**How JavaScript Works & Execution Context**

1. Execution Context is the wrapper/environment around our existing code.
2. Execution Context is of 2 types :-   
   a. Global Execution Context (GEC)  
   b. Functional/Local Execution Context(LEC)
3. Everything in JavaScript happens inside an Execution Context.
4. Execution Context has 2 components.  
   a. Memory component a.k.a. variable environment -> Here all variable and functions are stored as key-value pairs.  
   b. Code component a.k.a. Thread of execution -> > Here Code is executed one line at a time.
5. JavaScript is a synchronous, single threaded language. It means JS can execute one command at a time in a specific order.
6. Global Execution Context comprises of the following -
7. Global Object which is window object.
8. 'this' -> in GEC this refers to global object or window object.
9. Your code
10. Functional/Local Execution Context comprises the following –
11. Our Code
12. ‘this’

**How Javascript Code is Executed & Call Stack**

Phase 1 : Memory Creation Phase ---> This is when hoisting happens.

* JavaScript skims through the whole code and identifies the variables and functions
* At the time of skimming variables are assigned "undefined" as value.
* The functions are literally copied/stored in execution context.

Phase 2 : Code Execution Phase

* JavaScript program is executed line by line
* Variables identified in phase 1 are assigned their desired value and functions are invoked.
* Whenever JavaScript functions are invoked, a local execution context is created inside global execution context.

What is Call stack?

1. JavaScript manages execution of all local and global execution contexts using Call Stack. Indirectly it stores Execution Context.
2. Call Stack is a stack Data Structure wherein global execution context is stored at the bottom of stack, and local execution context are placed above it.
3. If execution context is created, it is pushed into the stack.
4. Once the execution context completes its execution, it is popped out of stack.
5. Call Stack are known by other fancy names.   
   Execution context Stack, Program Stack, Control Stack, Runtime Stack, Machine Stack.

**Hoisting in JavaScript**

Hoisting is a behavior in which we can access variables and functions even before they are declared.

Another Definition from mdn :- JavaScript Hoisting refers to the process whereby the interpreter appears to move the declaration of functions, variables or classes to the top of their scope, prior to execution of the code.

In terms of variables and constants, keyword var is hoisted and initialized with undefined, let and const are hoisted but not initialized, thus they lie in temporal dead zone during hoisting.

Watch this video(by Geeky Shows) for Hoisting - <https://www.youtube.com/watch?v=WooWDj9q188>

Example Of Variable Hoisting –

Using “num” even before it is declared.

console.log(num);   
var num;   
num = 6;  
console.log(num);  
Output:- undefined  
 6

Example Of Function Hoisting –

catName("Tiger");  
function catName(name) { console.log("My cat's name is " + name); }  
Output:- My cat's name is Tiger.

**How function works with JS & Variable Environment.**

**Shortest JS Program, window & this keyword.**

1. An Empty file is the shortest JS Program.
2. At global level "this" points to global window object. "this" is created for both functional execution context and global execution context.
3. Window is global object which is created along with Global Execution Context.

**Undefined v/s not defined in JavaScript**

1. null -> null is an actual value. typeof(null) is object.
2. undefined -> it means variable is declared but not initialized. typeof(undefined) is undefined.

**Scope Chain, Scope & Lexical Environment**

1. lexical means one inside other.
2. Scope chain -> Chain of References to the outside lexical environment.

**Let & const, Temporal Dead Zone.**

1. Temporal Dead Zone is prominent for "let" & "const". It is the time when a let variable is hoisted till it is initialized some value(time period in which let & const variable can’t be accessed).
2. Reference Error -> usually happens when we try to access a variable that is present in temporal dead zone or not present in our program.
3. Type Error -> usually happens when we try to re-assign a variable of type const.
4. Syntax Error -> there is syntactical error in the code.
5. "var", "temp" & "const" are also hoisted. var is hoisted into Global scope whereas let & const hoisted inside separate space (either Script or Block)

**Block Scope & Shadowing in JS**

1. Block in JavaScript combines multiple statements into one group. We group multiple statements together in block so that we can use it where JavaScript expects one statement. Simplest block is {}.
2. Block scope refers to all variables and functions that we can use inside the block.
3. let & const are block scoped. So let & const are accessible inside block.
4. **Shadowing**: same name of variable but different usage according to initialization.
5. Illegal Shadowing : let a = 20; {var a = 20;}

**Closure in JS**

1. Function bundled with lexical environment/reference is closure. In simple words, it is a feature in which inner function has access to outer function variable.
2. Each and every function in JavaScript has access to its outer lexical environment (means access to variables and functions which are present in environment of its parent). So even when a function is executed in some other scope which is not its original scope, it still remembers the outer lexical environment where it was originally present in the code.
3. Common uses of closures (Module Design Pattern, Currying, Function like once, memorize, maintaining state in async world, setTimeouts, Iterators & many more....)

**setTimeout + Closures interview Questions**

**First Class Functions**

1. Function Statement / Function Declaration / Function Definition  
   function a() {console.log("a called");}
2. Function Expression

var b = function() {console.log("b called");}

NOTE: DIFFERENCE BETWEEN FUNCTION EXPRESSION AND FUNCTION STATEMENT IS "HOISTING"

1. Anonymous Function

These are function without a name. They are used at places where functions are used as values.

e.g. var b = function() {console.log("b called");}

Function Statement cannot be used as Anonymous values.

1. Named Function Expression.

e.g. var a = function xyz() {console.log("xyz called");}

1. First Class Functions

The ability of functions to be used as values and can be passed as argument to another function and to be returned from functions.

In Simple words you can treat functions like variables.

**Callback Functions in JS. ft. Event Listeners.**

Function being passed as an argument to another function is called **callback function**.

The function taking that argument is called **higher order function**.

for example :-

function a() {

console.log("Namaste")

}

function y(x){

x();

}

y(a);

here y is higher order function and x is callback function.

Other examples of higher order functions are setTimeOut(), map(), filter(), reduce(), etc.

**Asynchronous JavaScript & Event Loops**

1. **Event Loop**: - The event loop is a constantly running process that monitors both the callback queue, and the call stack.

If the call stack is not empty, the event loop waits until it is empty and places the next function from the callback queue to the call stack.

If the callback queue is empty, nothing will happen.

1. **Call back queue (Task Queue)**: - This is where our asynchronous code gets pushed to, and waits for the execution.
2. **Micro task queue**: - Apart from Callback Queue, browsers have introduced one more queue which is “Job Queue”, reserved only for new Promise() functionality.

So when you use promises in your code, you add .then() method, which is a callback method.

These `thenable` methods are added to Job Queue once the promise has returned/resolved, and then gets executed.

Note : - Micro-task queue has more priority than callback queue.

**JS Engine Exposed, Google V8 architecture.**

JavaScript Runtime Environment includes

1. JS Engine - Heart of JavaScript Run time Environment.
2. Set of API's
3. Event Loop
4. callback queue
5. micro-task queue

Js Engine is software which follows 3 phases.

1. Parsing: - Entire code is given to Syntax parser which converts our code into Abstract Syntax Tree.
2. Compilation: - JavaScript uses JIT Compiler to compile our code. There are different kinds of optimizations performed in JIT for example: - in-lining, copy elision & inline caching.
3. Execution: - This phase uses Memory Heap and Call Stack for Execution. It also has Garbage collector which frees up memory heap space. Garbage Collector uses Mark and Sweep Algorithm to free up memory space.

**Trust Issues with setTimeOut**

**Higher Order Functions**

**Map, Filter & Reduce in JavaScript.**

**Immediately Invoked Function Expression (IIFE)**

It is a JS function that runs as soon as it is defined.   
It is also called as **Self executing anonymous function**.  
It contains 2 parts.

1. Anonymous function enclosed within group operator'()'.
2. Adding () at the end of function expression.

For example :-   
1. (function(){console.log("IIFE");})()  
2. (function(a,b){console.log(a+" "+b);})(10,20); // output 10 20

**Data Types in JS**

There are 7 primitive data types: string, number, bigint, boolean, null, undefined, and symbol.  
Undefined: - It is used for variable that does not exist or have not been assigned a value.  
Hint: NBUS(Number,Null,BigInt,Boolean,Undefined,String,Symbol)

**How to create a promise in JavaScript?**

1. A promise is an object that may produce a value sometime in the future: either a resolved value, or a reason that it's not resolved (e.g., a network error occurred).
2. A promise may be in one of 3 possible states:
3. Fulfilled
4. Rejected
5. Pending.
6. Promises are not callbacks. A promise represents the future result of an asynchronous operation.

**Creating Promises.**

const promise = new Promise((resolve, reject)=>{

resolve ("promise resolved");

or

reject ("promise rejected, error occurred");

or some time taking code.

// it is not necessary to have resolve and reject functions. What’s necessary is the callback function which is passed as argument to new Promise();

})

KEPT SOME WHERE IN THE CODE

promise

.then((response)=>{

console.log(response) // will print "promise resolved".

})

.catch(error=>{

console.log(error); // will print "promise rejected, error occured"

})

<https://www.javascripttutorial.net/es6/javascript-promises/>

**Common Example Of Asynchronous Tasks**

1. Fetching Data from an API:
2. Timers: setTimeOut,setInterval
3. Promises
4. File I/O (Node.js)  
   const fs = require('fs');  
    fs.readFile('file.txt', 'utf8', (err, data) => {  
    if (err) {// Handle error}   
    else {// Process file data}  
    });
5. User Interface Interactions  
   document.getElementById('myButton').addEventListener('click', () => {  
    // Code to run when the button is clicked  
    });
6. Database Operations  
    const mongoose = require('mongoose');  
    mongoose.connect('mongodb://localhost/mydb', { useNewUrlParser: true }).then(() => {  
    // Database connected, perform operations  
    }).catch(err => {  
    // Handle database connection error  
    });

**How to handle asynchronous calls/request in javascript?**

Using Async-Await, using callbacks and using Promises.

**Explain Async-Await along with example.**

The async / await is syntactic sugar for promises.  
https://www.javascripttutorial.net/es-next/javascript-async-await/

**How to copy an object in JavaScript and difference between shallow copy and deep copy?**

3ways of copying an object.

1. Object.assign()
2. Spread operator
3. JSON.stringify() and JSON.parse() methods

Syntax:-

const person = {

firstName: 'John',

lastName: 'Doe'

};

let p1 = {...person}; // performs shallow copy.  
let p2 = Object.assign({}, person); // performs shallow copy.  
let p3 = JSON.parse(JSON.stringify(person)); // performs deep copy

**Shallow Copy**

If you use the assignment operator for a reference value, it will not copy the value. Instead, both variables will reference the same object in the memory.

**Deep Copy**

The newly created variable will be disconnected from variable it is connected.

**What is Babel?**

Babel is a transpiler that is mainly used to convert ECMAScript 2015+ code into ES5 code, which is compatible version of JavaScript in current and older browsers.

Also it handles converting JSX to JavaScript code so that React can render our applications in browser.

**What is Webpack?**

Webpack is a dependency analyzer and module bundler.

For example, if module A requires B as a dependency, and module B requires C as a dependency, then webpack will generate a dependency map like C -> B -> A.

In practice it is much more complicated than this, but the general concept is that Webpack packages modules with complex dependency relationships into bundles.

Regarding webpack's relationship with babel: When webpack processes dependencies, it must turn everything into JavaScript because webpack works on top of JavaScript.

**.call(), apply(), bind() in js**

**Call**

It calls the method, taking the owner object as an argument. The keyword this refers to the “owner” of the function or the object it belongs to. <https://www.javascripttutorial.net/javascript-call/>

Call allows us to invoke a function and specify the value of ‘this’ explicitly.

for example:-

*var printEmployeeDetails = function() {console.log(this); }*

*var empDetail1 = { name: "Shivam", id: "234412"}*

*var empDetail2 = {name: "Raj", id: "434556"}*

*printEmployeeDetails.details.call(emp2); // in details function, this will refer to emp2;*

**apply**

The apply() method is used to write methods, which can be used on different objects. It is different from the function call() because it takes arguments as an array.

Just like Call, apply allows us to invoke a function and specify the value of this explicitly, but it takes an array as arguments.

*var printEmployeeDetails = function(state,country) {console.log(this.name+ “ ”+ this.state+“ ”+this.country); }*

*var empDetail1 = { name: "Shivam", id: "234412”};*

*var empDetail2 = { name: "Raj",id: "434556"}*

*printEmployeeDetails.apply(empDetails1,[“Texas”, “USA”]);*

*printEmployeeDetails.apply(empDetails2,[“Delhi”, “INDIA”]);*

**Bind**

for example:- The bind() method creates a new function. It binds an object to function. This object has to be passed as argument to bind.

Bind creates copy of existing function, which can be invoked later.

*var printEmployeeDetails = function(state,country) {console.log(this.name+ “ ”+ this.state+“ ”+this.country); }*

*var empDetail1 = { name: "Shivam", id: "234412”};*

*var empDetail2 = { name: "Raj",id: "434556"}*

*var newPrintEmployeeDetails1 = printEmployeeDetails.bind(empDetails1,“Texas”, “USA”);*

*var newPrintEmployeeDetails2 = printEmployeeDetails.bind(empDetails2,“Delhi”, “INDIA”);*

*newPrintEmployeeDetails1();newPrintEmployeeDetails2();*

**Partial Function & Currying in JavaScript**

Partial Function:

In JavaScript, a partial function refers to a function that has some of its arguments pre-specified or "partially applied." Partial application is a technique where you create a new function by fixing a certain number of arguments of an existing function, leaving the rest to be provided later when the new function is called.

Example1:

function add(a, b) { return a + b; }  
const add5 = add.bind(null, 5);  
console.log(add5(3)); // Output: 8

Explanation:

* 1. We have a simple add function that takes two arguments, a and b, and returns their sum.
  2. We create a new function called ‘add5’ by using the ‘bind’ method on the ‘add’ function. ‘add.bind(null, 5)’ binds the ‘add’ function to a new function (‘add5’) and sets ‘5’ as the first argument (‘a’) when ‘add5’ is called. The ‘null’ argument in ‘bind’ is used as the context (‘this’ value) for the function, but it's not relevant in this example.
  3. Now, add5 is a partially applied function that expects only one argument (b) to be passed when it's called. It effectively "remembers" that the first argument is 5.
  4. When we call add5(3), we're providing the second argument (b) with the value 3. The add5 function uses the pre-set 5 as the first argument (a) and adds it to 3, resulting in 8.

Example2:

function greet(greeting, firstName, lastName) { console.log(`${greeting}, ${firstName} ${lastName}`); }

// Create a partial function for friendly greetings  
const greetFriendly = greet.bind(null, 'Hello');

// Use the partial function  
greetFriendly('John', 'Doe'); // Output: Hello, John Doe  
greetFriendly('Alice', 'Smith'); // Output: Hello, Alice Smith

1. We have a greet function that takes three arguments: greeting, firstName, and lastName. It logs a friendly greeting with the provided information.
2. We create a partial function called greetFriendly using bind. We fix the greeting argument to 'Hello'. Now, greetFriendly is a function that expects only firstName and lastName to be provided when called.
3. When we use greetFriendly, we don't need to specify the greeting each time because it's already pre-set to 'Hello'. We only provide the firstName and lastName.
4. When we call greetFriendly('John', 'Doe'), it logs "Hello, John Doe". Similarly, calling greetFriendly('Alice', 'Smith') logs "Hello, Alice Smith".

Partials are basically functions that return functions with some already predefined arguments and need some arguments to be completed. In above example greetFriendly & add5 are partials.

Currying is applied on partial functions.

Currying is when you break down a function that takes multiple arguments into a series of functions that take only one argument.

Example 1:

// Non-curried function  
function add(a, b) { return a + b; }  
console.log(add(2, 3)); // Output: 5

// Curried function  
function curriedAdd(a) {  
 return function(b) {  
 return a + b;  
 };  
}

const add2 = curriedAdd(2); // Fix the first argument

console.log(add2(3)); // Output: 5

Example 2:

function multiply(a, b, c, d) { return a \* b \* c \* d; }

// Curried version with 4 arguments

function curriedMultiply(a) {  
 return function(b) {  
 return function(c) {  
 return function(d) {  
 return a \* b \* c \* d;  
 };  
 };  
 };  
}

// Usage

const multiplyBy2 = curriedMultiply(2); // Fix the first argument  
const multiplyBy2And3 = multiplyBy2(3); // Fix the second argument  
const multiplyBy2And3And4 = multiplyBy2And3(4); // Fix the third argument  
console.log(multiplyBy2And3And4(5)); // Output: 120 (2 \* 3 \* 4 \* 5)

Explanation:

1. We start with a multiply function that takes four arguments and returns their product.
2. We create a curried version of the multiply function called curriedMultiply. This curried function takes one argument at a time, returning a new function each time.
3. We create partially applied functions:

multiplyBy2 fixes the first argument as 2.

multiplyBy2And3 fixes the first two arguments as 2 and 3.

multiplyBy2And3And4 fixes the first three arguments as 2, 3, and 4.

Finally, when we call multiplyBy2And3And4(5), it's equivalent to curriedMultiply(2)(3)(4)(5), and it returns 120, which is the product of 2, 3, 4, and 5.

Note: multiply(2)(3)(4)(5) is same as multiplyBy2And3And4(5).

**Infinite Currying**

It is functional programming technique in which a function can accept an indefinite number of arguments, and return a new function after each argument application.

function infiniteCurry(fn, initialValue = 1) {  
 return function curried(...args) {  
 if (args.length === 0) {   
 // If no arguments are provided, return the accumulated value  
 return initialValue;  
 } else {  
 // Accumulate the result and return a new curried function  
 const result = fn(initialValue, args[0]);  
 return infiniteCurry(result, ...args.slice(1));  
 }  
 };

}

const multiply= infiniteCurry((x, y) => x \* y);

const result = multiply(1)(2)(3)(4)(); // This will return 10

In this example, infiniteCurry is a function that takes another function fn as its first argument and an optional initialValue to start the accumulation process. It returns a curried function curried that can accept any number of arguments using the currying syntax.

When you call multiply(1)(2)(3)(4)(), it accumulates the values using the add function and returns 24. The final () is used to signal the end of the currying chain and retrieve the accumulated result.

**Throttling in JavaScript**

Throttling is a technique in which, no matter how many times the user fires the event, the attached function will be executed only once in a given time interval.

Example of throttling

Imagine yourself as a 7-year-old toddler who loves to eat chocolate cake!. Today your mom has made one, but it's not for you, it's for the guests! You, being spunky, keep on asking her for the cake. Finally, she gives you the cake. But, you keep on asking her for more. Annoyed, she agrees to give you more cake with a condition that you can have the cake only after an hour. Still, you keep on asking her for the cake, but now she ignores you. Finally, after an interval of one hour, you get more cake. If you ask for more, you will get it only after an hour, no matter how many times you ask her.

This is what throttling is!

**Debouncing in JavaScript**

In the debouncing technique, no matter how many times the user fires the event, the attached function will be executed only after the specified time, once the user stops firing the event.

Consider the same cake example.

This time you kept on asking your mom for the cake so many times that she got annoyed and told you that she will give you the cake only if you remain silent for one hour. This means you won’t get the cake if you keep on asking her continuously - you will only get it one hour after last time you ask, once you stop asking for the cake. This is debouncing.

Debouncing in JavaScript is a practice used to improve browser performance. There might be some functionality in a web page which requires time-consuming computations. If such a method is invoked frequently, it might greatly affect the performance of the browser, as JavaScript is a single threaded language.

Debouncing is a programming practice used to ensure that time-consuming tasks do not fire so often, that it stalls the performance of the web page.

In other words, it limits the rate at which a function gets invoked.

**Event Delegation in JavaScript**

Instead of adding an event listener to each and every nested element, we can add an event listener to a parent element and call an event using the “.target” property of the event object.

For Example.

Suppose there are 3 nested div's (having id's as -> child, parent, grandparent).   
Each div has click event listener attached with it.  
Now without event delegation, if we click on child div, then click event associated with parent and grandparent will also be triggered.  
We did not want that un-necessary triggering of events to happen.  
So what we do is we add click event listener only to grand-parent, and use target property to determine on which div we have clicked.

**"This" in JavaScript**

In JavaScript, this keyword refers to an object. Which object depends on how this is being invoked (used or called).

Content from w3schools.com

1. In an object method, this refers to the object.  
   Example:   
   *const person = {  
    firstName: "John",  
    lastName: "Doe",  
    id: 5566,  
    fullName : function() {  
    return this.firstName + " " + this.lastName;  
    }  
   };*

*person.fullName(); // output -> John Doe*

1. Alone, this refers to the global object.

Example

let x = this;

console.log(x); // prints global object

1. In strict mode, when used alone, this also refers to the global object:

Example

'use strict'

let x = this;

console.log(x); // prints global object.

1. In a function, this refers to the global object.

Example

function myFunction() {

return this;

}

console.log(myFunction()); // prints global object.

1. In a function, in strict mode, this is undefined.

Example

"use strict";

function myFunction() {

return this;

}

console.log(myFunction()); // return undefined.

1. In an event, this refers to the element that received the event.

Example

<button onClick="this.style.display='none'">Click to Remove Me!</button> // on clicking this button, the button will be removed.

1. Methods like call(), apply(), and bind() can refer this to any object.

const obj = {

bool: true,

myFunc: myFunc,

}

obj.myFunc()

In above example this will point to Object 'obj'.

When we had nothing left of the .(dot) so it defaulted to the window object. But in this example, we have the object obj.

If we do

myFunc()

We again get the 'window' object. So, we can see that the value of this depends on how and where are we doing the calling.

What we did above in example is called 'implicit binding'.

There is another way to use 'this.

'Explicit binding' is when you force a function to use a certain object as its 'this'.

We can do explicit binding using .call, .bind & .apply

**Flattening an array in JavaScript**

Input: [1,2,3,4,5,[6,[7,8,9]]]

Output: [1,2,3,4,5,6,7,8,9]

*const flattenedArray = (arr) => {  
 const newFlatArray = arr.reduce(accumulator,item) => {  
 if(item.isArray) {accumulator = accumulator.concat(flattenedArray[item]);}  
 else {accumulator = [...accumulator,item];}  
 return accumulator;   
 }  
 return newFlatArray;*

*}*

**Flattening of object in JavaScript**

**Polyfills for forEach**

*Array.prototype.ourForEach = function (callBack) {  
//this is representing the array on which ourForEach is used.  
 for (var i = 0; i < this.length; i++) {  
 callback(this[i], i, this) // currentValue, index, array  
 }  
};*

*names.ourForEach(name => console.log(name));*

**Polyfills for map**

*Array.prototype.ourMap = function(callback) {  
 var arr = [];  
 for (var i = 0; i < this.length; i++) {  
 arr.push(callback(this[i], i, this))  
 }  
return arr // finally returning the array  
}*

**Polyfills for filter**

*Array.prototype.filterAlbums = function(callback, context) {  
 arr = []  
 for (var i = 0; i < this.length; i++) {  
 if (callback.call(context, this[i], i, this)) {  
 arr.push(this[i])  
 }  
 }  
return arr;  
}*

**Polyfills for call**

**Polyfills for Apply**

**Polyfills for Bind**

**Phases of JavaScript Event.**

1. Capturing Phase
2. Target Phase
3. Bubbling Phase

When a click event happens (or any other event that propagates):

* The event travels down from root of DOM hierarchy to target element (capture phase)
* The event occurs on the target (the target phase)
* Finally, the event bubbles up through DOM hierarchy, from target element to root element (the bubble phase).
* This mechanism is named *event propagation*

**Event Bubbling**

Event Bubbling is a concept in which an event triggered on a nested element will "bubble up" through its parent elements all the way to the root of the document, invoking event handlers attached to those parent elements along the way.

**JavaScript Argument Object (from w3schools)**

JavaScript functions have a built-in object called the arguments object.  
The argument object contains an array of the arguments used when the function was called (invoked).

function sum() {  
 let result = 0;  
 for (let i = 0; i < arguments.length; i++) {result += arguments[i]; }  
 return result;  
}

console.log(sum(1, 2, 3, 4)); // Output: 10

**setInterval**

The setInterval() method calls a function at specified intervals (in milliseconds).  
The setInterval() method continues calling the function until clearInterval() is called, or the window is closed.  
*myInterval = setInterval(function, milliseconds);*

**clearInterval**

The clearInterval() method clears a timer set with the setInterval() method.  
*clearInterval(myInterval)*

**setTimeOut**

The setTimeout() method calls a function after a number of milliseconds.  
*myTimeout = setTimeout(function, milliseconds);*

**clearTimeout**

The clearTimeout() method clears a timer set with the setTimeout() method.  
*clearTimeout(myTimeout);*

**blur**

The blur() method removes focus from a window.

**focus**

The focus() method sets focus to a window.

**CORS, Preflight Request, OPTIONS Method | Access Control Allow Origin Error**

**CORS**

Cross Origin Resource Sharing. It is a mechanism which uses additional HTTP headers to tell the browser, whether a specific web app can share resource with another web app, but both web apps should have different origin.

Break Down of CORS

By default, web browsers enforce the Same-Origin Policy (one domain/origin cannot access resources of another domain/origin and vice-versa).

In certain situations, we may want to make cross-origin requests from our web page to access resources hosted on another domain. For such situation, CORS provides a mechanism to explicitly declare which origins (domains) are allowed to access their resources.

Key CORS Headers are as follows:

* 1. Access-Control-Allow-Origin: Specifies the origins that are permitted to access the resource.
  2. Access-Control-Allow-Methods: Lists the HTTP methods allowed for the resource (e.g., GET, POST, PUT, DELETE).

**Pre Flight Request(OPTIONS)**

Sometimes, before making the actual request (like a GET or POST), the browser sends a "Preflight" request. We can think of it as asking, "Is it okay if I make this request?"

The Preflight request is like a polite way to check with the other website if it's safe to proceed.

The other website (the one you're requesting data from) responds to this Preflight request with CORS headers

If the Preflight response says it's okay, then the browser goes ahead and makes the actual request.

**Difference B/W Weak Set And Set.**

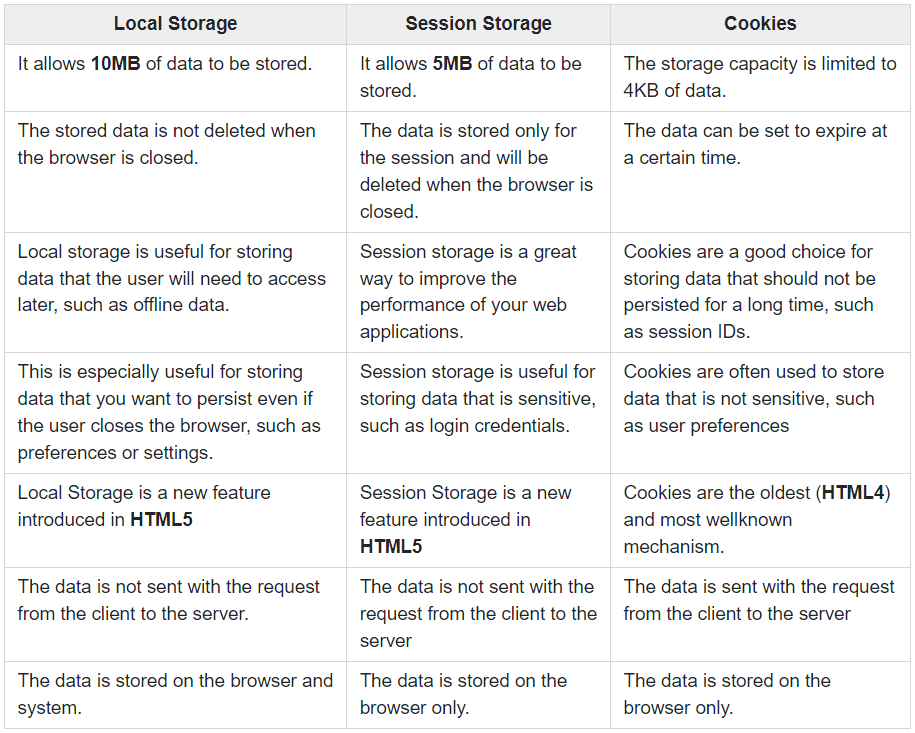
1. Weak Set cannot be iterated.
2. Only objects can be stored in WeakSet.

**Difference B/W Weak Map & Map**

1. It cannot be iterated.
2. Only object can be stored in values in WeakMap.

**LocalStorage v/s Session Storage v/s Cookies**

<https://www.tutorialspoint.com/difference-between-local-storage-session-storage-and-cookies-in-javascript>



**JavaScript Array methods**

1. forEach()  
2. filter()  
3. map()  
4. reduce()  
5. push()  
6. pop()  
7. shift() -> removes First element in the array.  
8. unshift() -> add an element at starting of array.

**9. slice()**

The slice() method returns selected elements in an array, as a new array.  
The slice() method selects from a given start, up to a (not inclusive) given end.  
The slice() method does not change the original array.

Syntax:- **array.slice(start, end)**  
**a. start** (Optional; Defines Start index in array; Default is 0; Negative numbers select from the end of the array)  
**b. end**(Optional; Defines end index in array; Default is last index; Negative numbers select from the end of the array)

*const fruits = ['apple', 'banana', 'orange', 'grape', 'kiwi'];*

*const firstThreeFruits = fruits.slice(0, 3); // slice the first three fruits from the array  
console.log(firstThreeFruits); // ['apple', 'banana', 'orange']*

*const lastTwoFruits = fruits.slice(-2); // slice the last two fruits from the array  
console.log(lastTwoFruits); // ['grape', 'kiwi']*

*const middleFruits = fruits.slice(1, -1); // slice the fruits between the first and last  
console.log(middleFruits); // ['banana', 'orange', 'grape']*

**10. splice()**

It adds and/or removes array elements. It overwrites the original array thus it is not recommended way.  
Syntax:- **array.splice(index, howmany, item1, ....., itemX);**  
**a. Index**(Required; Indicates the index from where items will be added or removed. Negative value defines the position from the end of the array.)  
**b. howMany**(Optional; Indicates number of items to be removed at the given index)   
**c. item1, ..., itemX** (Optional; New elements(s) to be added)

*const fruits = ['apple', 'banana', 'orange', 'grape', 'kiwi'];*

*const removedFruit = fruits.splice(2, 1); // remove one fruit at index 2  
console.log(removedFruit); // ['orange']  
console.log(fruits); // ['apple', 'banana', 'grape', 'kiwi']*

*fruits.splice(1, 0, 'pear', 'peach'); // add two fruits at index 1  
console.log(fruits); // ['apple', 'pear', 'peach', 'banana', 'grape', 'kiwi']*

*fruits.splice(3, 2, 'mango', 'papaya'); // replace two fruits starting at index 3  
console.log(fruits); // ['apple', 'pear', 'peach', 'mango', 'papaya', 'kiwi']*

**11. every()**

1. executes a function for each array element.
2. returns true if the function returns true for all elements.
3. returns false if the function returns false even for one element.
4. does not execute the function for empty elements.
5. does not change the original array.

Syntax:- array.every(callbackFunction(currentValue, index, arr), thisValue)

*const numbers = [2, 4, 6, 8, 10];*

*const allEven = numbers.every((num) => num % 2 === 0); // Check if all numbers are even  
console.log(allEven); // true*

*const allGreaterThan5 = numbers.every((num) => num > 5); // Check if all numbers are greater than 5  
console.log(allGreaterThan5); // false*

*const words = ['apple', 'banana', 'pear', 'grape', 'kiwi'];*

*const allMoreThan3Chars = words.every((word) => word.length > 3); // Check if all words have more than 3 characters  
console.log(allMoreThan3Chars); // true*

*const allStartsWithA = words.every((word) => word.charAt(0) === 'a'); // Check if all words start with letter 'a'  
console.log(allStartsWithA); // false*

**12. some()**

1. Checks if any array elements pass a test (provided as a callback function).
2. Returns true (and stops) if the function returns true for one of the array elements.
3. Returns false if the function returns false for all of the array elements.
4. Does not execute the function for empty array elements.
5. Does not change the original array.  
   **Syntax->array.some(callbackFunction(value, index, arr), this);**

*const numbers = [2, 4, 6, 8, 10];*

*const anyOdd = numbers.some((num) => num % 2 !== 0); // Check if any numbers are odd  
console.log(anyOdd); // false*

*const anyLessThan5 = numbers.some((num) => num < 5); // Check if any numbers are less than 5  
console.log(anyLessThan5); // false*

*const words = ['apple', 'banana', 'pear', 'grape', 'kiwi'];*

*const moreThan5Chars = words.some((word) => word.length > 5); // Check if any words have more than 5 characters  
console.log(moreThan5Chars); // true*

*const anyStartsWithA = words.some((word) => word.charAt(0) === 'a'); // Check if any words start with the letter 'a'  
console.log(anyStartsWithA); // true*

**13. find()**

1. Returns the value of the first element that passes a test.
2. Executes a function for each array element.
3. Returns undefined if no elements are found.
4. Does not execute the function for empty elements.
5. Does not change the original array.

**Syntax:- array.find(function(currentValue, index, arr),thisValue)**

*const numbers = [2, 4, 6, 8, 10];*

*const foundNumber = numbers.find((num) => num > 6 && num % 2 === 0); // Find the first even number greater than 6  
console.log(foundNumber); // 8*

*const fruits = [   
 { name: 'apple', color: 'red' }, { name: 'banana', color: 'yellow' },  
 { name: 'pear', color: 'green' }, { name: 'grape', color: 'purple' },   
 { name: 'kiwi', color: 'brown' },  
];*

*const foundFruit = fruits.find((fruit) => fruit.color === 'green'); // Find the first fruit that is green  
console.log(foundFruit); // { name: 'pear', color: 'green' }*

*const foundFruit2 = fruits.find((fruit) => fruit.name.charAt(0) === 'b'); // Find the first fruit that starts with the letter 'b'  
console.log(foundFruit2); // { name: 'banana', color: 'yellow' }*

**14. includes()**

The includes() method returns true if an array contains a specified value.  
 Note:- includes method also works for Strings as well.

**Syntax:- array.includes(element, start)**

a. element: (Required; The value to search for)  
b. start: (optional; Start position. Default is 0)

**15. indexOf()**

The indexOf() method returns the first index (position) of a specified value.  
The indexOf() method returns -1 if the value is not found.  
The indexOf() method starts at a specified index and searches from left to right.   
By default the search starts at the first element and ends at the last.   
Negative start values counts from the last element (but still searches from right to left).

Note:- This is also available for string.

**Syntax:- array.indexOf(item, start)**  
a. **item** : (Required; The value to search for)  
b. **start** : (Optional; Where to start the search;Default value is 0;Negative values start the search from the end of the array)

**16. lastIndexOf()**

The lastIndexOf() method returns the last index (position) of a specified value.  
The lastIndexOf() method returns -1 if the value is not found.  
The lastIndexOf() starts at a specified index and searches from right to left.  
By defualt the search starts at the last element and ends at the first.  
Negative start values counts from the last element (but still searches from right to left).

**Syntax:- array.lastIndexOf(item, start)**

a. **item** : (Required. The value to search for)  
b.**start :** ( Optional; Where to start the search; Default is the last element (array.length-1);  
Negative start values counts from the last element (but still searches from right to left).)

**17. find()**

The find() method returns the value of the first element that passes a test.  
The find() method executes a function for each array element.  
The find() method returns undefined if no elements are found.  
The find() method does not execute the function for empty elements.  
The find() method does not change the original array.

Syntax:- **array.find(function(currentValue, index, arr),thisValue)**

**18. findIndex()**

**19. join()**

The join() method returns an array as a string(joins all elements of an array into a string).   
The join() method does not change the original array.  
Any separator can be specified. The default is comma (,).

**Syntax:- array.join(separator)  
Separator** (Optional; Determines the separator to be used; Default is a comma.)

**20. concat()**

The concat() method concatenates (joins) two or more arrays.  
The concat() method returns a new array, containing the joined arrays.  
The concat() method does not change the existing arrays.

syntax:- **array1.concat(array2, array3, ..., arrayX);**

Note: This concat method can be used to concat 2 or more strings.

**21. sort()**

The sort() sorts the elements of an array.  
The sort() overwrites the original array.  
The sort() sorts the elements as strings in alphabetical and ascending order.

How sort functions work in javascript.

Sort function, sorts the data converting each element into string and then sequentially matching the each character using UTF-16 code unit values.

so if we are sorting numbers as well then those particular number array will be converted to string and sequentially it will be matched with its UTF-16 code unit values.

Important Note:-

Sorting alphabetically works well for strings ("Apple" comes before "Banana").  
But, sorting numbers can produce incorrect results.  
"25" is bigger than "100", because "2" is bigger than "1".  
You can fix this by providing a "compare function".

**Syntax:- array.sort(compareFunction);**

**compareFunction**

a. It is Optional.  
b. A function that defines a sort order. The function should return a negative, zero, or positive value, depending on the arguments:

function(a, b){return a-b}

When sort() compares two values, it sends the values to the compare function, and sorts the values according to the returned (negative, zero, positive) value.

example:-

var items = [   
 { name: 'Edward', value: 21 },{ name: 'Sharpe', value: 37 },{ name: 'And', value: 45 },  
 { name: 'The', value: -12 }, { name: 'Magnetic', value: 13 },{ name: 'Zeros', value: 37 } ];

// sort by value

items.sort(function (a, b) { return a.value - b.value });

// sort by name

items.sort(function(a, b) {

var nameA = a.name.toUpperCase(); // ignore upper and lowercase  
 var nameB = b.name.toUpperCase(); // ignore upper and lowercase  
 if (nameA < nameB) {return -1;}  
 if (nameA > nameB) {return 1;} // names must be equal  
 return 0;  
 });

**22. reverse()** ->

The reverse() method reverses the order of the elements in an array.  
The reverse() method overwrites the original array.

Syntax :- array.reverse()

**23. flat()**

The flat() method creates a new array with all sub-array elements concatenated into it recursively up to the specified depth.

**24. fill()**

The fill() method fills specified elements in an array with a value.  
The fill() method overwrites the original array thus it is not preferred.  
Start and end position can be specified. If not, all elements will be filled.

Syntax :- **array.fill(value, start, end)**

a. value (Required; The value to fill in)  
b. start (Optional; The start index (position). Default is 0.)  
c. end ( Optional; The stop index (position); Default is array length.)

25. isArray()

26. from()