**FRONTEND**

**JavaScript**

JavaScript only used to work on the browser(on client side). We needed to learn another language to handle the server side database tasks (ex. php) But node.js changed it. Now because of node.js, we can use JavaScript on server side too.

Browsers used V8 Engine to run JavaScript code. Node.js adopted that V8 Engine on server side. Node.js is runtime software which we installed on server side.

Now we can use JavaScript on both client side and server side.

**JS implementation:**

1. Inline (should avoid. bad practice)
2. Internal
3. External
4. Inline JS : We use event as an attribute of the tag. Some inline attributes events are : onclick, onfocus, ondblclick, onload & onchange.

1. <head>

2. <script>

3. document.write(“Welcome to Javascript World”);

4. </script>-

5. </head>

6. <body onload=”alert(‘Hello’)”>

7. <button onclick=”alert(‘hello world’)”></button>

8. </body>

**JavaScript Type Conversions**

To convert JavaScript variables to another datatype

* By the use of JavaScript functions
* Automatically by JavaScript itself

Some Type Conversion functions are :

1. Number()
2. parseInt()
3. parseFloat()
4. Boolean()
5. String()

Brief : <https://www.w3schools.com/js/js_type_conversion.asp>

#############################

Tip : ‘clg’ for shortcut for ‘console.log()’ in VsCode

#############################

1. var a = Number(promt(‘Enter a Number’));

#############################

1. var a = true; Boolean datatypes (true = 1, false = 0)

2. var b = true; [Plus operator will try to add every possible datatype that can be added.

4. console.log(a+b); Otherwise it will concat them ]

5.

6. output : 2

#############################

1. var a = “5”; [JavaScript try to convert datatype to a reasonable datatype so it can perform operations,

2. var b = true; If the datatype of variable are not conversable and can’t be added together, then it just

3. console.log(a+b); concatenate them ]

4.

5. output : 5true

#############################

1. var a = ‘hello’; 1. var a = 5; 1. var a = “5”;

2. var b = 2; 2. var b = 10; 2. var b = 5;

3. console.log(a-b); 3. var c = a == b; 3. var c = a == b;

4.   4. console.log(c); 4. var d = a === b; // will check datatype too

5. output : NaN [Not a Number] 6. 5. console.log(c);

7. output : false 6. console.log(d);

7.

8. output: true

9. false

#############################

Accessing JavaScript variables in HTML (without using DOM)

<https://github.com/69JINX/FrontEnd/blob/main/Javascript/Marksheet-Table%20-%20Inserting%20script%20between%20HTML%20tags.html>

#############################

Creating HTML tags by JavaScript(without using DOM) :

1. document.write(“<h1>Hello</h1>”);

2. document.write(“Hello” + “</br>”);

#############################

1. let str = “Hello”;

2. document.write(str[2]); // this will print ‘e’

#############################

**Template Literals (String Interpolations)**

Used to print text and variables together without using the ‘+’ operator. ${} is used to print variables. TL are used without BackTicks (not back slashes);

Without using TL :

1. document.write(“Hello” + name + “GM”);

With TL :

1. document.write(`Hello ${name} GM`);

#############################

Code written after return keyword in a function never get executed.

#############################

**Arrow Functions**

Arrow Functions were introduced in ES6. Arrow Functions allow us to write shorter function syntax.

1. const fun1 = (a,b) => a\*b;

It gets shorter if the function has only one line of statement and the statement returns a value, you can remove the brackets and the return keyword.

Note : Arrow functions are variables, just like regular variables, it shouldn’t be called before defining it. Not like regular function that can be define anywhere in the code and be called anywhere.

**Arrow Function : Normal Function :**

1. fun1(); 1. fun1();

2. let fun1 = () => { 2. function fun1(){

3. document.write(“Hello”); 3. document.write(“Hello”);

4. } 4. }

5.   5.

6. output : 6. output : Hello

7. Reference Error : Cannot access ‘fun1’

8. before initialization

#############################

1. let temp1 = {}; //object

2. let temp2 = []; // array

#############################

**We can change key/value of a const object/array**

In a constant object(const), even though it is const, we can change its key/value pairs and add/delete key/value Because we are not changing the whole object when we are altering the properties/methods of an object, we are not re-assigning or re-declaring the contact, it’s already declared and assigned. We’re just adding to the list of elements or properties to which the constant points

**So this works fine: and this works fine too: but these will through error:**

1. const obj = {}; 1. const arr = []; 1. const obj = {};

2. obj.foo = ‘bar’; 2. arrr.push(‘foo’); 2. obj = {foo:’bar’}; // error:reassigning

3. console.log(obj); // {foo:’bar’} 3. console.log(arr); // [‘foo’] 3.

4. obj.foo = ‘bar1’; 4. arr.unshift(‘foo2’); 4. const arr = [];

5. console.log(obj); // {foo:’bar1’} 5. console.log(arr); // [‘foo2’,’foo’] 5. arr = [‘bar’]; // error:reassigning

6. arr.pop();

7. console.log(arr); // [‘foo2’]

So the properties of elements of a constant object/array can be changed but whole object/array can’t

#############################

**Entry Control Loop**

for and while are entry control loop are entry control loop cause they check the condition before entering inside the loop.

**Exit Control Loop**

do-while is a exit control loop cause it run the loop then check the condition means even if condition is false, it will at least run one time.

#############################

3 types of function :

1. Without argument (simple function)
2. With argument
3. Functions the ‘return’ something

#############################

**Mutable v/s Immutable**

There are 2 types of datatype in JavaScript :

1. Mutable / Non-Primitive / Reference Type : Something that can not be changed or added to.

Eg : Object, Arrays, Functions

1. Immutable / Primitive : Something that can be changed or a new property be added to

Eg : Number, Boolean, String, Null, Undefined, Symbol

Immutable, once created can’t change their value but can be reassigned to a new one.

**Immutable: Mutable:**

1. let a = ‘john’; 1. let a = [1,2,3,4]

2. let b = a; 2. let b = a;

3. b = ‘steve’; 3. b[1] = 100;

4. console.log(a); 4. console.log(‘a = ’,a);

5. console.log(b); 5. console.log(‘b = ’,b);

6.   6.

7. output : john 7. output :

8. steve 8. a = [1, 100, 3, 4]

9. b = [1, 100, 3, 4]

**In memory: In memory:**

both variables point at different both variables point at same memory location because we have assigned

memory location of their own memory address of b to point at memory address of a by doing ‘let b = a’

(which is the same address)

a -> 0xf094c649 [john]

b -> 0x7de53064 [steve] a

0x6a99aa2f [1, 100, 3, 4]

b

By default, all Reference types (array, object, function) are mutable in JavaScript like if we normally assign a Reference datatypes of another data type, they both will point at same value at single memory address unlike C/CPP language where we have to use a pointer to do that.

1. int a = 10;

2. int\*b = &a; // pointer in C language

If you don’t want address of a Reference type to be copied to another variable when we assign them but only value to be copied, you can use the spread operator.

1. let str = ‘Hello’;

2. str[1] = a; // <- this won’t work because strings are immutable

3. console.log(str); // Hello

4. consolelog(str[1]); // e

#############################

**Spread Operator**

Spread Operator will copy the value of Reference datatype(or any other) of another data variable without copying the address. They both will point at different memory location.

Syntax : b = [...a]

**Without Spread Operator**

1. let a = [1, 2, 3, 4]

2. let b = a; [ value/s of a = value/s of b] (value/s will be copied)

3. b[1] = 100; [ address of a = address of b] (address will also be copied)

4. console.log(a); [ they will point at same address where a single array is stored ]

5. console.log(b);

6.

7. output:

8. [1, 100, 3, 4]

9. [1, 100, 3, 4]

**With Spread Operator**

1. let a = [1, 2, 3, 4]

2. let b = [...a]; [ value/s of a = value/s of b] (value/s will be copied)

3. b[1] = 100; [ address of a ≠ address of b] (address will not be copied)

4. console.log(a); [ they will have their own separate memory addresses ]

5. console.log(b);

6.

7. output:

8. [1, 2, 3, 4]

9. [1, 100, 3, 4]

#############################

**Checking Mutability with Object**

1. const obj = {

2. name : ‘john’,

3. gender : ‘male’,

4. age : 25

5. }

6. const obj1 = obj;

7. obj.name = ‘steve’;

8. console.log(obj);

9. console.log(obj1);

10. output :

11. {name:‘steve’,gender:‘male’,age:25}

12. {name:‘steve’,gender:‘male’,age:25}

Mutability is commonly associated with non-primitive data types (e.g., arrays, objects) in most programming languages, including C, C++, Java, and JavaScript. However, JavaScript explicitly labels these as "mutable data types." While the concept is similar across languages, non-primitive types aren't always mutable, as some languages or implementations offer immutable versions.

#############################

**Comparing Mutable Datatypes**

Why comparing Mutable datatypes with same values give false ?

Lets look at an example:

1. let a = ‘john’; // immutable datatype

2. let b = ‘john; // immutable datatype

3.

4. let x = {}; // mutable datatype

5. let y = {}; // mutable datatype

6.

7. let obj = { // mutable datatype

8. name : ‘steve’,

9. gender : ‘male’,

10. age: 25

11. };

12. let obj1 = { // mutable datatype

13. name : ‘steve’,

14. gender : ‘male’,

15. age: 25

16. };

17.

18. let arr = [1, 2, 3]; // mutable datatype

19. let arr1 = [1, 2, 3]; // mutable datatype

20.

21.

22. console.log(a == b);

23. console.log(obj == obj1);

24. console.log(x == y);

25. console.log(arr == arr1);

26.

27. output:

28. true

29. false

30. false

31. false

32.

When we compare two mutable datatypes, it returns false because JavaScript deals with similar object/array properties as a different one. That is in JavaScript, two object/array are considered equal only if they are the same object/array, not if they have the same properties and values, and in JavaScript object/array are assigned and compared by the reference not by the value.

In the above program, when we compared object obj with object obj1, we are comparing their reference address not their value. Because they have different address reference, the comparison will be false.

So can’t we get true ?

When we have the same properties and value, you we can!

In order to get true, the object must point at same memory address reference.

1. let obj = {name:‘steve’, age:22};

2. let refobj = obj; // create a reference to the object

3. console.log(obj == refobj);

4. console.log(obj === refobj);

5.

6. output : true

7. true

#############################

**Destructuring in JavaScript**

Destructuring assignment is a JavaScript expression that allows you to extract data from array, objects and maps and set them into new, distinct variables. Destructuring allows us to extract multiple properties or items from an array at a time.

Here are some examples of Destructuring:

**Basic Object Destructuring: Nested Object Destructuring:**

1. const user = { 1. const user = {

2. name : ‘Alex’, 2. name : ‘Alex’,

3. gender: ‘Male’, 3. address :{

4. age:25 4. country:‘USA’,

5. } 5. city:‘New York’

6. const { name } = user; // variable ‘name’ should 6. }

7. console.log(name); // be same as key from the 7. }

8.   // object 8. const { address:{city} } = user;

9. output : Alex 9. console.log(city);

10.

11. output : New York

**Array Destructuring: Default Values:**

1. const numbers = [1, 2, 3, 4, 5] 1. const user = {

2. const [first, second, third] = numbers; 2. name : ‘Alex’,

3. console.log(first, second, third); 3. age:43

4.   4. }

5. output : 1 2 3 5. const {name , age = 25} = user;

6. console.log(name, age);

7.

8. output : Alex 43

**Rest Parameter:**

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

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

3. console.log(first, reset);

4.

5. output : 1 2 [3, 4, 5]

#############################

**Adding and Deleting key from Object :**

**Deleting Key from Object: Adding Key from Object:**

1. const obj = { 1. const obj = {

2. name: ‘Alex’, 2. name: ‘Alex’,

3. age : 25 3. age : 25

4. } 4. }

5. delete obj.name; 5. obj.gender = ‘male’;

6. console.log(obj); 6. console.log(obj);

7.   7.

8. output : {age:25} 8. output : {name:‘Alex’, age:25, gender:‘male’}

#############################

**Why typeof array is object :**

In JavaScript, arrays are objects because JavaScript is a prototype-based language This means that there are only primitive types and objects. Arrays are a special case of objects inside the JavaScript engine they have :

* Special handling of indices : Array indices are represented as strings, that contain numbers.
* A length property : The length property of an array indicates the number of elements in the array.
* Methods : Array have a number of methods such as push(), pop(), shift() and unshift() that can be used to add, remove, and manipulate elements in the array.

The type of operator in JavaScript return “object” for arrays. This is because arrays are object in JavaScript even though they have some special properties and methods.

1. var arr = [1, 2, 3];

2. console.log(typeof arr);

3.

4. output : Object

If you need to check if a variable is an Array, you can use the Array.isArray() method. This method returns true if the variable is an array otherwise false.

1. var arr = [1, 2, 3];

2. console.log(Array.isArray(arr));

3.

4. output : true

To check the size of array, we use array.length

1. let arr = [1, 2, 3]

2. console.log(arr.length); // 13

Its is an property of array object, not a method. That’s why we didn’t use parenthesis at the end of arr.length

#############################

**Browser Object Model / BOM / Window Object**

The BOM is used to interact with the browser. The default object of browser is ‘window’ means you can call all the functions of window by specifying window or directly. For example

1. window.alert(‘hello’); // all three are same

2. alert(‘hello’);

3. this.alert(‘hello’); // this represents the parent object, in this case the top upper object of alert is window

You can use a lot of properties (other objects) defined underneath the window object like document, history, screen, navigate, location, innerHeight, innerWidth etc.

If we print ‘this’, the window object will be printed.

1. console.log(this);

2. console.log(window); // both will print window object.

**this inside an object this inside a arrow function**

**If we try to print ‘this’ inside an but if we use an arrow function inside an object and try to print ‘this’**

**object made by programmer, then we might think that it will print the object itself but when using**

**then this will print the object ‘this’ inside an arrow function. It always print the root parent object**

**which is its parent object which is the window object itself.**

1. const obj = { 1. const obj = {

2. fun : function(){ 2. fun : () => {

3. console.log(this); 3. console.log(this);

4. } 4. }

5. } 5. }

6. obj.fun(); 6. obj.fun();

7.   7.

8. output : {fun:f} 8. output : window {window:Window,document...}

1. const obj = {

2. fun : () =>{

3. this.console.log(this); // will run because this inside an arrow function represents the window object.

4. // which in this cause is correct

5. },

6. fun1 : function(){

7. this.console.log(this); // will throw error because this inside a normal function represents the parent object

8. // which is obj and obj doesn’t have any console name of function

9. }

10. }

11.

**Document Object Model / DOM / Document Object**

DOM creates a tree like structure of whole HTML page.

Every node have only three type relationship between other nodes :

1. Parent
2. Child
3. Sibling

In DOM, every element node is an objects

Whenever a new node is added to the tree, whole DOM tree is destroyed and recreated again, this slows down the webpage. That’s why we use React which provide Virtual DOM.

Dynamic changes using DOM :

1. Text change
2. Attribute
3. CSS properties
4. HTML Structure

DOM Selectors :

1. **Class** : document.getElementByClassName(‘class’);
2. **Id** : document.getElementById(‘id’); return HTML Collection
3. **TagName** : document.getElementByTagName(‘tag’);
4. document.querySelector(‘.class’/ ‘#id’ / ‘tag’); ---------------> return single HTML element that comes first in tree, if there are more, the ignores rest all
5. document.querySelectorAll(‘.class’/ ‘#id’ / ‘tag’); -----> return NodeList

**HTML Collection v/s NodeList**

|  |  |
| --- | --- |
| **HTML Collection** | **NodeList** |
| It only contains tags | It can contain any node like text, attribute, comment, new line etc. |
| Cannot iterate over its elements using forEach() | It is possible to iterate over it with forEach() |
| HTML Collection is the HTML DOM is live. It is automatically updated when the underlying document is changed. | It is live & static both in different conditions querySelectorAll() returns a static NodeList |

**Difference in BOM and DOM**

DOM : The Document Object Model (DOM) and the Browser Object Model (BOM) are two important concepts of JavaScript.

The DOM is programming interface for HTML and XML documents. It represents the page so that programs can change the document structure, style and context. The DOM provides a standard object model for accessing, manipulating and navigating HTML and XML documents.

BOM : The BOM is collection of objects that represents the browser window and its contents. It allows JavaScript to interact with browser, such as opening and closing windows, navigating history and manipulating cookies.

Change text written between tag :

1. element.innerText = ‘new\_Text’;

Attribute Selector :

To select the attribute, we can directly write the name of the Attribute.

1. imgElement.src = “img1.jpg”;

**Using ‘this’ keyword in EventListener**

Arrow functions always give window object when using ‘this’ but a normal function give that specific object (parent object) when printing ‘this’.

**X Wrong**

1. img1.addEventListener(‘click’, () => {

2. console.log(this); // this will print window object

3. this.src = ‘img2.jpeg’; // this won’t work, will give error

4. }

**🗸 Right**

1. img1.addEventListener(‘click’, function() {

2. console.log(this); // this will print image tag

3. this.src = ‘img2.jpeg’; // this will work

4. }

**Key events** :

* keypress ● keyup ● keydown

**Changing CSS with JavaScript of the element** :

Syntax : elementobj.style.css-property = “css-propery-value”

eg:

1. img1.style.src = “img2.jpeg”;

2. div1.style.backgroundColor = “blue”;

Note : when specifying css properties in JS, if there is a hyphen in the name, then it should be removed and make the first letter capital after hyphen.

CSS JavaScript

border-radius borderRadius

**getComputedStyle()**

We can set CSS property of any elementobj with the above syntax but we can’t access or get css property of any elementobj with this method. That’s why we need to use the getComputedStyle() method where we pass the element as argument.

getComputedStyle() method gets the computed CSS properties and values of an HTML element in an object form (in key:value pairs).

getComputedStyle() only take one element as argument. It only work when there is only one element, it doesn’t work the ‘the DOM selector’ that give an html collection or nodelist.

eg:

1. getComputedStyle(document.getElementByClassName(‘para’)); // error : HTML collection

2. getComputedStyle(document.getElementByClassName(‘para’)[0]); // run successfully without any error

3. getComputedStyle(document.querySelector(‘p’)); // run successfully without any error

1. const element = document.getElementById(‘test’);

2. const obj = getComputedStyle(element);

3. let bgColor = obj.getPropertyValue(‘background-color’);

4. console.log(bgColor);

5.

6. output : rgb(173, 216, 230)

**EventObject** (4-july-24)

If we have to make multiple event listeners for multiple element, it would increase code lines and make it lengthy. That’s why e.target is used which is a property of event object ‘e’. e.target gets the whole element where user slick or hover or performed any action.

1. <div id=“frame”>

2. <img src=“imgmain.jpeg”/>

3. </div>

4. <div class=“boxes”>

5. <img src=“img1.jpeg”/>

6. <img src=“img2.jpeg”/>

7. <img src=“img3.jpeg”/>

8. </div>

**Without EventObject (e.target) With EventObject (e.target)**

1. img1.addEventListener(‘click’, ()=> { 1. boxes.addEventListener(‘click’, (e)=> {

2. mainimg.src=img1.src; 2. if(e.target.tagName == ‘IMG’)

3. }); 3. {

4. Img2.addEventListener(‘click’, ()=> { 4. mainimg.src = e.target.src;

5. mainimg.src=img1.src; 5. }

6. }); 6. }

7. Img3.addEventListener(‘click’, ()=> {

8. mainimg.src=img1.src;

9. });

We didn’t had to make event listeners for every element. Just added an event listener on parent div and set e.target If target click give IMG tag then pass that img src to mainimg.

#############################

innerText = only give text that is inside the tag

innerHTML = give entire html element

#############################

JS program to show html element on window screen scroll :

1. window.addEventListener(“scroll”, () => {

2. if(window.scrollY >= 100){

3. para.style.opacity = 1;

4. }

5. });

#############################

when targeting section with anchor tag to jump on a section the page jump directly without any smoothness in scroll so to scroll smooth, apply this css property to html tag :

**CSS**

1. html{

2. scroll-behaviour:smooth;

3. }

#############################

**Date Object**

Date() is already inbuild class in JavaScript. We created a date object to access methods/properties of Date() object

1. let date = new Date(); // date constructor

2. date.getMonth();

3. date.getHours(); // there are many functions of date, see on web for more

4. // above getHours method give 24 format, to get in 12 format :-

5. let finalHours = (hours>12)?hours-12:hours;

#############################

**Delete Parent node/element** :

1. element.parentNode.remove();

**to get sibling node** :

1. element.nextElementSibling;

**toggel class** :

1. element.classList.toggle(‘class\_name’);

#############################

**Get height of window/viewport:**

1. window.innerHeight;

**get position of any html element according to screen :**

1. element.getBoundingClientRect();

Don’t use window.scrollY when doing some changes on scroll in any element because screen size might be different in other devices and this won’t work according to planned, that’s why window.innerHeight and getBoundingRect() is used together.s

#############################

**Higher Order Functions**

In JavaScript, a higher order function is a function that can take another function as argument or return functions are their result.

Some Higher Order Functions are :

* map() ● filter() ● reduce() ● forEach() ● sort() ● reduce ● setTimeout()

#############################

floor() ---> 2

2.7

ceil() ----> 3

#############################

**Difference between method and function**

>> A function is a block of code written to perform some specific set of tasks. We can define a function using the function keyword by name and optional parameters.

>> A JavaScript Method is a property of an object that contains a function definition. Methods are functions stored as object properties. Object methods can made by following syntax.

1. const object = {

2. methodName : function(){

3. // method content

4. }

5. }

6. object.methodName(); // accessing method of object, just like arr.push() or any other predefined method

#############################

**Anonymous Function**

It is a function that does not have any name associated with it. Normally we use the function keyword before the function name to define a function in JavaScript, however in anonymous function of JavaScript we use only function keyword without the function name.

An Anonymous function is not accessible after its initial creation, it can only be accessed by a variable it is stored in as a function as a value.

We can also declare an Anonymous function using the arrow function.

**Anonymous Function: Anonymous Arrow Function:**

**syntax**: **syntax**:

1. function() { 1. () => {

2. // function body 2. // function body

3. } 3. }

**Eg:**

1. let fun1 = function(a){

2. console.log(a);

3. }

4. fun1(a);

**Anonymous function as IIFE:**

1. (function(){

2. console.log(‘Hello’);

3. })();

Anonymous functions are mostly used in Higher order function or as a IIFE(Immediately Invoked Function Express)

#############################

**JavaScript Program to remove duplicate values from array (using array’s includes() method)**

1. const arr = [2, 8, 7, 6, 2, 4, 8, 7, 5, 6, 1, 2, 6]

2. const arr1 = [];

3. arr.forEach(

4. (v) => {

5. If(!arr1.includes(v)){

6. arr1.push(v);

7. }

8. }

9. );

10. console.log(“Array After Duplicate Values removed : ” + arr1);

Practice questions :

* program to find max
* program to find min
* program to find 2nd max
* program to find 2nd min

#############################

**Asynchronous nature of JavaScript**

Nature of JavaScript is Asynchronous like that it will not stop other next lines of code if one line of code is taking too much time, it will run all codes that can run and other codes that take time will print in the end. It is useful but not always, imagine we are fetching some data from somewhere else and when that data get fetch, we will print the data. Due to asynchronous nature of JS, it won’t wait for the data to fetch and execute next line where we are printing the fetch, Now that might cause problems because we haven’t wait for the data and directly printed it, this will cause garbage data/undefined to appear to user.

That’s why Synchronous nature of JS was introduced.

To run the written code is synchronous way, we use Promises.

**Asynchronous way:**

1. let data;

2. setTimeout(()=>{ // we have created a setTimeout function to mimic how a fetched data/connection would take time

3. data = ‘some data’; // to fetch data

4. },5000);

5. console.log(data); // now here the data will be printed ‘undefined’ because it will take 5 second to fetch data

Synchronous Way :

As the name suggest, synchronous means to be in a sequence. i.e. every statement of the code gets executed one by one. Here tasks are performed sequentially with the help of a call stack. Each individual task must complete before the next one can begin. This results in delays in execution if any task takes a significant amount of time to execute.

Promise object is used.

**Promise**

Promise has 3 States :

1. Pending
2. Fulfill
3. Reject

Promises are :

1. Created and
2. Handled

Mostly, we handle already created promises in APIs when we fetch an API, it return a promise with one of the state from 3 states.

We can’t directly access an promises, we need to handle it by two methods :

1. Then and Catch method
2. Async & Await

Back in the days, there wasn’t any technology in JS for synchronous data flow, we had to rely on other 3rd party applications like ‘Q and BlueBird’

**Syntax:**

1. const promise\_name = new Promise((resolve, reject)=>{

2. resolve({object}/[array]/value); // sending data with resolved promise

3. reject(throw “error”); // in case need to reject promise

4. });