1) Introduction

1) what is JS?

JS is a programing language which is used to develop dynamic web and mobile apps.

By using JS we can manipulate (add, update, delete) DOM.

By using JS we can perform validations.

2) What is ES?

Ecma Script is a specification for JS.

3) What is nodejs?

Node js is a run time environment for JS. By using nodejs we can create APIs.

Node js is not a programing lanaguage. Node js is not a library.

Node js is not a framework.

2) Setup Development environment?

2) Front-end

- We need a Browser (Chrome).

- We need an IDE to write code (vs code).

- We need a HTML file to run js code in front-end.

- Every browser has js engine in it to run js code.

3) Back-end

- We need to install Nodejs to run js code in back-end.

- We need an IDE to write code.

- Nodejs is a runtime environment for js.

- By using nodejs js we can create APIs.

- How to run js file in nodejs

=> node filename

3) Basics

=> When to store data?

We need to store data in memory, when there is need in app.

=> How to store data in memory?

By using var, let, const, we can store data in memory.

We can perform some operations on the data which is stored in memory

=> What to store in memory?

We have to store values (data) by using data types. These are the real values to do any functionality.

1) variables (we need variables to manipulate/perform some operation/update/delete/save)

We will declare variables by using var, let, const keywords.

2) data types

Primitive data types: we use primitives data types to store single value in memory.

Note: All primitive data types will store value directly in memory.

1) string: String is collection of characters to be stored in memory.

By using '' "" `` we can store string in memory.

2) number: We can store numbers in memory by using number data type.

whether it is integer or float number.

3) boolean:

We can store true or false in memory by using boolean datatype.

4) undefined:undefined means a variable has been declared but it's value has not been assigned.

5) null: Null means an empty value. The variable which has been assigned as null contains no value.

Reference data types: we use reference data types to store multiple values in memory.

Note: All reference data types will store value somewhere in memory location. The stored memory location address will be stored in main memory.

1) Object:

By using object we can store multiple values in single memory location in the form of

key & value pair. By using dot operator we can access the object values in app.

Objects are often used to model real-world entities such as a person, car, or any other entity that has properties and behaviors.

2) Array:

By using array we can store collection of values in single memory location. It stores only values. Internally js attaches index numbers to the values in array. By using index numbers we can access array values in app.

3) Function:

- Funtion is block of code. By using a function we can do some task and return some value.

- For every function call seperate excution context will be created.

- For every excution context, there are memory creation phase and code excution phase.

- We can stores multiple values in functional scope.

ES-6

4) Map

5) WeakMap

6) Set

7) WeakSet

3) typeOf operator

- By using typeof operator we can find data type for the value which is stored in memory.

value : datatype

'sachin': string

40: number

true/false: boolean

undefined: undefined

null: object

{}: object

[]: object

function(){}: function

4) operators:

why operators?

we use operators to develop some logic or expression in combination with variables.

1) Arithmatic

We use arithmatic operators to perform some mathematical operations.

+ add

- subtraction

\* multification

/ devision

% remainder

\*\* exponential

++ incremental : It increases 1 at a time.

\_\_ decremental : It decreases 1 at a time.

2) Assignment (=)

By using assignment operator we can assign/store value (data) to a variable.

3) Comparision

It compares the two values, the result from this operators will be true/false.

1) Rational/Relational

>

>=

<

<=

2) Equality

1) Loose equality (==) It compares only value of variables

2) Strict equality (===) It compares value and data type of variables

undefined == null

undefined === null

3) Not equality

1) Loose inequality (!=)

The != operator is the inequality operator. It checks whether two values are not equal, regardless of their types. If the values are different, it returns true. If the values are the same, it returns false.

2) Strict inequality (!==)

The !== operator is the strict inequality operator. It checks whether two values are not equal and whether they are of the same type. If the value or the type are different, it returns true; otherwise, it returns false.

4) Ternary operator

We use ternary operator to render content conditionally.

let age = 15;

let vote = age >= 18 ? 'Having vote' : 'Not having vote';

console.log(vote);

5) Logical operator

In JavaScript, logical AND (&&) and logical OR (||) are operators used to perform logical operations on boolean values or expressions.

1) logical and &&

2) logical or ||

3) ! Operator

The exclamation mark (!) is the logical NOT operator.

When used, it converts a true value to false and vice versa.

! is the logical NOT operator, used for negating boolean values.

6) Control statements

We use control statements to develop some logic or functionality when we have multiple conditions.

1) if else - In JavaScript, the if...else statement is used for conditional execution of code. It allows you to perform different actions based on a specific condition.

The syntax of the if...else statement is as follows:

if (condition) {

// Code block to be executed if the condition is true

} else {

// Code block to be executed if the condition is false

}

2) switch case (It does not work for step value)

- In JavaScript, the switch statement is another way to perform conditional excution of code based on the value of an expression. It is often used as an alternative to multiple if...else statements.

- The switch statement evaluates an expression once and then matches the value of the expression to a case label. If a matching case label is found, the corresponding block of code is executed.

7) Loops

- We use loops to do same task agin and again simply.

- We use loops to access memory value multiple times in an application.

Conditinal Loops

1) for loop:

In JavaScript, a for loop is used to execute a block of code repeatedly for a specified number of times.

2) while loop:

In JavaScript, a while loop is used to execute a block of code repeatedly as long as a specified condition is true. The loop continues until the condition evaluates to false.

3) do while loop:

In JavaScript, a do-while loop is similar to a while loop, but with a slight difference. The primary difference is that in a do-while loop, the loop body is executed at least once before the loop condition is checked.

This ensures that the loop body is executed at least once, regardless of whether the condition is initially true or false. The do-while

4) infinity loop:

An infinite loop in JavaScript is a loop that runs infinitely, continuously executing the same code block without ever stopping. This usually happens when the loop condition always evaluates to true

\*\*break=> by using Break keyword we can break the loop and make exit the loop.

\*\*continue=> by using continue keyword we can make jump the loop.  
Non conditional loops

5) for in loop:

We use this loop to iterate keys in object.

In JavaScript, the for...in loop is used to iterate over the enumerable properties of an object. It allows you to loop through the keys (property names) of an object and access their corresponding values.

6) for of loop:

In JavaScript, the for...of loop is used to iterate over the values of iterable objects(array/string object/set/map/arguments object/generator object/).. It provides an easy and concise way to loop through arrays, strings, sets, maps, and other objects that are iterable.

Note:

- It's important to note that the for...in loop should be used for iterating over object properties. If you want to loop through elements of an array, it is recommended to use the for...of loop or a simple for loop with an index.

- keep in mind that the for...of loop is not suitable for iterating over regular objects (objects created with {}) since they are not iterable by default. For iterating through object properties, you should use the for...in loop.

8) Funtions

1) What is function?

=>Function is a block of code which is used to do some task and return value.

=>It stores multiple values in functional scope.

=>For every function call seperate excution context will be created.

For every excution context there are memory creation phase and code excution phase.

2) How can we define function?

1) Function declaration

2) Function expression

1) named fucntion expression

2) anonymous function expression

3) arrow function (ES-6)

3) parameters vs arguments

Parameters

Parameters are the placeholders or variables defined in the function's declaration.

- They are like local variables that store the values passed to the function when it is called.

- They are defined within the function's parentheses in the function declaration and serve as placeholders for the arguments that will be provided when the function is called.

Arguments

Arguments are the actual values or expressions passed to a function when it is called.

- They represent the data that you want to work with within the function.

- The number of arguments should match the number of parameters defined in the function.

- Arguments are provided in the function call and are placed inside the parentheses.

4) what is default parameter

- Default parameters were introduced in ECMAScript 6 (ES6) and have since become a common feature in modern JavaScript.

- To define default parameters in a function, you assign a default value to a parameter in the function's declaration.

- Default parameters in JavaScript allow you to specify default values for function parameters.

5) Varying no of parameters or arguments

If we have varying no of params and arguments then we have to handle with below concepts.

=> arguments object (ES-5)

It takes all values at a time and stores in memory.

This is available in all functions except arrow function.

arguments object is an iterable object, it has symbol.iterator() method.

=> rest parameter (ES-6)

It starts with ...

It takes all values and stores in array.

It should be last parameter in parameter list.

6) Scope

1) Global scope

2) Function / local scope

3) Block scope

Note:

1) var is functionl scope

2) let, const is block scope

-In Js, "scope" refers to the context in which variables, functions are stored and can be accessed.

-The scope determines the accessability and lifetime of these variables and functions.

-Every execution context will create a new scope.

-Every function call will create new execution context.

JavaScript has two main types of scope:

Global Scope:

Variables declared or stored outside of any function or block have global scope.

They can be accessed from any part of the code, including inside functions and blocks.

Local Scope:

Variables declared or stored within a function or a block have local scope. They are only accessible within that specific function or block.

Note:

Local variables take precedence over global variables if they share the same name.

Function scope, which means that variables declared with var are only scoped within the function where they are defined.

However, with the introduction of let and const in ES6, block scope was introduced. Variables declared with let or const are scoped to the nearest enclosing block (defined by curly braces {}).

7) Hoisting:

=> Hoisting is default behaviour of javascript of moving all variable

declarations and function declarations to top of current scope.

=> Hoisting lets you allow to access memory value even before execution of code.

In JavaScript, both let and const declarations are hoisted, but they behave differently compared to var declarations.

When you declare a variable using var, it gets hoisted to the top of its scope and is initialized with undefined. However, let and const are also hoisted to the top of their scope, but they are not initialized. This means that you cannot access them before the actual declaration (you'll get a ReferenceError), which is known as the "temporal dead zone."

console.log(x); // undefined

var x = 5;

console.log(y); // ReferenceError:

let y = 10;

console.log(z); // ReferenceError: z

const z = 15;

In the above example, x is declared using var, so it gets hoisted and initialized with undefined. y and z are declared using let and const, respectively. While they are hoisted, accessing them before the actual declaration results in a ReferenceError due to the temporal dead zone.

both let and const declarations are hoisted in JavaScript. However, unlike var, they are not initialized with undefined. Instead, they enter a "temporal dead zone" until their declaration is encountered in the code. Attempting to access a variable declared with let or const before its declaration will result in a ReferenceError due to being in this temporal dead zone. This behavior ensures that variables declared with let or const are not accessible before they are defined in the code execution flow.

8) var vs let vs const

var let const

1) scope functional block block

2) hoisting yes yes but differently yes but differently

3) re-declaration yes no no

4) re-assignment yes yes no

5) initialization no need no need need

9) Closure:

=>Closure is a concept in js, it allows to inner function to access outer scope variables even after the outer function execution content is over.

9) Objects

1) What is an object?

- In JavaScript, an object is a reference data type that stores a collection of related data and functionality.

- It is a container for key-value pairs, where each key is a unique identifier and each value can be of any data type, including other objects, arrays, functions, primitives (such as numbers, strings, and booleans)

- Objects in JavaScript are often used to model real-world entities such as a person, car, or any other entity that has properties and behaviors.

- JavaScript also has some built-in objects like Array, Date, Math. These built-in objects provide specialized functionality for specific use cases.

- You can access the properties and methods of an object using dot notation or bracket notation.

2) How to create object:

1) Object literal way

3) Factory function

4) Constructor function

5) Classical way (ES-6)

3) Objects are dynamic:

You can add, modify, or delete properties and methods of an object at runtime, making objects dynamic and flexible in JavaScript.

-add

-update

-delete

4) Iterating object:

- for in loop

- Object.keys/ Object.entries /Object.values

Object.keys => This method will create array with looped keys in object.

Object.values => This method will create array with looped values in object.

Object.entries => This method will create array with looped keys and values in object.

5) Copy primitives and reference types:

=>primitives copy

By default deep copy

=>Object copy

normal copy => = (assignment operator)

swallow copy => Object.assign({}, source object) or spreed operator.

deep copy => JSON.parse(JSON.stringfy(obj))

(it will copy only pimitives and obj, it will ignore method copy)

cloneDeep() => lodash

(it will copy all pimitives and obj and method copy)

6) Math object

This is predefined object in javascript to do math related functionality.

1) min => It gives min value.

2) max => It gives max value

3) ceil => It rounds to up value when we have float value.

4) floor => It rounds to down value when we have float value.

5) round => It rounds to up value when we have 0.5 or above and down value when we have below 0.5.

6) random => It generates random value.

- Math.random() always returns a number lower than 1.

- Math.floor(Math.random() \* 10);

- Returns a random integer from 0 to 9:

- Math.floor(Math.random() \* 100);

- Returns a random integer from 0 to 99:

7) pow => In JavaScript, the Math.pow() function is used to calculate the power of a number. It takes two arguments: the base number and the exponent. Both Math.pow() and the exponentiation operator have the same functionality, but using the exponentiation operator can be more concise and readable in many cases.

8) sqrt => In JavaScript, the Math.sqrt() function is used to calculate the square root of a given number. It takes a single argument, which is the number whose square root you want to find. Math.sqrt(number);

7) Date object

This is predefined object in javascript.

By using this we can create dates.

1) var now = new Date() ===> It gives current date.

2) var x = new Date(2040, 0, 30, 9:00) ===> It is number date type we can create custom dates.

3) var x = new Date('2040 0 30 9') ===> It is string date type we can create custom dates.

8) Garbage collector

=> In low level languages like c, c++ when we create varible we need to allocate memory to it.

=> when we use it we have to de-allocate memory. we dont have this concept in javascript. we can easily create a variable at the time we initialized this variable the memory is automatically allocated to this variable next we can use that when we are done using we dont have to de-allocate memory.

=> So JS engine has garbage collector. The job of this garbage collector is to find the variable which are no longer used and then de-allocate the memory. Memory allocation and de-allocation allomatically done behind the scene you have no control on it.You cant tell garbage collector when to run and what varibales to remove from the memory.

=> Based on some complex algorithms(mark and sweep) the garbage collector run in the background it figurs out what variables are not used and then it automatically removes from memory.

9) Template literal (ES-6)

Synonyms:

Template Literals

Template Strings

String Templates

=> Template strings are a powerful feature of modern JavaScript released in ES6.

=> It lets you insert variables and expressions into strings without needing to concatenate + like in older versions of JavaScript.

=> It allows us to create strings that are complex and contain dynamic elements.

=> Template Literals use back-ticks (``) rather than the quotes ("") to define a string.

=> Template literals allows multiline strings.

syntax: `${}`

10) String Object

1) String length ===> It gives string length.

2) Removes white spaces

=>trim() ===> It removes white space both sides.

=>trimStart() ===> It removes white space at starting

=>trimEnd() ===> It removes white space at ending

3) Extracting part of string

=> slice(startindex, endindex)

=> substring(startindex, endindex)

=> substr(startindex, length)

slice()

-------

-The slice() method extracts a part of a string.

-The start and end parameters specifies the part of the string to extract.

-A negative number selects from the end of the string.

subString()

-The substring() method extracts characters from start to end (exclusive).

-Start or end values less than 0, are treated as 0. It does not work for negative values.

subStr()

-The substr() method extracts a part of a string.

-The substr() method begins at a specified position, and returns a specified number of characters.

-To extract characters from the end of the string, use a negative start position.

4) Extracting string characters.

=> charAt(index) - It returns the character at specified index.

=> charCodeAt(index) - It returns unicode of character at specified index.

5) Replacing string content

=> replace(exitingstring, newstring).

=> by default replace method replaces only first match.

=> replace method is case sensitive.

6) Converting to upper and lower case

=> toUpperCase ==> It converts string to UPPERCASE.

=> toLowerCase ==> It converts string to lowercase.

7) To Join two or more strings.

=> we use concat() method.

=> It is used to concat two or more strings.

=> concat method is used instead of plus operator.

8) String padding

=>padStart(target\_length, pad\_string)

=>padEnd(target\_length, pad\_string)

=>target\_length - The desired length of the resulting string after it has been padded.

9) Converting string into array.

=>split('')

-The split() method splits a string into an array of substrings.

-The split() method returns the new array.

-The split() method does not change the original string.

-If (" ") is used as separator, the string is split between words.

10) Methods for finding specific string.

=>indexOf() ==> It gives first occurance index number. if not return -1

=>lastIndexOf() ==> It gives last occurance index number.if not return -1

=>includes() ==> It gives true if string has that that.

=>startsWith() ==> It gives true if it has started with same string as you asked.

=>endsWith() ==> It gives true if it has ended with same string as you asked.

1) What is an array.

- In JavaScript, an array is a data structure used to store a collection of values.

Arrays allow you to store multiple values of different types or the same type into a single variable, making it easier to manage and manipulate collections of data.

Each value in an array is referred to as an "element," and each element has an associated index, starting from 0 for the first element.

Here's how you can create an array in JavaScript:

Creating an array of numbers

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

Creating an array of strings

const fruits = ['apple', 'banana', 'orange'];

Creating an array of mixed types

const mixedArray = [1, 'hello', true, null, undefined];

You can access elements in an array using their index:

console.log(numbers[0]); // Outputs: 1

console.log(fruits[1]); // Outputs: banana

- Arrays have a variety of methods that you can use to manipulate their contents, such as adding or removing elements, iterating through elements.

Here are a few common array methods:

push: Adds an element to the end of the array.

pop: Removes the last element from the array.

shift: Removes the first element from the array.

unshift: Adds an element to the beginning of the array.

length: Returns the number of elements in the array.

splice(): Can be used to add, remove, or replace elements at a specific position in the array.

concat(): Combines two or more arrays to create a new array.

slice(): Creates a new array by extracting a portion of the existing array. Creates a shallow copy of a portion of an array.

forEach: Executes a function for each element in the array.

map: Creates a new array by applying a function to each element in the array.

filter: Creates a new array with elements that pass a test condition.

2) adding of elements to an array

push: It adds new item to end of array.

unshift: It adds new item to start of array.

splice: It adds new item at any place in array.

3) removing specific element from an array

pop: It removes end value in array.

shift: It removes start value in array.

splice: It removes value in any where in array.

4) removing all elements from an array

[]

length = 0

splice()

5) finding of elements in an array

primitives:

indexOf: It finds index of value at first occurance in array.

If that value is not find in array it gives -1

lastIndexOf: It finds index of value at last occurance in array.

If that value is not find in array it gives -1

includes: If given is in array then it gives true as output.

If given value is not found in array it gives false as output.

reference:

find:

It finds the whether the given value is in array then it returns the same value.

If the given value is not there in array then it returns undefind.

findeIndex:

It finds the whether the given value is in array then it returns the index number

of that value.If the given value is not there in array then it returns -1.

6) combining two or more arrays as one array.

1) concat() => es-5

var combineArr = arr1.concat(arr2)

2) spread operator => es-6

var combineArr = [...arr1, ...arr2]

7) copy/clone an array.

1) Normal copy => (= assignment operator)

2) Shallow copy

=> ES-5 slice()

=> ES-6 spread operator

3) Deep copy

=> without method JSON.parse(JSON.stringfy(array))

=> with method lodash (cloneDeep(array))

1) Normal copy

2) Shallow copy

1. Using the slice() Method:

- The slice() method can be used to create a shallow copy of an array. It takes two optional arguments: the starting index and the ending index. If no arguments are provided, it copies the entire array.

const originalArray = [1, 2, 3, 4, 5];

const shallowCopy = originalArray.slice();

2. Using the Spread Operator ([...]):

- The spread operator allows you to create a shallow copy of an array in a concise way.

const originalArray = [1, 2, 3, 4, 5];

const shallowCopy = [...originalArray];

- Both of these methods create a new array that contains the same elements as the original array. However, keep in mind that the elements themselves are still references to the same objects. If the array contains objects or arrays, changes made to those objects or arrays will be reflected in both the original array and the shallow copy.

const originalArray = [{ value: 1 }, { value: 2 }, { value: 3 }];

const shallowCopy = originalArray.slice();

shallowCopy[0].value = 100;

console.log(originalArray[0].value); // Output: 100

console.log(shallowCopy[0].value); // Output: 100

- As seen in the example, modifying an object within the shallow copy also affects the corresponding object in the original array. If you need to create an independent copy where changes to one array don't affect the other, you would need a deep copy.

3) Deep copy

- you can create a deep copy of an array using the lodash library in JavaScript. Lodash provides a convenient method called \_.cloneDeep() that allows you to create a deep copy of an array or any nested data structure, handling all the complexities of deep copying effectively.

- cloneDeep() function from the lodash library takes care of creating a complete and independent copy of the array and its nested elements, ensuring that modifications to the deep copy do not affect the original array.

8) iterating of elements in an array

1) for of:

=>for of loop iterates an array values only.

2) forEach:

=>The forEach() method calls a function for each element in an array.

=>forEach() method is used to loop through values and index of an array.

forEach method does not return any value. It returns undefined.

9) Joining of an array

join()

=>The join() method returns an array as a string.

=>The join() method does not change the original array.

=>Any separator can be specified. The default is comma (,).

10) testing of an array elements

1) some():

=>some method in JavaScript is used to check whether at least one of the elements of the array satisfies the given condition or not.

=>The only difference is that the some() method will return true if any predicate is true while every() method will return true if all predicates are true.

2) every():

=>every method in JavaScript is used to check whether all the elements of the array satisfy the given condition or not.

=>The output will be false if even one value does not satisfy the element,

else it will return true, and it opposes the some() function.

11) filtering of array

filter()

=>The filter() method creates a new array filled with elements that pass a test provided by a function.

=>The filter() method does not execute the function for empty elements.

=>The filter() method does not change the original array.

12) Sorting of an array

1) Ascending order

2) Descending order

primitives:

sort()

reverse()

reference:

sort + comparison function

=> nested for loop

13) mapping of array

map()

=>The map() method creates a new array populated with the results of

calling a provided function on every element in the calling array.

=>It returns new array with transforming values.

14) reduce of array

reduce()

=>The reduce() method executes a reducer function for array element.

=>The reduce() method returns a single value: the function's accumulated result.

=>The reduce() method does not change the original array.

15) Flattening of array(converting multi dimensional array into single dimension array).

1) flat() method

2) Array.isArray()

16) How to remove dulpicate elements in an array.

1) Set with spread operator

2) indexOf()

17) map() vs forEach()

- The map() method returns a new array, whereas the forEach() method does not return a new array

- The map() method is used to transform the elements of an array, whereas the forEach() method is used to loop through the elements of an array.

18) Array.from()

- The Array.from() method returns an array from any object with a length property.

- The Array.from() method returns an array from any iterable object.

- Array.from() is an ECMAScript6 (ES6) feature.

- Array.from(object, ()=>{} )

- object: This parameter holds an object that will convert into an array.

This method can create an array from:

-array-like objects - The objects that have length property and have indexed elements like String.

-Iterable objects like Map or Set.

Return Value

-Returns a new Array instance.

The Array.from() method, which is new in ES6, creates a new instance of the Array from an object that acts like an array or is iterable. The syntax for the Array.from() method is shown below:

Array.from(target [, mapFn[, thisArg]])

target is an object to be converted into an array that is iterable or array-like. The map function, or mapFn, should be used on each element of the array.The value used to call the mapFn method is thisArg.

A new instance of Array is returned by the Array.from() method, and it contains every element of the source object.

19) at()

The at() method takes an integer value and returns the item at that index, allowing for positive and negative integers. Negative integers count back from the last item in the array.

20) fill()

The fill() method in JavaScript is used to fill all the elements of an array from a start index to an end index with a static value.

It mutates the original array and returns the modified array.

The fill() method in JavaScript is used to fill all the elements of an array from a start index to an end index with a static value.

It mutates the original array and returns the modified array.

21) Array.isArray()

The Array.isArray() static method determines whether the passed value is an Array or not.

11) DOM

1) What is DOM?

- DOM stands for Document Object Model. It is a programming interface for web documents, primarily used to access and manipulate the content and structure of web pages. The DOM represents the page as a tree-like structure where each element on the web page, such as HTML tags, text, and attributes, is treated as an object in the tree.

Here are some key points about the DOM:

- The DOM represents the structure of HTML document as a tree, where each node in the tree corresponds to an element or part of the document.

- The DOM is dynamic, You can use JavaScript to interact with the DOM, change content, add or remove elements, and respond to user actions.

- The DOM allows you to attach event listeners to elements on a web page so that you can respond to user interactions like clicks, mouse movements, and keyboard input.

- Web browsers use the DOM to render web pages. When a web page is loaded, the browser parses the HTML and constructs the DOM tree, which it uses to display and render the page.

- Here's a simple example in JavaScript of how you can use the DOM to manipulate a web page:

- Get a reference to an DOM element with an 'id' attribute of 'myId'

var element = document.getElementById("myId");

- Change the text content of the element

element.textContent = "Hello, World!";

- Add a new element to the page

var newElement = document.createElement("p");

newElement.textContent = "This is a new paragraph.";

document.body.appendChild(newElement);

In this example, we use the DOM to select an element with the ID "myId" change its text content, and create a new paragraph element that is appended to the document body. This demonstrates how the DOM can be used to manipulate the content a web page dynamically.

2) How to examine DOM

console.dir(document) => It gives whole DOM with lot of properties.

3) How to read DOM properties

document.propertyname => We can read DOM properties by using dot operator or []

4) How to select DOM elements

=> id

var para1 = document.getElementById('one');

=> class

var x = document.getElementsByClassName('one');

=> tag

var x = document.getElementsByTagName('p');

=> querySelector

var x = document.querySelector('#one');

var x = document.querySelector('.one');

var x = document.querySelector('p');

=> querySelectorAll

var x = document.querySelectorAll('#one');

var x = document.querySelectorAll('.one');

var x = document.querySelectorAll('p');

Note: By using id we can get only one element. id is unique.(single element as result)

By using querySelector we can get first element only.(single element as result)

By using className we will get html collection in array.

By using tagName we will get html collection in array.

By using querySelectorAll we will get node list in array.

5) Traversing of DOM

element to parent

=> element.parentElement

element to child

=> element.firstElementChild

=> element.lastElementChild

=> element.children

element to siblings

=> element.previousElementSibling

=> element.nextElementSibling

6) Adding / removing / replacing DOM elements

1) Adding

parent.appendChild(child)

parent.insertBefore(new child, existing child)

2) Replacing

parent.replaceChild(new child, existing child)

3) Remove

element.remove()

parent.removeChild(child)

7) Adding events to DOM elements from Javascript.

var button = document.getElementById('btn')

DOMelement.addEventListener('eventname', function(event handler), false/true)

- false: Bubling phase

- true: Capturing phase

8) Events handling in JS.

1) Event bubling:

- Event bubbling means propagation of an event is done from child element to ancestor elements in the DOM.

- When an event happens on an element, event first runs the handlers on it, then on its parent, then all the way up on other ancestors. Propagation path is bottom to top.

2) Event capturing:

- Event capturing means propagation of event is done from ancestor elements to child element in the DOM. Propagation path is top to bottom.

3) Event deligation:

- Event delegation in JavaScript is a pattern that efficiently handles events.

- Events can be added to a parent element instead of adding to every single element.

- It refers to the process of using event propagation to handle events at

a higher level in the DOM than the element on which the event originated.

- This can be done on particular target element using the .target property of an event object.

- when you click the td, the event bubbles up to the table which handles the event.

Why Event Delegation?

- It is useful because the event can be listened to on multiple elements by using

just one event handler.

- It also uses less memory and gives better performance.

- Apart from this,it also requires less time for setting up the event handler on element.

9) How to improve JS app performance when we firing event in app:

1) Normal event: It is about firing event every time.

2) Throttling event: Throtteling is about firing event after every certain time.

3) Debouncing event: Debouncing is about firing event after every certain time, provided in between there was no event firing.

12) Asyncronous JS

Synchronous JavaScript:

As the name suggests synchronous means to be in a sequence, i.e. every statement of the JS code gets executed one by one. So, basically a statement has to wait for the earlier statement to get executed.

Let us understand this with the help of an example.

console.log("Hi"); // First

document.write("Hari") ;// Second

document.write("How are you"); // Third

In the above code snippet, the first line of the code 'Hi' will be logged first then the second line 'Hari' will be logged and then after its completion, the third line would be logged 'How are you'.

So as we can see the codes work in a sequence. Every line of code waits for its previous one to get executed first and then it gets executed.

Asynchronous JavaScript:

Asynchronous code allows the program to be executed immediately where the synchronous code will block further execution of the remaining code until it finishes the current one. This may not look like a big problem but when you see it in a bigger picture you realize that it may lead to delaying the User Interface.

console.log("Hi");

setTimeout(() => {

console.log("Hello here what is go to happen");

}, 2000);

console.log("End");

- So, what the code does is first it logs in 'Hi' then rather than executing the setTimeout function it logs in End and then it runs the setTimeout function. At first, as usual, the Hi statement got logged in. As we use browsers to run JavaScript, there are the web APIs that handle these things for users.

- So, what JavaScript does is, it passes the setTimeout function in such web API and then we keep on running our code as usual. So it does not block the rest of the code from executing and after all the code its execution, it gets pushed to the call stack and then finally gets executed. This is what happens in asynchronous JavaScript.

// console.log('Hello');

// setTimeout(() => {

// console.log('one');

// }, 4000);

// Promise.resolve()

// .then(() => console.log('PR-1'))

// .then(() => console.log('PR-2'));

// console.log('bye');

// setTimeout(() => {

// console.log('two');

// }, 0);

1) Callback functions

- A JavaScript callback is a function which is to be executed after another function has finished execution.

- A more formal definition would be - Any function that is passed as an argument to another function so that it can be executed in that other function is called as a callback function.

- When we have to process the data further then callback syntax will create callback hell situation. we can not read the callback code.

Why use callback function in JavaScript?

Asynchronous programming:

- Callbacks are used to handle the results of asynchronous operations, which means that the operation does not block the execution of the rest of the program.

- Instead, the program continues to run and the callback function is executed when the operation is complete.

2) Promises

Promises are a way to implement asynchronous programming in JavaScript(ES6 which is also known as ECMAScript-6). A Promise acts as a container for future values.

- A promise is a JavaScript object that allows you to make asynchronous calls. It produces a value when the async operation completes successfully or produces an error if it doesn't complete.

- let promise = new Promise(function(resolve, reject) { Do something and either resolve or reject})

- Promise has 3 states 1) pending 2) success 3) failure.

a) Promise.all()

Promise.all() will reject as soon as one of the Promises in the array rejects.

b) Promise.allSettled()

- whereas Promise.allSettled() waits for all the promises to settle (either resolve or reject) before returning an array of objects representing each promise's outcome.

- Promise.allSettled will never reject - it will resolve once all Promises in the array have either rejected or resolved.

c) Promise.race()

- The Promise.race() method returns a promise that fulfills or rejects as soon as one of the promises in an iterable fulfills or rejects, with the value or reason from that promise.

- It returns the winning promise only.

example:

var p1 = new Promise((res, rej) => {

setTimeout(() => {

res('pr-1');

}, 3000);

});

var p2 = new Promise((res, rej) => {

setTimeout(() => {

res('pr-2');

}, 2000);

});

var p3 = new Promise((res, rej) => {

setTimeout(() => {

res('pr-3');

}, 5000);

});

Promise.all([p1, p2, p3]).then((res) => {

console.log(res);

});

Promise.allSettled([p1, p2, p3]).then((res) => {

console.log(res);

});

Promise.race([p1, p2, p3]).then((res) => {

console.log(res);

});

3) async and await

- Async and await are built on promises. The keyword “async” accompanies the function, indicating that it returns a promise.

- Within this function, the await keyword is applied to the promise being returned.

- The await keyword ensures that the function waits for the promise to resolve.

API CALLS

The fetch() method in JavaScript is used to request data from a server. The request can be of any type of API that returns the data in JSON . The fetch() method requires one parameter, the URL to request, and returns a promise.

api for the get request:

fetch('url')

.then(response => response.json())

.then(data => console.log(data));

URL: It is the URL to which the request is to be made.

Return Value:

It returns a promise whether it is resolved or not. The return data can be of the JSON format. It can be an array of objects or simply a single object.

NOTE: Without options, Fetch will always act as a get request.

Making Post Request using Fetch:

Post requests can be made using fetch by giving options as given below:

var obj = {

userId: 101,

title: 'my title',

body: 'my body',

};

fetch('https://jsonplaceholder.typicode.com/posts', {

method: 'POST',

headers: {

'Content-type': 'application/json',

},

body: JSON.stringify(obj),

}).then((res) => {

console.log(res);

});

13) iterators and generators

- Iterator is a new concept introduced in ES-6. It's a kind of new mechanism to iterate or

traverse through data structures.

- Arrays, strings, Maps, Sets all these data collections are iterable.

Iterable:(object)

An iterable is any object that impliments a method whose key is symbol.iterator

and symbol.iterator method is going return an iterator object.

Iterator:(object) It returns by symbol.iterator()

- What is an iterator? An iterator is an object that is going to impliment next method.

- This next method knows how to access elements in a collection , next method returns an

object(IteratorResult object).

- That IteratorResult object contains two properties {value: any datatype, done: boolean}

var iterable = [1, 2, 3, 4, 5];

let iterator = iterable[Symbol.iterator]();

console.log(iterator);

console.log(iterator.next());

console.log(iterator.next());

console.log(iterator.next());

console.log(iterator.next());

console.log(iterator.next());

console.log(iterable.next());

let person = {

fname: 'Hari',

lname: 'Ravilla',

};

person[Symbol.iterator] = function () {

var properties = Object.keys(person);

var count = 0;

var isDone = false;

var next = () => {

if (count >= properties.length) {

isDone = true;

}

return { done: isDone, value: this[properties[count++]] };

};

return { next };

};

for (var v of person) {

console.log(v);

}

By default iterable data structures

-array

-Map

-Set

-String

-Generator object

-arguments object

By default not iterable data structures

-object

Generator:

-Generators can help you to pause & resume the code.

-It is a function which can return multiple values in phases.

-The function\* is the keyword used to define a generator function.

-Yield is an operator which pauses the generator.

-Yield itself is capable of returning any value.

-Genarators are iterable.

-Next () method returns an object, which has two keys.

1) value

2) boolean

-Generator function returns an itarator object.

-To excute generator function we have to call next() method.

-When you want to come out of generator function or terminate it you can write return()method,

Writing yield inside finally will not allow the return() to terminate gen function.

Note: When a generator function is called, it does not call the function instead it returns a generator object.

Note: next method will start the excution till the yield operator. Next method returns an object which has two keys {value: any datatype, done: true/false}

14) ES-6 Modules

- Consider a scenario where parts of JavaScript code need to be reused. ES6 comes to rescue with the concept of Modules.

- A module organizes a related set of JavaScript code. A module can contain variables and functions.

- A module is nothing but a chunk of JavaScript code written in a file.

- By default, variables and functions of a module are not available for use.

- Variables and functions within a module should be exported so that they can be accessed (imported) from within other files.

Named Exports

Named exports are distinguished by their names. There can be several named exports in a module. A module can export selected components using the syntax given below −

Syntax 1

-using multiple export keyword

export component1

export component2

...

...

export componentN

Syntax 2

Alternatively, components in a module can also be exported using a single export keyword with {} binding syntax as shown below −

-using single export keyword

export {component1,component2,....,componentN}

Default Exports

Modules that need to export only a single value can use default exports. There can be only one default export per module.

Syntax

export default component\_name

However, a module can have one default export and multiple named exports at the same time.

Importing Named Exports

While importing named exports, the names of the corresponding components must match.

Syntax

import {component1,component2..componentN} from module\_name

However, while importing named exports, they can be renamed using the as keyword.

import {original\_component\_name as new\_component\_name }

Importing Default Exports

Unlike named exports, a default export can be imported with any name.

Syntax

import any\_variable\_name from module\_name

Note: We have to do extra 2 tasks 1) type: "module" 2) add .js to file extension

15) ES-6 Tooling

1) Babel

=> Babel is a transpiler for ES6 to ES5, so that browser can understand the JS.

=> Babel can convert the advanced js and jsx into pure js, which is understand by browser.

=> It allows web developers to take advantage of the newest features of the JS language.

This is just for example to show how babel work

==> npm init --yes

==> npm install babel-cli@6.26.0 babel-core@6.26.0 babel-preset-env@1.6.1 --save-dev

==> Create build folder

==> In scripts: "babel": "babel --presets env index.js -o build/index.js"

==> npm run babel

Note: It will create new js file in build folder with ES-5 syntax, We mensioned index.js file in script to run, But if we have more files to run we need to add more files. Solution for this is we have to use Webpack it will take all files at a time and convert into ES-5 them make these files into bundle. Before creating bundle, each file will be taken by babel to convert into ES-5. After completion of conversion, all files will be bundled by webpack.

2) Webpack

=> Webpack is an open-source JavaScript module bundler. It is a build tool that is primarily used for bundling assets of a web application, such as JavaScript files, CSS stylesheets,and images. Webpack takes various modules and their dependencies and bundles them into a single or multiple output files, typically optimized for deployment in a web browser.

=> Webpack can also optimize code and assets for production, reducing the file size of the application and improving performance.

Note: Webapack runs our code through babel and it converted to ES-5.

==> Finally make changes in index.html:

- remove type="module" in script tag and

- change src path, serve the "dist/main.bundle.js"

- In scripts: "build": "webpack -w" This is for automatic build for every changes made.

- run everytime to build: num run build

16) ES-6 Destruturing

- The destructuring is a JavaScript expression that makes it possible to unpack values from arrays or unpack properties from objects, into individual variables.

1) Objects

2) Arrays

17) Browser APIs (Timer functions)

1) setTimeout

setTimeout(function, milliseconds)

Executes a function, after waiting a specified number of milliseconds.

2) setInterval

setInterval(function, milliseconds)

Same as setTimeout(), but repeats the execution of the function continuously.

3) clearTimeout

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

var myTimeout = setTimeout(function, milliseconds);

clearTimeout(myTimeout);

4) clearInterval

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

let myVar = setInterval(function, milliseconds);

clearInterval(myVar);

18) Miscellaneous topics (related to functions)

1) IIFE (Immediately Invocable Function Expression) (ES-5)

=>An IIFE is a way to excute functions immediately as soon as they are created.

=>The most usecase of an IIFE is to restrict the scope of variables to local so that they don't pollute the global context.

=>It helps in making our variables and methods private.

=>Any function or variable defined inside IIFE can not be accessed outside of the IIFE block, thus preventing global scope from getting polluted.

=>It's often used to create private variables and functions.

2) Function currying?

=>Function currying is a technique of trasforming (converting) a function with multiple parameters into multiple functions with single parameter.

=>It helps to avoid passing same value again and again.

3) first class function?

=>A programming language is said to have First-class functions when functions in that language are treated like any other variable.

=>For example, in such language, a function can be passed as an argument to other functions, can be returned by another function and can be assigned as a value to a variable.

4) heigher order function?

=>A function that receives another function as an argument or that returns a new function or both is called Higher-order function. Higher-order functions are only possible because of the First-class functions

5) pure function?

=>Pure function always returns the same result if the same arguments are passed in.

=>It does not depend on any state, or data, change during a program's execution. It must only depend on its input arguments .

=>They do not have any side effects like network or database calls and do not modify the arguments which are passed to them.

=>We can predict the output from pure function.

6) impure function?

=>Any function that changes the internal state of one of its arguments or the value of some external variable is an impure function.

=>They may have any side effects like network or database calls and it may modify the arguments which are passed to them.

=>We can not predict output from impure fucntion.

7) Recursion function:

- A function that calls itself is recursion. Recursion in JavaScript is a recursive call of the function to itself, where the function calls itself, again and again, (recursively) until given condition becomes false.

calculate factorial:

function factorial(n) {

if (n <= 0) {

return 1;

} else {

return n \* factorial(n - 1);

}

}

let result = factorial(5);

console.log(result);

19) Types of errors in javascript

1) Syntax error:

The error occurs when you use a predefined syntax incorrectly.

ex :

var obj = {

name = 'hari'

}

2) Reference error:

A ReferenceError occurs when you try to access a variable that doesn't exist in memory.

ex:

console.log(x);

3) Type error:

A TypeError occurs when the variable exists, but the operation you're trying to perform on it is not appropriate for the type of value it contains.

ex :

var x = 100;

x.push(10);

20) Errors handling in JavaScript:

- Handling errors in JavaScript is essential to ensure your code is clean and provides a good user experience. JavaScript provides several mechanisms to handle errors effectively. Here are some common methods:

Using try...catch:

- The try...catch block allows you to handle exceptions (errors) that occur within a specific block of code. You can place the code that might throw an error inside the try block, and if an error occurs, it will be caught and processed in the catch block.

try => try statement lets you test a block of code for error.

catch => catch statement lets you handle the error.

throw => throw statement lets you create custom errors.

try{

code to be tested for error

}

catch(error){

error handling

}

5) Type conversion

Type conversion is the process of converting data of one type to another.

For example: converting String data to Number.

There are two types of type conversions in JavaScript.

Implicit Conversion - automatic type conversion by js.

Explicit Conversion - manual type conversion by developer.

JavaScript Implicit Conversion

In certain situations, JavaScript automatically converts one data type to another. This is known as implicit conversion.

Example 1: Implicit Conversion to String

1) numeric string used with + gives => string type as output

let result;

result = '3' + 2;

console.log(result) // "32"

result = '3' + true;

console.log(result); // "3true"

result = '3' + undefined;

console.log(result); // "3undefined"

result = '3' + null;

console.log(result); // "3null"

Note: When a number is added to a string, JavaScript converts the number to a string before concatenation.

Example 2: Implicit Conversion to Number

1) numeric string used with - , / , \* results => number type as output.

let result;

result = '4' - '2';

console.log(result); // 2

result = '4' \* 2;

console.log(result); // 8

result = '4' / 2;

console.log(result); // 2

Example 3: Non-numeric String Results to NaN

=> non-numeric string used with - , / , \* results to NaN type as output.

let result;

result = 'hello' - 'world';

console.log(result); // NaN

result = '4' - 'hello';

console.log(result); // NaN

Example 4: Implicit Boolean Conversion to Number

=> if boolean is used, true is 1, false is 0

let result;

result = '4' - true;

console.log(result); // 3

result = 4 + true;

console.log(result); // 5

result = 4 + false;

console.log(result); // 4

Note: JavaScript considers 0 as false and all non-zero number as true. And, if true is converted to a number, the result is always 1.

Example 5: null Conversion to Number

=> null is 0 when used with number

let result;

result = 4 + null;

console.log(result); // 4

result = 4 - null;

console.log(result); // 4

Example 6: undefined used with number, boolean or null

=> Arithmetic operation of undefined with number, boolean or null gives NaN

let result;

result = 4 + undefined;

console.log(result); // NaN

result = 4 - undefined;

console.log(result); // NaN

result = true + undefined;

console.log(result); // NaN

result = null + undefined;

console.log(result); // NaN

JavaScript Explicit Conversion

You can also convert one data type to another as per your needs. The type conversion that you do manually is known as explicit type conversion.

In JavaScript, explicit type conversions are done using built-in methods.

1. Convert to Number Explicitly

To convert numeric strings and boolean values to numbers, you can use Number(). For example,

let result;

// string to number

result = Number('324');

console.log(result); // 324

// boolean to number

result = Number(true);

console.log(result); // 1

result = Number(false);

console.log(result); // 0

In JavaScript, empty strings and null values return 0. For example,

let result;

result = Number(null);

console.log(result); // 0

let result = Number(' ')

console.log(result); // 0

If a string is an invalid number, the result will be NaN

let result;

result = Number('hello');

console.log(result); // NaN

result = Number(undefined);

console.log(result); // NaN

result = Number(NaN);

console.log(result); // NaN

You can also generate numbers from strings using parseInt(), parseFloat(), unary operator + and Math.floor(). For example,

let result;

result = parseInt('20.01');

console.log(result); // 20

result = parseFloat('20.01');

console.log(result); // 20.01

result = +'20.01';

console.log(result); // 20.01

result = Math.floor('20.01');

console.log(result); // 20

To convert other data types to strings, you can use either String() or toString(). For example,

//number to string

let result;

result = String(324);

console.log(result); // "324"

result = String(2 + 4);

console.log(result); // "6"

//other data types to string

result = String(null);

console.log(result); // "null"

result = String(undefined);

console.log(result); // "undefined"

result = String(NaN);

console.log(result); // "NaN"

result = String(true);

console.log(result); // "true"

result = String(false);

console.log(result); // "false"

// using toString()

result = (324).toString();

console.log(result); // "324"

result = true.toString();

console.log(result); // "true"

To convert other data types to a boolean, you can use Boolean().

In JavaScript, undefined, null, 0, NaN, '' converts to false. For example,

let result;

result = Boolean('');

console.log(result); // false

result = Boolean(0);

console.log(result); // false

result = Boolean(undefined);

console.log(result); // false

result = Boolean(null);

console.log(result); // false

result = Boolean(NaN);

console.log(result); // false

All other values give true. For example,

result = Boolean(324);

console.log(result); // true

result = Boolean('hello');

console.log(result); // true

result = Boolean(' ');

console.log(result); // true

Truthy values and falsy values in javascript

1) (false); // false

2) (undefined); // false

3) (null); // false

4) (''); // false

5) (NaN); // false

6) (0); // false

=> (true); // true

=> ('hi'); // true

=> (1); // true

=> ([]); // true

=> ([0]); // true

=> ([1]); // true

=> ({}); // true

=> ({ a: 1 }); // true

23) Data structures

In JavaScript, keys of objects are always strings or symbols. Even if you use a number as a key, it gets converted to a string. Here's an example to illustrate this:

let obj = {

name: "Alice",

age: 25,

1: "one",

true: "boolean",

null: "null",

undefined: "undefined",

symbolKey: Symbol("symbolValue")

};

for (let key in obj) {

console.log(`${key}: ${typeof key}`);

}

In this example, even though some keys appear to be numbers, booleans, null, or undefined, they are all converted to strings. The output will be:

1: string

name: string

age: string

true: string

null: string

undefined: string

symbolKey: string

Symbols are a special case. If a symbol is used as a key, it does not show up in a regular for...in loop or Object.keys() method. To handle symbols, you can use Object.getOwnPropertySymbols.

Here's how you can identify both string and symbol keys in an object:

let obj = {

name: "Alice",

age: 25,

[Symbol("symbolKey")]: "symbolValue"

};

// Check string keys

for (let key in obj) {

console.log(`${key}: ${typeof key}`);

}

// Check symbol keys

let symbolKeys = Object.getOwnPropertySymbols(obj);

for (let sym of symbolKeys) {

console.log(`${sym.toString()}: ${typeof sym}`);

}

In JavaScript, keys of objects are always strings or symbols. Even if you use a number as a key, it gets converted to a string. Here's an example to illustrate this:

javascript

Copy code

let obj = {

name: "Alice",

age: 25,

1: "one",

true: "boolean",

null: "null",

undefined: "undefined",

symbolKey: Symbol("symbolValue")

};

for (let key in obj) {

console.log(`${key}: ${typeof key}`);

}

In this example, even though some keys appear to be numbers, booleans, null, or undefined, they are all converted to strings. The output will be:

1: string

name: string

age: string

true: string

null: string

undefined: string

symbolKey: string

Symbols are a special case. If a symbol is used as a key, it does not show up in a regular for...in loop or Object.keys() method. To handle symbols, you can use Object.getOwnPropertySymbols.

Here's how you can identify both string and symbol keys in an object:

let obj = {

name: "Alice",

age: 25,

[Symbol("symbolKey")]: "symbolValue"

};

// Check string keys

for (let key in obj) {

console.log(`${key}: ${typeof key}`);

}

// Check symbol keys

let symbolKeys = Object.getOwnPropertySymbols(obj);

for (let sym of symbolKeys) {

console.log(`${sym.toString()}: ${typeof sym}`);

}

This will output:

name: string

age: string

Symbol(symbolKey): symbol

In summary, JavaScript object keys are either strings or symbols. Regular object keys (including those that look like numbers or booleans) are converted to strings, and symbol keys need to be handled separately using Object.getOwnPropertySymbols().

When to use map:

With map, you can use any type (and values) as keys.

Map provides better performance for large quantities of data.

Use a map for better performance when adding and removing data frequently.

When to use an object:

Objects can only use strings, and symbols as keys.

Objects are perfect for small to medium-sized sets of data.

Objects have better performance and are easier to create.

**Map**

Map is a collection of keyed data items, just like an Object. But the main difference is that Map allows keys of any type.

how to create empty

var obj = {}

var map = new Map();

how to initialize with values

var obj = {name: 'sachin', age: 40}

const map = new Map([ [ 'one', 1 ], [ 'two', 2 ] ]);

how to add new properties

obj[color] = 'red'

obj.color = 'red'

map.set(key, value) -- stores the value by the key.

how to get or access values

obj.key

obj['key']

map.get(key) -- returns the value by the key, undefined if key doesn't exist in map.

how to check specific key is there

The in operator checks if a key exists in the object, including keys from the prototype chain.

The hasOwnProperty method checks if a key exists directly on the object, not considering keys from the prototype chain.

console.log('name' in obj)

console.log(obj.hasOwnProperty('name'))

map.has(key) -- returns true if the key exists, false otherwise.

how to delete specific key

delete obj.name;

map.delete(key) -- removes the value by the key.

how to delete all keys

Using a loop with the delete operator

for (let key in obj1) {

if (obj1.hasOwnProperty(key)) {

delete obj1[key];

}

}

Setting the object to an empty object

obj = {};

console.log(obj); // {}

map.clear() -- clears the map

how to find length

alt:1

let length = Object.keys(obj).length;

alt:2

let length = 0;

for (let key in obj) {

if (obj.hasOwnProperty(key)) {

length++;

}

}

alt:3

let length = Object.getOwnPropertyNames(obj).length;

map.size -- returns the current element count.

how to loop keys

alt:1

for (let key in obj) {

if (obj.hasOwnProperty(key)) {

console.log(`${key}: ${obj[key]}`);

}

}

alt:2

let keys = Object.keys(obj);

for (let i = 0; i < keys.length; i++) {

let key = keys[i];

console.log(`${key}: ${obj[key]}`);

}

alt:3

Object.keys(obj).forEach(key => {

console.log(`${key}: ${obj[key]}`);

});

Similar to objects, there are three methods you can use for looping over Maps:

map.keys() returns an iterable containing the keys

map.values() returns an iterable containing the values

map.entries() returns an iterable containing the [key, value] pairs

how to convert

let map = new Map(Object.entries(obj));

let obj = Object.fromEntries(map);

how to destructure

let { one, two } = obj;

let { one, two } = map;

console.log(one) // undefined

// But you can destructure it similar to an array where you destructure by the order items were added into the map

let [ firstEntry, secondEntry ] = map;

console.log(firstEntry) // ["one", 1]

console.log(secondEntry) // ["two", 2]

-remember the regular Object? it would convert keys to string.

-Map keeps the type, so these two are different.

-As we can see, unlike objects, keys are not converted to strings. Any type of key is possible.

-Map can also use objects as keys.

map vs weakMap

In a Weak Map, every key can only be an object and function. It is used to store weak object references.

One main difference when using a WeakMap is that the keys have to be objects, not primitive values. Which means they will pass by reference.

So why use a WeakMap? The major advantage of using a WeakMap over a Map is memory benefits.

Objects that are not-reachable get garbage collected, but if they exist in as a key in another reachable structure then they won't get garbage collected.

WeakMap does not prevent garbage-collection of it's key objects.

WeakMaps only have the following methods: get, set, delete, has.

It does not support for clear, size, keys, values, forEach

set vs weekSet

It is used to store a collection of objects similar to that of set the only difference is that these values are only object and not some particular data type.

Sets can store any value. WeakSets are collections of objects only.

WeakSet does not have size property.

WeakSet does not have clear, keys, values, entries, forEach methods.

WeakSet is not iterable.

summary

-Use Map when you need to associate values with keys.

-Use WeakMap when you want to associate values with keys and the keys should not prevent garbage collection.

-Use Set when you need a collection of unique values.

-Use WeakSet when you need a collection of unique objects, and the objects should not prevent garbage collection.

24) Browser Object Model (Bom)

The Browser Object Model (BOM) allows JavaScript to "talk to" the browser.

The Browser Object Model (BOM) is a set of objects provided by web browsers that allows JavaScript to interact with the browser environment.

Here are some key objects in the Browser Object Model:

The Browser Object Model (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:

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

The window object represents a window in browser. An object of window is created automatically by the browser.

Window is the object of browser, it is not the object of javascript. The javascript objects are string, array, date, math etc.

1. window Object:

-A window object is created automatically by the browser itself.

-The window object is the top-level object in the BOM hierarchy and represents the browser window.

-It contains properties and methods that allow to interaction with the browser window.

-The window object is supported by all browsers. It represents the browser's window.

-All global JavaScript objects, functions, and variables automatically become members of the window object.

-Global variables are properties of the window object.

-Global functions are methods of the window object.

-Even the document object (DOM) is a property of the window object.

-Two properties can be used to determine the size of the browser window.

window.innerHeight - the inner height of the browser window (in pixels)

window.innerWidth - the inner width of the browser window (in pixels)

-timimg functions:

window.setTimeout()

window.setInterval()

window.clearTimeout()

window.clearInterval()

-popup functions:

window.alert()

window.confirm()

window.prompt()

window.open() - open a new window

window.close() - close the current window

window.moveTo() - move the current window

window.resizeTo() - resize the current window

2. document Object:

The document object represents the HTML document loaded in the browser window.

It provides methods and properties to manipulate the content and structure of the document.

// Examples of document object properties

console.log(document.title); // Title of the document

console.log(document.body); // Body element of the document

// Examples of document object methods

document.getElementById('myElement'); // Returns the element with the specified ID

document.createElement('div'); // Creates a new HTML element

3. navigator Object:

The navigator object provides information about the browser and the user's system.

Examples of navigator object properties

console.log(navigator.appName);

The appName property returns the application name of the browser.

console.log(navigator.platform);

Operating system platform

navigaror.language

The language property returns the browser's language:

Is The Browser Online?

navigator.onLine;``

The onLine property returns true if the browser is online:

4. screen Object:

The screen object represents the user's screen and provides information about its size and resolution.

// Examples of screen object properties

console.log(screen.width); // Width of the screen

console.log(screen.height); // Height of the screen

5. location Object:

The location object provides information about the current URL and allows you to navigate to different URLs.

The window.location object can be used to get the current page address (URL) and to redirect the browser to a new page.

// Examples of location object properties

console.log(location.href); // Full URL of the current page

console.log(location.pathname); // Path of the current URL

// Example of location object method

location.assign('https://www.example.com'); // Navigates to the specified URL

window.location.href returns the href (URL) of the current page

window.location.hostname returns the domain name of the web host

window.location.pathname returns the path and filename of the current page

window.location.protocol returns the web protocol used (http: or https:)

The window.location.assign() method loads a new document.

<h3>The window.location object</h3>

<script>

function newDoc() {

window.location.assign("https://www.youtube.com")

}

</script>

6) History object

The history object contains the URLs visited by the user (in the browser window).

The history object is a property of the window object.

The history object is accessed with:

window.history or just history:

back() Loads the previous URL (page) in the history list

(Will only work if a previous page exists in your history list)

forward() Loads the next URL (page) in the history list

(Will only work if a next page exists in your history list)

go() Loads a specific URL (page) from the history list

(Will only work if the previous pages exist in your history list)

lengthReturns the number of URLs (pages) in the history list

][history.go(0) reloads the page.

history.go(-1) is the same as history.back().

history.go(1) is the same as history.forward().