**JavaScript**

**General Points**

1. JavaScript on Server helps in Powering websites, Communicate with databases and native feel to web apps.
2. JS is used in IOT for programming devices and in React Native for mobile app development.
3. Initially jQuery(JavaScript Library) is used to make the JS compatible for Different Browsers, we just needed to add it and use it feature which work on all browser like same(the solution was not efficient BTW).
4. Old Code == Legacy Code 🡪 same thing
5. Unlimited arguments in JS is done with Normal and Arrow function.
   1. In normal function we don’t have to take any arguments e.g.: function fun-name (){ console.log(arguments)}.
   2. In Arrow Function we mention arguments in spread operator type and then we can use inside as an array e.g.: const func = (…args) => {console.log(args)}
6. Anonymous Function = const func\_Name = function(){console.log()}
7. Self-Invoking Function = let func\_Name = (function(N)=>{console.log(N)})(10 -> Value of N)
8. isNaN = it will give true to all value who are not in digit including Undefined. Undefined will result true. But null will be false. Type of Null = “object”.
9. to Check type use “” when mentioning the Data Type like: type of “HELLO” == ‘string’ => true.
10. isFinite => will give true to all number only. Null will also be true. Undefined is false .
11. When Variables are not declared with let and const it becomes a global variable. Same with “var” (declared variable becomes global), let and const introduced the concept of block scope in es7 after 2015.
12. Object.assign({},OBJ\_NAME): Will copy properties of on object to another.
13. Object.keys(OBJ\_NAME), Object.values(OBJ\_NAME).
14. Object.entries(OBJ\_NAME) = to give key and value in an array.
15. OBJ\_NAME.hasOwnProperty(“key”): To check if a specific key or property exist in an object.
16. For-Of is used for Arrays and For-In is used for Objects.
17. In a function, we can pass the callback function as a parameter. But if we don’t pass it, the. Function which is using the callback function, works fine. As we have to declare the callback function prior to the normal callback function (which the function which is using the Callback function).
    1. const callback = function(element) { sum += element } arr.forEach(callback). FOR EACH function does not change the original array or make new array. I guess mostly used to make new results based on that array.
18. when we create an object form class that process is called Class Instantiation
19. A set can have property. Of inserting an arbitrary element and deleting an arbitrary element. Because when you. Print a set, there is no pattern of printing they will. Show up in output randomly We can use add function to add elements in set, delete to delete insets and has to check whether the element exists in a set or not.

Ex. *Set(6) {1, 2, 3, 4, 5,6}*

1. A map is more like an object. It is not clearly work on key value pair objective. When we provide input it take? One in the left, which is like symbol of key and one in the right, which is symbol of value and we can input it with the help of set function.

Example: *Map(3) {"Finland" => "Helsinki", "Sweden" => "Stockholm", "Norway" => "Oslo"}*

1. With the help of object entities, we can. Use objects and print their value and key collectively. With the help of for in loop but in case of map we can do with for off loop without using object class.

For example: *for (const [country, city] of countriesMap){*

*console.log(country, city)*

*}*

1. When we want to declare a function inside a class, the use of function keyword as prefix is not needed.
2. Static functions do not have access to instance variables because they don't have an instance of the class to access them from.
3. In classes if you want to use a value inside a function but the value is coming from the object you created for that very class. It’s not going to work because the value which is passed in class as an object is only for the instances of that class not for the class itself, in order to use a value during class declare as static property.
4. instance level variable cannot accessible in static function.
5. Key difference between JSON and JS object is in JSON the key is under “\_” and in object it is not.
6. Closures: they are used to make a variable private by making the variable inside a nested function. Closures can be declared in return statement as self-invoking function.
7. High-level functions, also known as higher-order functions, are functions that can accept other functions as arguments and/or return functions as results.
8. *Passing a function as an argument* is not strictly necessary, but it is often done to make the function more flexible and reusable. By passing a function as an argument, you allow the caller of the function to specify custom behavior without modifying the function itself. This promotes code reusability and modularity, as the same function can be used with different callback functions to achieve different results. Stops us to declare same function again for just little difference.
9. Tip: do not get fixated by the way HTML tag is printing in your console. It varies a lot.
10. Interesting Fact: When we get data form JSON file or online API *sort() will not be applied to it*, u have to bypass the inbuilt function and return (1,0,-1) based on ascending or descending order.

Example: jsonData.countries.country.sort((a, b) => {

// Convert countryName to lowercase for case-insensitive sorting

const countryNameA = a.countryName.toLowerCase();

const countryNameB = b.countryName.toLowerCase();

// Compare the country names

if (countryNameA < countryNameB) {

return -1; // Country A comes before Country B

}

if (countryNameA > countryNameB) {

return 1; // Country A comes after Country B

}

return 0; // Country names are equal

1. window.getComputedStyle(document.querySelector(‘.<class-name>’), null).getPropertyValue(<css-property-name>) => get the property used in css of the class defined in HTML tag.
2. To remove a class from a HTML tag we should use classList.remove(‘<class-name>’) not removeAttribute(‘class’) because it will remove the whole class attribute instead of a lone class.
3. document.getElementById("my-element").getBoundingClientRect() => retrieve the size and position of a specific element relative to the viewport. It returns a DOMRect(top, left, bottom, right,width and height) object containing valuable information about the element's location and dimensions*. It can be helpful for detecting collisions between elements on the page, which can be useful for creating interactive elements or games.*
4. Document.<---->. style.animationDuration = changes the animation speed which was declared in CSS attribute named *animation*.
5. const Music = new Audio("./Sounds/music.mp3") => way to import sound in JS.
6. Using *for of* for iteration of OBJECT makes the iteration go through its prototype as well as object[<properties>], but if we use *for in* with Object.keys(<object-name>) it will undergo iteration only for the object[<property>] not the prototype the object was created with or prototype of the object in general.
7. Const newObj = Object.create(<prototype-object>) => way to create a new object with old prototype properties too.
8. Const str = “Hello” is a string literal (primitive JS DataType).
9. Const str2 = new String(“World”) is a String Object. And every string object have a unique signature so new String(“World”) === str2 is FALSE as well as “World” === str2.
10. Constructor functions are used to create objects with specific properties and behaviors.
    1. Example:

function Icecream(flavor) {

    this.flavor = flavor;

    this.meltIt = function() {

        console.log(`The ${this.flavor} icecream has melted`);

    }

}

let kiwiIcecream = new Icecream("kiwi");

let appleIcecream = new Icecream("apple");

kiwiIcecream; // --> Icecream {flavor: 'kiwi', meltIt: ƒ}

appleIcecream; // --> Icecream {flavor: 'apple', meltIt: ƒ}

The Icecream constructor function serves as a blueprint to *create ice cream objects* with flavor properties and a meltIt method.

1. inheritance in JavaScript is based around the prototype object.
2. Node.js is a standalone language. npm is node package manager. Frameworks and libraries are used as node.js modules.
3. JavaScript treats numbers with leading zeros as octal (base 8) literals if they are parsed in strict mode.
4. After using flexGrow: {1} the next component is throws at right hand side.
5. Event.target 🡺 will refer to the button that was clicked.
6. Event.currentTarget 🡺 refer to the button that contains the event handler(if it’s make sense).
7. When we do not pass a parameter in filter method all the values of the array will not saved into the new array as when there is no condition to check the filter will mark the condition as true.
8. What happens here is that myFunction(param) is immediately invoked when the component renders. This means the function executes right away, not when the button is clicked. If this function triggers a state update or re-render, it can cause an infinite loop because every re-render will keep calling the function again.
   1. <button onClick={myFunction(param)}>Click me</button>
9. What to do to avoid infinite loop: To avoid this issue, you need to pass a reference to the function instead of invoking it immediately. This is done by using an arrow function or a wrapper function.
   1. <button onClick={() => myFunction(param)}>Click me</button>
   2. Only invoked when button is clicked
10. Arrow Functions and Closures: When you use const handleClick = (param) => (event) => {}, you're creating a closure. The outer function (param) returns an inner function (event) that can access both the param and event variables when the event is triggered.
    1. const handleClick = (param) => (event) => {

console.log(param, event); };

<button onClick={handleClick(param)}>Click me</button>

1. **Nullish coalescing operator (**??**)** 🡺 This operator returns the right-hand side ('Loading...') if the left-hand side (bio) is null or undefined. Unlike the logical OR operator (||), which returns the right-hand side if the left-hand side is falsy (like null, undefined, false, 0, NaN, or an empty string ""), the nullish coalescing operator only checks for null or undefined.
   1. <p><i>{bio ?? 'Loading...'}</i></p>
2. Higher order function like const func = () => () =>{return } returns a function

*const add = (x) => x + 1;*

*const multiply = (x) => x \* 2;*

*const compose = (f, g) => (x) => f(g(x));*

*const addThenMultiply = compose(multiply, add);*

*console.log(addThenMultiply(5)); // Outputs: 12*

1. In JavaScript, a function () {} or () => {} always creates a different function, similar to how the {} object literal always creates a new object.
2. Function is JS are first class object. And like any object u can add properties to it.
   1. E.g., suppose sum is a function to add 2 numbers. We can do sum.FullName = “summation”.
   2. Console.log(sum.FullName) // answer: summation
3. **Question**: When and Why I some cases to store key and values we should use Map ()?
   1. **Answer**: Why use Map instead of a basic object?
      1. Map benefits:
         1. Keys in a Map can be any data type, whereas object keys are always strings or symbols.
         2. Map preserves the order of entries, while objects do not guarantee order consistency.
         3. Map has better performance for frequent additions and lookups when compared to plain objects for large datasets.

* *Can we use an object?* Yes, but when arguments (like numbers or arrays) are converted to strings as object keys (via JSON.stringify), it might cause collisions or unintended results. Map handles these scenarios better.

1. CODE notes
   1. const memoizedSum = memorize(sumFn); // here sumFn is a function that adds 2 number
   2. When memoizedSum(2,2) is called, a single closure is called.
   3. Now as long memoizedSum exist memoized(fn) will remain in memory and unless we clear it or reassign memoizedSum(2,2).
   4. Suppose we want to monitor memoizedSum(2,2) calling and store it in a cache. We can return a function from memorize(sumFn).
   5. Now the hierarchy looks like memorize(sumFn) – return --> memoFn(). And now memoizedSum(2,2) is based on behaviour of memoFn() not memoized(sumFn).
   6. So, as we know 1st line creates the single closure means not changes is done related to memoize() but memoFn().
   7. Declare cache mechanism in memoized() and monitor memoizedSum() calls.
2. We can clear cache record of those memoized functions by following methods
   1. Manual clearing with clearCache or re-initialization.
   2. Automatic cache management by limiting size or using WeakMap. This allows garbage collection to clean up memory when the object references are no longer used.
   3. Selective deletion of specific entries using a deleteCache method.
3. Async function will inherently return a promise.
4. ! operator in Typescript
   1. CODE:
      1. const buttonDiv = document.querySelector(“button”)!
   2. Answer:
      1. ! is a TypeScript feature called the non-null assertion operator. It is not standard JavaScript.
      2. The ! tells TypeScript that you are certain the value will not be null or undefined.
      3. It effectively overrides TypeScript's strict null-checking and asserts to the compiler that document.querySelector("button") will always return a valid value.
5. sort() sorts elements as strings by default, not as numbers. For instance: [7, 4, 4, 3, 5] → ['3', '4', '4', '5', '7']. This is because .sort() converts numbers into strings and compares them lexicographically (e.g., "12" comes before "3").
   1. .sort((a, b) => a - b) tells the sort function to compare numbers directly instead of treating them as strings. It ensures ascending numerical sorting.
6. clearTimeout (activeTimeoutId) = this will cancel the running process of active setTimeout/setInterval process.
7. “abcd”.split(“”) == equals == **[…str]** 🡪 they both creates [ 'a', 'b', 'c', 'd' ]
8. Object.entries(object) converts object into array of key value pair e.g., [ [ 'a', 4 ], [ 'b', 4 ]]
9. To convert array of key value pair into object use Object.fromEntries(KvParray) e.g., {a: 4, b: 1}
10. Logic to count the number of occurrence in a string 'abascssdsbabab':
    1. CODE:

for (const char of string) {

// Increment count for the character or initialize it to 1

result[char] = (result[char] || 0) + 1;   
}

* 1. Explanation:
     1. If char exists in result, increment its count.
     2. Otherwise, initialize it to 1.
     3. Base Condition: When nothing exist in the result object, ***(result[char] || 0)*** this checks that is there any key of named result[char] ( in this case : a,b,c,d,e,…..) if no (as we know in first iteration result is empty). We have (undefined || 0).
     4. JS will **choose 0** over undefined as || makes JS choose the first truthy value in the given options.
     5. Every time we have similar key come in the queue we will have (result[char] || 0) checking, which will **now choose the existing key** and check for Key[Value] and when found **add +1** to the old value resulting in increasing the count of occurrence and maintaining the occurrence count logic.
     6. And, Yes the result object will keep replacing the old data with new data (is not an update result[char] = count+1 will replace the old Key[Value] by using its data with new Key[Value] with updated count).
     7. This logic uses the benefits of the fact that an object will always have unique keys.

1. Calling reject/resolve before the completion of Promise will lead to **hard Pushed reject/resolve.**
   1. Example Snippet

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

reject(“Time up”)

}).to((res)=> resolve(res)).catch((res)=>reject(res))

* 1. This will lead to hard reject no matter what the full filled promise conclusion is.

1. JavaScript **arrays are sparse**, meaning they can have "gaps" between defined indexes.
   1. Example: [1, 2, 3, 4, <96 empty items>, 500]
   2. Empty slots act like undefined when accessed but are different from explicitly set undefined values.
   3. Instead of filling the gap with any value (like null or 0), JavaScript just marks these indices as **empty slots**.
   4. These empty slots are not enumerable, which means they won't show up in array iteration methods like forEach or map.
2. parseInt()
   1. **parseInt()** attempts to convert a string to an integer by **parsing it from left to right**.
   2. It stops parsing as soon as it encounters a character that *not valid for a base-10 integer*.
   3. If the first character itself is invalid, it returns NaN.
   4. *Leading whitespace* in the string is *ignored.*
   5. Sample:
      1. console.log(parseInt("7FF", 16));
      2. Output: 2047 (hexadecimal to decimal)
3. Console.log([1,2,3,4,5].push(5)) // Output: 5 🡺 Reason: push() return the new length of array.
4. “#”.repeat(3) // Output: ###
5. valueA <== valueB is incorrect syntax. <== is not a valid operator. JavaScript does not recognize it as a combination of valid operators.
6. A **for loop** in JavaScript consists of three parts: initialization, condition, and increment. These **three parts are optional**, but here's how they behave in these specific cases:
   1. for (false; false; false)
   2. for ("string"; "string"; "string")
7. How to **traverse to an array of promises** with functions inside and resolve in parallel.
   1. Promise.all(Arr => Arr.map(fn => fn()))🡺 returns a new promise.
   2. Some Infi about Promise.all().then()
      1. RESOLVES with an array of results.
      2. REJECTS with the first error encountered.
      3. The resolved array maintains the same order as the input promises
      4. If ANY promise rejects, Promise.all rejects immediately.
      5. Non-promise values are automatically wrapped in resolved promises
         1. Promise.all([1, Promise.resolve(2), 3]).then(results => console.log(results));
         2. Output = [1, 2, 3]
8. Modifying an array while iterating over it can lead to unpredictable behavior. Use while loops, array copies, or other iteration methods to avoid issues.
9. Using prototypes is useful when you want to add shared behavior across all instances of a specific object type (e.g., all arrays).
10. To access the **object that is passed** with the prototype function can be **access with ‘this’** keyword.
    1. Const Arr = [1,2,3] 🡺 Array.prototype.hey = () => { return **this.length**+”hey”} 🡺 Arr.hey()
    2. Output = 3hey
11. ArrValue.slice() will make a copy if ArrValue assignable to another variable.
12. When you use the + operator (or other arithmetic operations) on objects, JavaScript attempts to convert the objects into primitive values.
    1. JavaScript first looks for the valueOf() method on the object.
    2. If valueOf() returns a primitive (like a number), that value is used for the operation.
    3. if valueOf() does not exist or does not return a primitive, it falls back to toString() to get a primitive string value.
13. When converting an object to a string, JavaScript looks for the toString() method on the object.
    1. If toString() is defined and returns a string, that value is used as the string representation of the object.
    2. If toString() does not exist, JavaScript may fall back to valueOf().
    3. The toString() method is specifically designed to return a readable string representation of an object. It’s used in contexts like String(obj) or when the object is interpolated into a template literal (e.g., `${obj}`).
14. The Object.assign() method is used to copy the values of all enumerable own properties from one or more source objects to a target object. It returns the target object.
    1. Object.assign(target, ...sources)
    2. const mergedObj = Object.assign({}, obj1, obj2, obj3); 🡺 this creates a new object if you place a {} in start.
    3. If a source object has properties that are objects themselves, only the reference is copied, not the actual object (i.e., shallow copy).
       1. const obj = { a: 1, nested: { b: 2 } };
       2. const shallowCopy = Object.assign({}, obj);
       3. shallowCopy.nested.b = 42;
       4. console.log(obj.nested.b); // 42
15. The Object.values() method returns an array of a given object's own enumerable property values, in the same order as they appear in the object.
    * 1. Object.values(obj)
      2. const obj = { a: 1, b: 2, c: 3 };
      3. const values = Object.values(obj);
      4. console.log(values); // [1, 2, 3]
16. There is no inbuilt method to sort an object.
17. Object.fromEntries(AARRRRAAYY) 🡺 convert the array with [key,value] to {key: value} object.
18. Remove falsy value from an array use using Boolean() inside filter():
    1. [1, 2, 0, null, "", 3].filter((value) => Boolean(value));
19. **Object.is(value1, value2)**: Compares two values for strict equality.
    1. console.log(Object.is(0, -0)); // false
20. **Object.freeze(obj)**: Makes an object immutable (prevents adding, removing, or modifying properties).
    1. const obj = { a: 1 }; Object.freeze(obj); obj.a = 42; // No effect console.log(obj); // { a: 1 }
21. **Object.seal(obj)**: Prevents adding or removing properties but allows modifying existing properties.
22. **obj.hasOwnProperty(property)**: Checks if a property is a direct property of the object (not inherited).
23. 0000000000000000

:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

**Q&A**

1. What is Global in JS or in Programming Language?
2. What is Window Object in JavaScript?
3. Advantages of callback function if we don’t pass the callback function as a parameter, what will happen?
4. Actual meaning and usage of higher order function in JavaScript.