Context:

Inside a regular function, the value of this depends on how the function is called.

In strict mode ("use strict";), if the function is called without any context (i.e., not as a method of an object or not using call() or apply()), this is undefined. But without strict mode it points towards the window object. But in arrow function it points towards the window object.

```
function foo() {
  console.log(this); // In non-strict mode: window, In strict mode: undefined
}
foo();
```

Method Context:

Inside a method of an object, this refers to the object itself.

Example:

```
const myObject = {
  prop: 'Hello',
  method: function () {
    console.log(this.prop); // 'Hello'
  },
};
myObject.method();
```

Constructor Context:

Inside a constructor function (a function used with the new keyword to create objects), this refers to the newly created object.

Example:

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

```
const person1 = new Person('Alice');

console.log(person1.name); // 'Alice'
```

Explicit Binding:

You can explicitly set the value of this using methods like call(), apply(), or bind().

EX:

```
function greet() {
  console.log(`Hello, ${this.name}`);
}

const person = { name: 'Alice' };

greet.call(person); // Explicitly bind `this` to the `person` object.
```

Rest Parameter (...) :

The rest parameter allows you to collect a variable number of function arguments into an array. It provides a way to work with an indefinite number of arguments as an array within a function.

Here's how the rest parameter works:

```
function sum(...numbers) {
  let result = 0;
  for (let number of numbers) {
    result += number;
  }
  return result;
}

console.log(sum(1, 2, 3, 4, 5)); // 15
```

In this example, the sum function takes any number of arguments, and the rest parameter ...numbers collects them into an array called numbers. You can then loop through this array to perform operations on the arguments.

Spread Operator (...):

The spread operator allows you to spread the elements of an iterable (e.g., an array) into another array, function argument, or object. It is used for unpacking values from an iterable, making it easier to work with arrays, objects, and other iterable data structures.

Here are some common use cases for the spread operator:

1. Spreading Arrays:

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

2. Combining Arrays:

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

3. Copying Arrays:

```
const originalArray = [1, 2, 3];
const copyArray = [...originalArray];
console.log(copyArray); // [1, 2, 3]
console.log(originalArray === copyArray); // false (not the same reference)
```

4. Spreading into Function Arguments:

```
function multiply(x, y, z) {
  return x * y * z;
}
const numbers = [2, 3, 4];
const result = multiply(...numbers);
console.log(result); // 24
```

5. Spreading into Objects (Object Cloning and Merging):

```
const person = { name: 'Alice', age: 30 };
const updatedPerson = { ...person, age: 31 };
console.log(updatedPerson); // { name: 'Alice', age: 31
```

In this example, the spread operator is used to create a new object (updatedPerson) by copying all properties from the person object and overriding the age property.

Array Destructuring:

Array destructuring enables you to unpack values from an array and assign them to variables. It is particularly useful when working with functions that return arrays or when you want to access specific elements of an array easily.

Example:

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

// Destructuring assignment

const [a, b, c] = myArray;

console.log(a); // 1

console.log(b); // 2

console.log(c); // 3
```

You can also use array destructuring with rest elements to collect remaining items into a new array:

```
const numbers = [1, 2, 3, 4, 5];

const [first, second, ...rest] = numbers;

console.log(first); // 1

console.log(second); // 2

console.log(rest); // [3, 4, 5]
```

Object Destructuring:

Object destructuring allows you to extract values from objects and assign them to variables with the same property names. It's especially helpful when you want to work with objects in a more readable and concise way.

Here's how object destructuring works:

```
const myObject = { name: 'Alice', age: 30 };

// Destructuring assignment

const { name, age } = myObject;

console.log(name); // 'Alice'
```

console.log(age); // 30

You can also assign values to new variable names using object destructuring:

const person = { name: 'Bob', age: 25 };

// Destructuring assignment with new variable names

const { name: fullName, age: years } = person;

console.log(fullName); // 'Bob'

console.log(years); // 25

Object Methods in JS:

1. Object.freeze(obj):

The Object.freeze() method is used to freeze an object, making it immutable. Once an object is frozen, you cannot add, delete, or modify its properties or values.

Attempting to change a property's value or add/delete properties will have no effect, and JavaScript will not throw an error.

This method recursively freezes all nested objects and their properties.

2. Object.isFrozen(obj):

The Object.isFrozen() method checks if an object is frozen (immutable). It returns true if the object is frozen, otherwise false.

3. Object.seal(obj):

The Object.seal() method seals an object, making it non-extensible. While you can still modify the values of existing properties, you cannot add or delete properties.

Unlike Object.freeze(), sealing an object allows you to change the values of its existing properties, but you cannot add or remove properties.

4. Object.isSealed(obj):

The Object.isSealed() method checks if an object is sealed (non-extensible). It returns true if the object is sealed, otherwise false.

5.Object.assign():

The Object.assign() method is a built-in method in JavaScript that is used for copying the values of all enumerable properties from one or more source objects into a target object. It is often used to create a new object by merging properties from multiple source objects into a single destination (target) object. The Object.assign() method does a shallow copy of object properties, meaning that nested objects are not deeply copied; rather, references to nested objects are copied.

Here's the basic syntax of Object.assign():

```
Object.assign(target, source1, source2, ...);
```

target: The target object that will receive the copied properties. This object will be modified in place and also returned as the result.

source1, source2, ...: One or more source objects whose properties will be copied into the target object.

Here's an example of how to use Object.assign():

Example:

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

const source1 = { b: 3, c: 4 };

const source2 = { d: 5 };

// Merging properties from source1 and source2 into target

const mergedObject = Object.assign(target, source1, source2);

console.log(target); // { a: 1, b: 3, c: 4, d: 5 }

console.log(mergedObject === target); // true (the same object reference is returned).
```

In this example:

target is the object that receives the merged properties.

source1 and source2 are the source objects whose properties are copied into target.

The properties from source1 and source2 are merged into target, and the resulting modified target object is returned.

It's important to note that Object.assign() modifies the target object in place and also returns the modified target object. If there are properties with the same names in both the target and source objects, the properties in the source objects overwrite the corresponding properties in the target object. Additionally, if any of the source objects are null or undefined, they are simply ignored.

6. Object.keys(obj):

The Object.keys() method returns an array containing the keys of the given object.

7. Object.values(obj):

The Object.values() method returns an array containing the values of the enumerable properties of the given object.

8. Object.entries(obj):

The Object.entries() method returns an array of key-value pairs (entries) as arrays, where each sub-array contains a property name (key) and its associated value. It provides an easy way to iterate through both keys and values of an object.

structuredClone():

The primary purpose of structuredClone() is to create a deep clone of an object. It can clone not only simple objects but also complex data structures, such as nested objects, arrays, functions (in some cases), dates, and more.

EXAMPLE:

```
const originalObject = {
  name: 'Alice',
  age: 30,
  friends: ['Bob', 'Charlie'],
};
const clonedObject = structuredClone(originalObject);
console.log(clonedObject);
```

JSON:

JSON (JavaScript Object Notation) is a lightweight data interchange format used for storing and exchanging data between a server and a client, or between different parts of a JavaScript application. It is a text-based format that is easy for both humans and machines to read and write.

JSON Objects:

JSON objects are collections of key-value pairs. Keys are strings, and values can be strings, numbers, objects, arrays, booleans, null, or nested JSON objects.

```
let obj={
  "name": "Alice",
  "age": 30,
  "city": "New York"
}
```

1. JSON.stringify():

The JSON.stringify() method is used to convert a JavaScript object or value into a JSON-formatted string.

It takes an object or value as its argument and returns a string representing the JSON version of that object or value.

Example:

```
const person = {
  name: "Alice",
  age: 30,
  city: "New York",
};
const jsonString = JSON.stringify(person);
console.log(jsonString);
```

2. JSON.parse():

The JSON.parse() method is used to parse a JSON-formatted string and convert it into a JavaScript object.

It takes a JSON-formatted string as its argument and returns a JavaScript object.

Example:

```
const jsonString = '{"name":"Alice","age":30,"city":"New York"}';
const person = JSON.parse(jsonString);
console.log(person.name); // "Alice"
```

Shallow Copy:

A shallow copy of an object creates a new object with a new reference, but it does not create new copies of nested objects or arrays within the original object. Instead, it copies references to those nested objects or arrays. As a result, changes made to nested objects in the copied object will affect the original object, and vice versa.

Here's how to achieve a shallow copy in JavaScript:

1.Using the spread operator (...):

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

```
// Modifying a property in the shallow copy

shallowCopy.b.c = 3;

console.log(originalObject.b.c); // 3 (originalObject is affected)
```

2.Using Object.assign():

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

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

// Modifying a property in the shallow copy

shallowCopy.b.c = 3;

console.log(originalObject.b.c); // 3 (originalObject is affected)
```

Deep Copy:

A deep copy of an object creates a completely independent copy of the original object, including all nested objects and arrays. Changes made to the copied object or its nested objects will not affect the original object, and vice versa.

Here's an example of how to achieve a deep copy using a structuredClone():

```
// Create an object to be deep copied

const originalObject = {

  a: 1,

  b: {

    c: 2,

    d: [3, 4],

  },

};

// Deep copy using structuredClone()

const deepCopy = structuredClone(originalObject);

// Modify the deep copy

deepCopy.b.c = 99;

deepCopy.b.d.push(5);

// Verify that the originalObject is not affected

console.log(originalObject); // { a: 1, b: { c: 2, d: [3, 4] } }

console.log(deepCopy);       // { a: 1, b: { c: 99, d: [3, 4, 5] } }
```

2.Using JSON.parse() and JSON.stringify():

```
// Create an object to be deep copied
const originalObject = {
  a: 1,
  b: {
    c: 2,
    d: [3, 4],
  },
};
// Deep copy using JSON.stringify() and JSON.parse()
const deepCopy = JSON.parse(JSON.stringify(originalObject));
// Modify the deep copy
deepCopy.b.c = 99;
deepCopy.b.d.push(5);
// Verify that the originalObject is not affected
console.log(originalObject); // { a: 1, b: { c: 2, d: [3, 4] } }
console.log(deepCopy);       // { a: 1, b: { c: 99, d: [3, 4, 5] } }
```

In JavaScript, the call(), apply(), and bind() methods are used to manipulate the context (the value of the this keyword) and pass arguments to functions.

1. call() Method:

The call() method is used to invoke a function with a specified this value and a list of arguments. It allows you to call a function as if it were a method of a specific object, even if the function is not originally a method of that object.

Syntax: function.call(thisArg, arg1, arg2, ...);

thisArg: The object to which the this keyword will refer when the function is executed.

arg1, arg2, ...: Arguments to be passed to the function.

```
const person = {
  firstName: 'John',
  lastName: 'Doe',
  fullName: function() {
    return this.firstName + ' ' + this.lastName;
  }
};
const greeting = person.fullName.call(person);
console.log(greeting); // 'John Doe'
```

In this example, call() is used to execute the fullName function with the person object as the context.

2. apply() Method:

The apply() method is similar to call() but accepts arguments as an array or an array-like object.

Syntax: function.apply(thisArg, [argsArray]);

thisArg: The object to which the this keyword will refer when the function is executed.

argsArray: An array or array-like object containing the arguments to be passed to the function.

Example:

```
function sum(a, b) {
  return a + b;
}
const args = [3, 4];
const result = sum.apply(null, args);
console.log(result); // 7
```

In this example, apply() is used to call the sum function with arguments from the args array.

3. bind() Method:

The bind() method creates a new function that, when called, has its this value set to a specific object, and it can also pre-fill arguments. Unlike call() and apply(), bind() does not immediately execute the function; instead, it returns a new function with the specified context and argument values.

Example:

```
const greet = function() {
  console.log(`Hello, ${this.name}`);
};
const person = {
  name: 'Alice',
};
const greetAlice = greet.bind(person);
greetAlice(); // 'Hello, Alice'
```

Key Differences:

call() and apply() immediately invoke the function, while bind() returns a new function without executing it.

Set Object:

A Set is a collection of unique values, which means that each value can appear only once in a Set. Sets are particularly useful when you need to keep track of a list of values without any duplicates.

Map Object:

A Map is a collection of key-value pairs, where each key can be associated with a value. Maps are versatile and useful when you need to store and retrieve data based on a specific key.

Key Differences:

Sets are collections of unique values, while Maps are collections of key-value pairs.

Sets are primarily used when you need to maintain a list of unique values, whereas Maps are used when you want to associate data with specific keys.

Sets only store values, whereas Maps store both keys and values, allowing you to look up values based on their corresponding keys.

Both Sets and Maps maintain the order of insertion, meaning the order in which values or key-value pairs are added is preserved.

Sets have methods like add(), has(), and delete(), while Maps have methods like set(), get(), has(), and delete() for managing their contents.

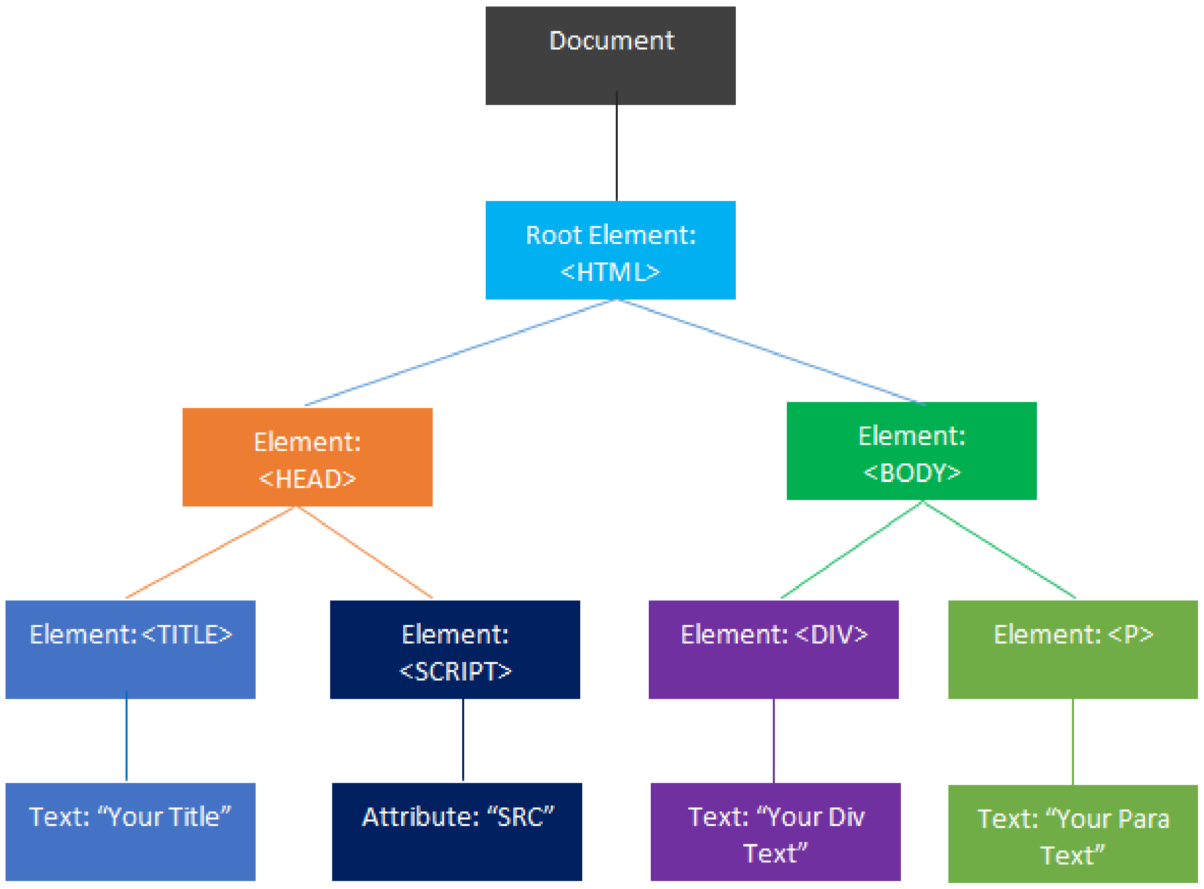
Document Object Model (DOM):

The DOM is a hierarchical representation of a web page's structure.

It provides a way to interact with HTML or XML documents using JavaScript.

When a web page is loaded, the browser creates a Document Object Model of the page.

The HTML DOM model is constructed as a tree of Objects:



A Simple DOM Tree

With the object model, JavaScript gets all the power it needs to create dynamic HTML:

- JavaScript can change all the HTML elements in the page
- JavaScript can change all the HTML attributes in the page
- JavaScript can change all the CSS styles in the page
- JavaScript can remove existing HTML elements and attributes
- JavaScript can add new HTML elements and attributes
- JavaScript can react to all existing HTML events in the page
- JavaScript can create new HTML events in the page.

Direct Access From DOM(document Object):

```
console.log(document.title);

console.log(document.URL);
console.log(document.contentType); //text/html
console.log(document.characterSet); //utf-8
console.log(document.children);
console.log(document.addEventListener); //method
console.log(document.append);
console.log(document.appendChild);
console.log(document.body);
console.log(document.styleSheets);
console.log(document.images);
console.log(document.links);
console.log(document.COMMENT_NODE);
console.log(document.scripts);
console.log(document.close);
console.log(document.open); //method
console.log(document.cloneNode); //method
console.log(document.contains); //method
console.log(document.cookie); //important for server side sending small chunk of data.
console.log(document.createAttribute);
console.log(document.getElementById);
console.log(document.getElementsByTagName);
console.log(document.getElementsByClassName);
console.log(document.querySelector);
console.log(document.querySelectorAll);
console.log(document.write);
console.log(document.doctype);
console.log(document.createElement);
console.log(document.createComment);
console.log(document.createEvent);
```

Attribute methods in JS:

getAttribute(attributeName):

This method allows you to retrieve the value of a specific attribute of an element.

setAttribute(attributeName, value):

Use this method to set the value of a specific attribute for an element.

hasAttribute(attributeName):

This method checks if an element has a specific attribute and returns a boolean value.

removeAttribute(attributeName):

Use this method to remove a specific attribute from an element.

ClassList property:

In JavaScript, the classList property of an HTML element provides an interface to manipulate the class attributes of that element. The classList property is particularly useful for adding, removing, toggling, and checking the presence of CSS classes on an element. Here are some common methods and properties of the classList object:

classList.add(className1, className2, ...):

Adds one or more CSS classes to the element.

classList.remove(className1, className2, ...):

Removes one or more CSS classes from the element.

classList.toggle(className):

Toggles a CSS class on the element. If the class is present, it removes it; if it's absent, it adds it.

classList.contains(className):

Checks if the element has a specific CSS class and returns true or false.

<u>innerText:</u>

innerText is a property that represents the text content within an HTML element, excluding any HTML tags or elements inside the element.

It returns or sets only the plain text content and does not interpret or modify any HTML.

It is a safer option when you want to work specifically with the text inside an element and avoid issues related to HTML injection or unintentional code execution.

<u>innerHTML:</u>

innerHTML is a property that represents the HTML content within an HTML element, including any HTML tags or elements inside the element.It can be used to both read and set the HTML content, making it a powerful tool for dynamically generating or modifying content.

the main difference between innerText and innerHTML is that innerText deals with the plain text content of an element and does not consider HTML tags, while innerHTML deals with the HTML content, including any tags and elements within the element.

DOM inbuilt methods:

Accessing Elements:

- document.getElementById(id): Retrieves an element with the specified id.
- document.getElementsByClassName(className): Retrieves elements with the specified class name, returning them as an HTMLCollection.
- document.getElementsByTagName(tagName): Retrieves elements with the specified tag name, returning them as an HTMLCollection.
- document.querySelector(selector): Retrieves the first element that matches the specified CSS selector.
- document.querySelectorAll(selector): Retrieves all elements that match the specified CSS selector, returning them as a NodeList.

Creating and Modifying Elements:

- document.createElement(tagName): Creates a new HTML element with the specified tag name.
- element.appendChild(newChild): Appends a new child node (element or text) to an existing element.
- element.removeChild(child): Removes a child node from an element.
- element.innerHTML: Gets or sets the HTML content within an element (can be used for creating or modifying elements).

Styling Elements:

element.style.property: Allows you to access or modify CSS properties of an element directly (e.g., element.style.color).

Dealing with NodeList vs. HTMLCollection:

When you use methods like querySelectorAll or getElementsByTagName, you get a NodeList or HTMLCollection, respectively. These are array-like objects but not actual arrays. To convert them to an array, you can use the Array.from method or the spread operator:

EX:

```
const elements = document.querySelectorAll(".myClassName");

const elementArray = Array.from(elements); // Convert NodeList to an array
```

DOM Events:

Events in the context of the Document Object Model (DOM) in JavaScript refer to specific interactions or occurrences that take place in a web page. These interactions can include user actions (e.g., clicking a button, moving the mouse).

Event handling:

Event Handling in the DOM involves writing JavaScript code to respond to these events. Event handling allows you to define how your web page or application should react when these events occur.

addEventListener():

The addEventListener method in JavaScript is used to attach event handlers (functions) to DOM elements, allowing you to respond to specific events that occur on those elements. It is a fundamental method for event handling in web development.

syntax: element.addEventListener(eventType, eventHandler, options);

1. **eventType**: A string specifying the type of event you want to listen for (e.g., "click," "keydown," "submit").
2. **eventHandler:** A function that gets executed when the specified event occurs on the element.
3. **options (optional):** An optional object that can be used to configure the event listener, such as specifying whether the event should use capturing (true) or bubbling (false) as the propagation mode.

Event Object:

The event handler function receives an event object as a parameter. This object provides information about the event, such as its type, target element, and additional properties specific to the event type.

You can access this event object and use its properties to gather information about the event.

Types of events in DOM:

Click Event:

The click event occurs when a mouse click (left button) is detected on an element.

It's often used for handling user interactions such as button clicks, links, or toggling elements.

Mouseover and Mouseout Events:

mouseover occurs when the mouse pointer enters an element.

mouseout occurs when the mouse pointer leaves an element.

These events are useful for creating hover effects or tooltips.

Keyup, Keydown, and Keypress Events:

keydown occurs when a key is pressed down.

keyup occurs when a key is released.

keypress (less commonly used) occurs when a key is pressed and released.

These events are used to capture keyboard input and can be used for tasks like form validation or creating keyboard shortcuts.

Input Event:

The input event is triggered when the value of an input field or textarea changes, typically as the user types or pastes content.

It's commonly used to provide real-time feedback or perform actions as the user interacts with an input element.

Change Event:

The change event occurs when the value of an input field, select dropdown, or textarea changes and then loses focus (e.g., clicking outside the input).

It's often used for form validation or handling changes in select menus.

Submit Event:

The submit event occurs when a form is submitted, either by clicking a submit button or pressing Enter in a text input.

It's used to handle form submissions, validate user input, and send data to a server.

These events are fundamental to building interactive web applications.

Event Propagation:

Event Propagation refers to how events travel through the Document Object Model (DOM) tree. The DOM tree is the structure which contains parent/child/sibling elements in relation to each other.

Phases in Event Propagation:

Capturing Phase:

The event starts at the root of the DOM tree and moves down towards the target element.

During this phase, event handlers attached with the capture option set to true are executed on ancestor elements before reaching the target.

Target Phase:

This phase occurs when the event reaches the target element.

The event handler attached directly to the target element is executed in response to the event.

The event object contains information about the event and can be used by the handler to perform actions based on the event.

Bubbling Phase:

After the target phase, the event continues to propagate back up the DOM tree, moving away from the target element.

During this phase, event handlers attached without specifying the capture option or with false (the default) are executed on ancestor elements.

preventDefault() Method:

The preventDefault() method is used to prevent the default action associated with an event from occurring. Each type of event has a default action associated with it. For example, clicking on a link (an anchor element) typically navigates to the linked page, which is the default action for a click event on a link.

By calling event.preventDefault() within an event handler, you can prevent this default behavior from happening. This is commonly used to implement custom behavior when, for example, you want to handle navigation within a single-page application without a full page reload.

stopPropagation() Method:

The stopPropagation() method is used to stop the propagation of an event through the DOM tree. Events typically propagate from the target element (where the event occurred) up to the root of the DOM (the window object) in a process known as event bubbling.

By calling event.stopPropagation() within an event handler, you can prevent the event from continuing to propagate up or down the DOM tree. This can be useful when you want to handle an event only at a specific level of the DOM hierarchy without triggering other event handlers further up or down the tree.

localStorage and sessionStorage:

localStorage and sessionStorage are two web storage APIs in JavaScript that allow you to store key-value pairs locally on a user's web browser. They are useful for persisting data between page loads or maintaining data within a session, but they have different lifespans and use cases:

localStorage:

Lifetime: Data stored in localStorage persists even after the browser is closed and is available across browser sessions. It is stored until explicitly removed by the user or JavaScript code.

Scope: Data stored in localStorage is accessible to all windows or tabs from the same origin (i.e., the same protocol, domain, and port).

Use Cases:

- Storing user preferences or settings that should persist across sessions.
- Caching data to reduce server requests and improve app performance.
- Storing data for offline use in web applications.

sessionStorage:

Lifetime: Data stored in sessionStorage is available only for the duration of the page session. It is cleared when the page is closed or the session ends (e.g., when the user closes the browser or tab).

Scope: Data stored in sessionStorage is limited to the current window or tab. It is not accessible from other windows or tabs even if they are from the same origin.

Use Cases:

- Storing temporary data that should be available only during the current session.
- Implementing shopping cart functionality or managing state during a single browsing session.

Common Methods for localStorage and sessionStorage:

setItem(key, value):

Adds a key-value pair to the storage. If the key already exists, it updates the associated value.

getItem(key):

Retrieves the value associated with the specified key.

removeItem(key):

Removes the key-value pair associated with the specified key from storage.

clear():

Removes all key-value pairs from storage.

Regular expressions:

Regular expressions, often referred to as regex or regexp, are powerful patterns that allow you to search for and manipulate text in JavaScript and many other programming languages. They are used to match and manipulate strings based on specific patterns or rules. In JavaScript, you can work with regular expressions using the RegExp object or by using regex literals enclosed in forward slashes (/).

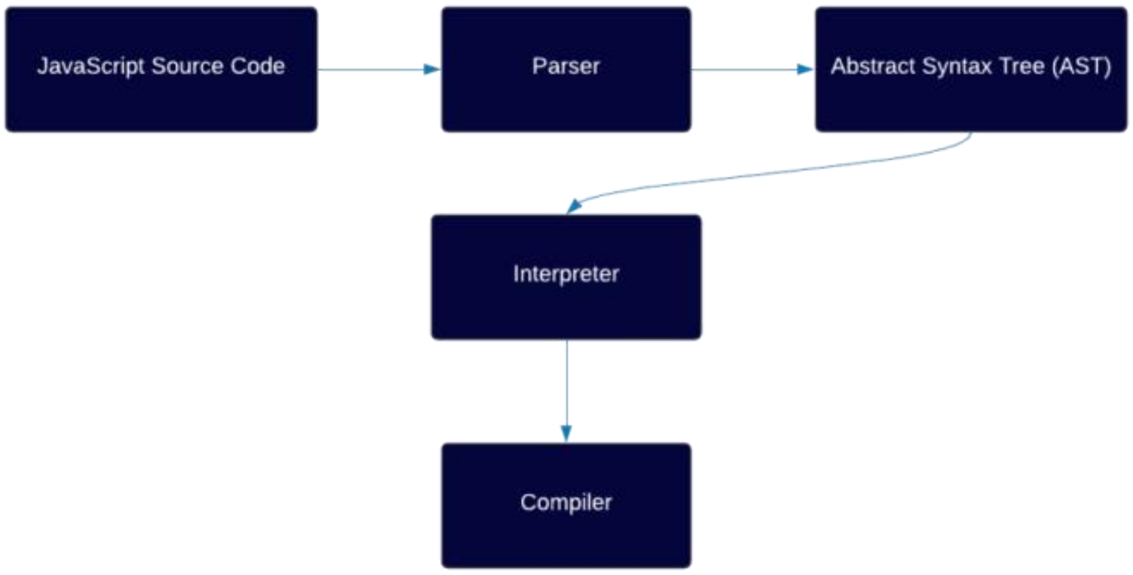
JS Engine:

A JavaScript engine is a software component that interprets and executes JavaScript code. It is a crucial part of web browsers and many other applications that rely on JavaScript for scripting and interactivity. It is the execution unit of browser.

List of JS Engines:

- Google Chrome - V8,
- Edge (Internet Explorer) - Chakra
- Mozilla Firefox - Spider Monkey
- Safari - JavaScript Core

How JS Engine works:



1. The JavaScript engine first performs lexical analysis or scanning, where it reads the source code character by character and groups characters into meaningful tokens, like keywords, identifiers, literals, and operators.

2.After lexical analysis, the engine parses the tokens into an Abstract Syntax Tree (AST), which represents the syntactic structure of the code.

The AST is a hierarchical tree-like structure that defines the relationships between different parts of the code, such as statements, expressions, and operators.

3.After the result of AST, the interpreter checks the code line by line and generate the byte code. But in some engines use Just-In-Time (JIT) compilation techniques to optimize the code. This involves analysing the code's execution patterns and applying various optimizations to make it run faster.

4.After compilation and optimization, the JavaScript code is executed. During execution, the engine manages the call stack, which keeps track of function calls and their contexts (local variables and state).

Garbage Collection:

While executing the code, the engine performs automatic memory management, known as garbage collection. It identifies and frees up memory occupied by objects that are no longer referenced, preventing memory leaks.

Global Execution Context:

In JavaScript, the Global Execution Context (GEC) is the top-level context or environment in which your JavaScript code is executed. It represents the outermost scope of your code, and it's created when your script is initially run. The GEC has a crucial role in managing and coordinating the execution of your entire JavaScript program. Let's dive into the phases of the Global Execution Context:

Creation Phase:

Variable Object (VO) Creation:

The GEC begins with the creation phase, during which JavaScript sets up the environment for your code. One of the first tasks is to create the Variable Object (VO), also known as the Global Object (in the case of a browser environment, it's the window object). The VO holds all globally declared variables and functions. Variables are initialized to undefined during this phase.

Scope Chain: The scope chain is established, which consists of the VO and any outer function or lexical scopes that might be present. In the global context, the scope chain typically contains only the global object.

Hoisting:

During the creation phase, function declarations are hoisted to the top of their containing scope. This means that functions can be called before they are defined in the code.

Variable declarations (using var) are also hoisted to the top of their containing function or global scope but are initialized with undefined. This is why you can access variables before they are declared without getting an error.

Execution Phase:

After the creation phase, JavaScript proceeds to the execution phase. This is where the actual code is executed line by line.

For each statement and expression in the code:

- Variables that have been declared (but not initialized) are assigned their initial value, which is undefined.
- Function declarations are made available for use throughout the entire scope.
- Code is executed sequentially, and any functions, variables, or operations are processed as they are encountered.
- If there are function calls, new execution contexts (Local Execution Contexts) are created for those functions, and the same creation and execution phase steps are followed within those contexts.
- The scope chain is used to resolve variable references, starting with the innermost scope and progressing outward until a match is found or the global scope is reached.

Cleanup:

Once the code in the GEC has been executed, the GEC itself doesn't get destroyed. However, the variables and functions declared within it remain accessible as properties/methods of the global object (e.g., window in a browser environment).

If a variable or function was declared using let or const in the global scope, they won't become properties of the global object.

Call Stack in JS:

In JavaScript, the call stack is a data structure used to keep track of function calls and their respective execution contexts during the execution of a program. It operates on a last-in, first-out (LIFO) basis, meaning that the most recently called function is the first one to be executed and removed from the stack when it's completed.

Promises in JS:

Promises in JavaScript provide a way to work with asynchronous operations in a more structured and manageable manner. They allow you to represent a value that might not be available yet but will be at some point in the future. Promises have become a standard mechanism for dealing with asynchronous code, and they simplify error handling and improve code readability.

Promise States:

A promise can be in one of three states: pending, fulfilled (resolved), or rejected.

- Initially, a promise is in the "pending" state, meaning it hasn't been resolved or rejected yet.
- When a promise's asynchronous operation completes successfully, it transitions to the "fulfilled" state with a resolved value.
- If an error occurs during the asynchronous operation, the promise transitions to the "rejected" state with a rejection reason (an error object).

Promise Creation:

You can create a promise using the Promise constructor, which takes an executor function as its argument. The executor function is called immediately and is passed two functions as arguments: resolve and reject.

You call resolve(value) when the asynchronous operation is successful and reject(reason) when an error occurs. The value is the result of the successful operation, and the reason is an error object.

Promise instances in JavaScript come with several instance methods that allow you to interact with and manipulate promises. These methods make it easier to work with asynchronous operations and handle the results or errors.

then(onFulfilled, onRejected):

The then method is used to specify what should happen when a promise is fulfilled (resolved) or rejected.

It takes two optional callback functions as arguments: onFulfilled and onRejected. These functions are called based on the promise's state.

If the promise is fulfilled, onFulfilled is called with the resolved value as its argument.

If the promise is rejected, onRejected is called with the rejection reason (an error object) as its argument.

The then method returns a new promise, allowing you to chain multiple then calls together.

catch(onRejected):

The catch method is a shorthand for specifying what to do when a promise is rejected. It takes a single callback function, onRejected.

This method is often used for error handling, especially when you want to handle errors for all preceding promise chain steps.

finally(onFinally):

The finally method allows you to specify a callback function, onFinally, that will be executed regardless of whether the promise is fulfilled or rejected. It is often used for cleanup operations.

The finally method returns a new promise that is resolved with the original promise's resolution value or rejection reason after onFinally has been executed.

Promise Chaining:

Promises allow you to chain asynchronous operations together using the .then() method. The .then() method is called when a promise is fulfilled and takes a callback function that processes the resolved value.

You can chain multiple .then() calls to create a sequence of asynchronous steps.

Additionally, you can handle errors by chaining a .catch() method at the end of the chain to catch any rejections in the promise chain.

Promise static methods:

Promise.all(iterable):

The Promise.all method takes an iterable (typically an array) of promises as its input and returns a new promise.

It waits for all the promises in the iterable to either fulfill or reject.

If all promises fulfill, the returned promise fulfills with an array of their resolved values in the same order as the original iterable.

If any promise in the iterable rejects, the returned promise immediately rejects with the reason of the first rejected promise.

Promise.any(iterable):

The Promise.any method takes an iterable of promises. It returns a new promise that fulfills with the value of the first promise in the iterable to fulfill. If all promises in the iterable are rejected, it will give aggregate error.

This method is useful when you want to work with the first successful result among multiple promises.

Promise.allSettled(iterable):

The Promise.allSettled method is also introduced in ECMAScript 2021 (ES12).

It takes an iterable of promises and returns a new promise that fulfills with an array of objects, each representing the outcome (fulfilled or rejected) of a promise in the iterable.

Each object in the array contains a status property (either "fulfilled" or "rejected") and a value or reason property, depending on whether the promise was fulfilled or rejected.

Promise.race(iterable):

The Promise.race() static method takes an iterable of promises as input and returns a single Promise . This returned promise settles with the eventual state of the first promise that settles.

window.fetch():

The fetch() method in JavaScript is a modern API for making network requests, primarily for fetching resources (like data or files) from a server or other online sources. It provides a more flexible and powerful way to work with HTTP requests compared to older techniques like XMLHttpRequest. fetch() returns a promise, which makes it easy to handle asynchronous operations, and it is widely used for making AJAX requests in web applications.

syntax: fetch(url);

json():

The json() method in JavaScript promises is used to parse the response body of an HTTP request (typically a fetch request) as JSON. It is a convenient way to work with JSON data that is often received from web APIs or other server endpoints.

async and await:

async and await are JavaScript language features introduced in ECMAScript 2017 (ES8) that simplify working with asynchronous code, making it more readable and easier to reason about. They are often used in conjunction with promises to write asynchronous code that appears more synchronous.

async Function:

The async keyword is used to declare a function as asynchronous.

An async function returns a promise implicitly, whether you explicitly return a promise or not.

Inside an async function, you can use the await keyword to pause the execution of the function until a promise is resolved.

Example:

```
async function fetchData() {

  const response = await fetch("https://example.com/api/data");

  const data = await response.json();

  return data;  }
```

await Operator:

The await keyword can only be used inside an async function.

It is used before a promise to pause the function's execution until the promise is resolved.

When used with a promise, await returns the resolved value of the promise.

If the promise is rejected, it throws an error that can be caught using a try...catch block or by chaining a .catch().

Error Handling:

You can handle errors in async functions using traditional try...catch blocks or by chaining a .catch() after an await expression.

If an error occurs in any async function, it will reject the promise returned by the function.

Benefits:

async and await make asynchronous code look more like synchronous code, improving readability and maintainability.

They simplify error handling, as you can use standard try...catch blocks for error handling.

They help avoid "callback hell" (also known as "pyramid of doom"), which can occur with deeply nested callbacks in asynchronous code.

Limitations:

await can only be used inside async functions.

async functions always return promises, even if they return a non-promise value.

Use Cases:

async and await are used for any asynchronous operation, such as making network requests, reading files, or working with databases.

They are commonly used with promises and other asynchronous patterns to write more readable and maintainable code.

try and catch block:

In JavaScript, the try and catch blocks are used for error handling, allowing you to handle and gracefully recover from errors or exceptions that may occur during the execution of your code. Additionally, JavaScript provides the Error object and its various subtypes to represent and work with errors in a structured manner.

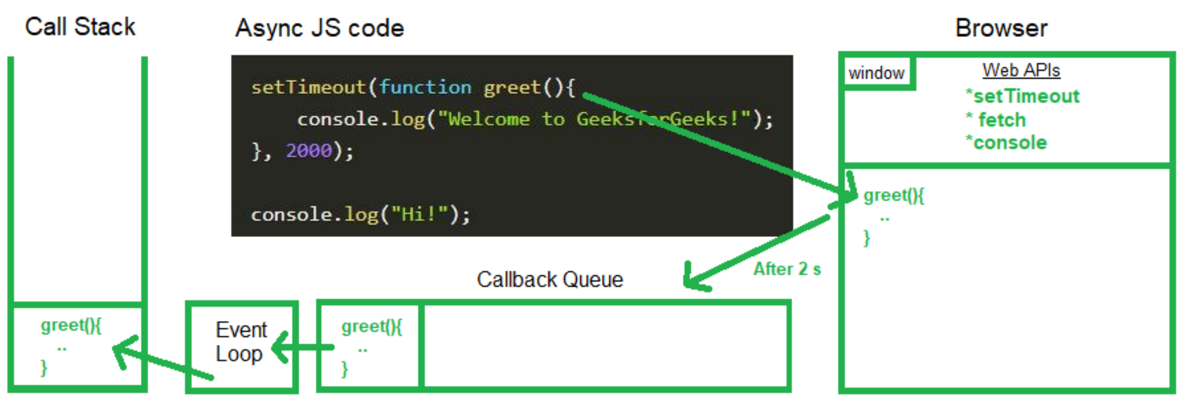
The try and catch blocks are used together to handle exceptions:

The try block contains the code where you anticipate an error might occur.

The catch block contains code to handle the error if it occurs.

If an error occurs in the try block, the code in the catch block is executed.

How Asynchronous code works?



```
setTimeout(function greet(){
    console.log("Welcome to GeeksforGeeks!");
}, 2000);

console.log("Hi!");
```

Asynchronous code execution in JavaScript is a fundamental concept that allows you to perform tasks that might take time to complete without blocking the main execution thread.

setTimeout and setInterval:

When you call setTimeout or setInterval, the provided callback function (or code) is not executed immediately. Instead, it's added to the browser's Web API environment, and a timer is set.

After the specified delay for setTimeout or at each interval for setInterval, the callback is moved to the callback queue.

The JavaScript engine, while executing code on the main thread, checks the callback queue during each cycle of the event loop to see if there are any functions waiting to be executed.

When the callback reaches the front of the queue and the main thread is free (not executing any other code), the callback is executed.

Callback queue:

A callback queue is a queue of tasks that are executed after the current task. The callback queue is handled by the JavaScript engine after it has executed all tasks in the microtask queue.

Microtask Queue:

Microtask Queue is like the Callback Queue, but Microtask Queue has higher priority. All the callback functions coming through Promises and Mutation Observer will go inside the Microtask Queue. For example, in the case of . fetch(), the callback function gets to the Microtask Queue.

Event loop:

- The event loop is a continuous process that continually checks the state of the callback queue while the execution stack is empty.
- If the execution stack is empty, the event loop takes the first task (callback) from the callback queue and pushes it onto the execution stack.
- The callback function is then executed in the order it was added to the queue.

Prototypes:

Prototype inheritance is a fundamental concept in JavaScript that enables object-oriented programming and code reusability. In JavaScript, all objects have a prototype, and they can inherit properties and methods from their prototype object.

Every JavaScript object has a prototype, which is another object.

The prototype object contains properties and methods that can be inherited by other objects.

Prototype Chain:

When you access a property or method on an object, JavaScript first checks if that property or method exists directly on the object.

If it doesn't find the property or method, JavaScript looks for it in the object's prototype.

This process continues up the prototype chain until the property or method is found or the chain ends at the global Object.prototype.

prototype Property:

Functions in JavaScript have a special property called prototype.

When you create an object using a constructor function, the object's prototype is set to the constructor's prototype property.

You can add properties and methods to the constructor's prototype, and they will be inherited by all objects created from that constructor.

Modules in JS:

JavaScript modules are a way to organize and encapsulate code into reusable and maintainable units. They provide a mechanism for breaking up your JavaScript code into smaller, self-contained files or modules, each with its own scope. Modules help improve code structure, promote code reusability, and prevent global scope pollution. In modern JavaScript, there are different module systems, including CommonJS, AMD (Asynchronous Module Definition), and ES6 (ECMAScript 2015) modules. The most commonly used module system in modern web development is ES6 modules.

ES6 Modules (ESM):

ES6 modules are the official standard for module management in JavaScript, and they are supported natively in modern browsers and in Node.js (since Node.js v13.2.0).

1.Named Export:

Exporting from a Module:

To export values (variables, functions, classes, etc.) from a module, you use the export keyword.

Importing from a Module:

To import values from a module into another module, you use the import statement.

Example:

Export:

```
// myModule.js
export const myVariable = 42;

export function myFunction() {
  // ...
}

export class MyClass {
  // ...
}
```

import:

```
// anotherModule.js
import { myVariable, myFunction, MyClass } from './myModule.js';

console.log(myVariable); // 42
myFunction();
const instance = new MyClass();
```

While importing the name should be same.We can able to import more than one object/function/variables using named export.

Default Exports:

You can export a default value from a module using the export default syntax.

Export:

```
// myModule.js
const defaultExport = 42;
export default defaultExport;
```

import:

```
// anotherModule.js
import myDefaultExport from './myModule.js';
```

```
console.log(myDefaultExport); // 42
```

we can able to import only one object/function/variable using default export and name need not to be same in the original js file.

CommonJS Modules (CJS):

CommonJS modules are primarily used in Node.js for server-side JavaScript development. They use require() to import modules and module.exports to export values.

Example:

Exporting:

```
// myModule.js

const myVariable = 42;

function myFunction() {

  // ...

}

module.exports = {

  myVariable,

  myFunction,

};
```

Importing from a Module:

```
// anotherModule.js

const { myVariable, myFunction } = require('./myModule');

console.log(myVariable); // 42

myFunction();
```

Arguments Object:

The arguments object is an array-like object available within the body of every function in JavaScript.

It allows you to access the arguments that were passed to the function, even if the function didn't explicitly declare named parameters.

You can access the arguments using numeric indices (e.g., arguments[0], arguments[1], etc.).

The arguments object is not a true array; it is an array-like object, so it lacks array methods like map, forEach, and reduce. You can convert it to a real array using methods like Array.from(arguments) or the spread operator ([...arguments]).

Arity of a Function:

The term "arity" refers to the number of arguments a function accepts.

For example, a function that takes no arguments has an arity of 0, a function that takes one argument has an arity of 1, and so on.

In JavaScript, "unary," "binary," and "ternary" functions refer to functions that accept a specific number of arguments: one, two, and three, respectively. These terms are often used to describe the arity (the number of arguments) of a function.

Unary Function:

A unary function is a function that accepts one argument.

Unary functions are quite common in JavaScript, as many built-in functions and methods take a single argument.

Binary Function:

A binary function is a function that accepts two arguments.

Binary functions are also common in JavaScript and are often used for operations that involve two values.

Ternary Function:

A ternary function is a function that accepts three arguments.

Ternary functions are less common than unary or binary functions and are typically used for specific scenarios where three arguments are needed.

n-ary function:

An "n-ary" function in JavaScript is a function that accepts a variable number of arguments. Unlike unary (1 argument), binary (2 arguments), or ternary (3 arguments) functions, an n-ary function can take any number of arguments, including zero.

JS Classes:

JavaScript classes are a way to create objects with shared properties and methods, following a blueprint or template. They were introduced in ECMAScript 2015 (ES6) and provide a more structured and object-oriented approach to creating and managing objects in JavaScript. Classes in JavaScript are a fundamental part of modern web development. Here's an overview of how classes work in JavaScript

Class Declaration:

You can declare a class using the class keyword, followed by the class name. The class can have a constructor method that is called when you create a new instance of the class. Class methods can also be defined within the class.

EX:

```
class MyClass {

 constructor(property1, property2) {

   this.property1 = property1;

   this.property2 = property2;  }
```

```
  myMethod() {
    // Method logic
  }
}
```

Creating Instances:

To create instances (objects) of a class, you use the new keyword followed by the class name, passing any required arguments to the constructor.

EX: const myObject = new MyClass("value1", "value2");

Constructor:

The constructor method is a special method called when a new instance of the class is created. It is used to initialize the object's properties.

EX:

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

Methods:

You can define methods within a class. These methods can be called on instances of the class.

EX:

```
class Circle {
  constructor(radius) {
    this.radius = radius;
  }
  getArea() {
    return Math.PI * this.radius ** 2;
  }
}
const myCircle = new Circle(5);
console.log(myCircle.getArea()); // Output: 78.53981633974483
```

Inheritance:

JavaScript classes support inheritance, allowing you to create a subclass (child class) that inherits properties and methods from a superclass (parent class).

EX:

```
class Animal {
  constructor(name) {
    this.name = name;
  }
  makeSound() {
    console.log("Animal makes a sound");
  }
}
class Dog extends Animal {
  makeSound() {
    console.log("Dog barks");
  }
}
const myDog = new Dog("Buddy");
myDog.makeSound(); // Output: "Dog barks"
```

Static Methods:

Static methods are defined on the class itself, not on instances. They are called on the class, not on instances of the class.

EX:

```
class MathUtils {
  static add(x, y) {
    return x + y;
  }
  static subtract(x, y) {
    return x - y;
  }
}
console.log(MathUtils.add(5, 3));      // Output: 8
console.log(MathUtils.subtract(10, 4)); // Output: 6
```

Private Class Fields:

In newer JavaScript versions (ES2022 and beyond), you can use private class fields to create private variables within a class.

EX:

```
class Example {
  #privateField = 42;
  getPrivateField() {
    return this.#privateField;
  }
}
```