Js tutorial

1. const → value will not change furthur.

2. console.table([variables,…]) → give you all data in tabular form

3. there is 3 types of variables in js var, let, const

don’t use var because of issues in block scope and function scope.

4. “use strict” // treat all Js code as newer version of JS.

5. //alert() is function which run well with in console but not in node js….

6. js has number range number =>2^53

for large numbers then using bigint

Data types

1. number => 2^53

2. String

3. bigint

4. boolean

5. null → is standalone value and data type also in JS

6. undefined → value is not given

7. symbol   
this is primitive data types

non-primitive data types

null is object.

Undefined is undefined.

8. if you are converting a string such as “34ab” then it will converted to number but actual stored value will be NaN→ Not a Number.

Same for the undefined → NaN

null → 0, true → 1, false → 0

if you are converting “” to boolean it will be false.

Also “random string” then it will return true.

Console.log(+true); gives you 1

console.log(true+); is illegal in JS.

9. console.log(null > 0) → false

console.log(null == 0) → false

console.log(null >= 0) → true  
reason:- the comparison <, >, <=, >= and equality == works differently comparison converts null to a number treating it as a zero. That’s why null >= 0 gives true.  
Undefined comparison and equality both gives value false.  
10. === → strict check. It compares both value and data type.

11. `half of 100 is ${100 / 2}`

When you write something inside ${} in a template literal, its result will be

computed, converted to a string, and included at that position. This example

produces the string "half of 100 is 50".

12. The ?? operator resembles || but returns the value on the right only if the

one on the left is null or undefined, not if it is some other value that can be

converted to false. Often, this is preferable to the behavior of ||.   
 console.log(0 ?? 100);

// → 0

console.log(null ?? 100);

// → 100  
13. primitive data types are string, int, bigint, number, Boolean, null, undefined, symbol → pass by value

non-primitive data types (reference type) are arrays, objects, functions → it is pass by reference

14. Memory management in Javascript

stack is used for primitive data types, Heap is used for the non-primitive data types.

userTwo

Before :- {

email: "User@gmail.com",

upi:"user@ybl"

}  
after :- {  
email: “bhavesh@google.com”

upi: “user@ybl”

userOne

myName

anotherName

myName

Stack heap

code :- let myName = "Bhavesh"

let anotherName = myName

anotherName ="Bhavesh Gadling"

*console.log*(myName)

*console.log*(anotherName)

let userOne = {

*email*: "User@gmail.com",

*upi*:"user@ybl"

}

let userTwo = userOne;

*userTwo.*email = "Bhavesh@google.com"

*console.log*(*userOne.*email)

*console.log*(*userTwo.*email)

15. for writing concatination of strings and number we write like :-  
 *console.log*(`Hello my name is ${name} and i am having ${repoCount} repos on github`)

this is called as string interpolation.

16. string is a object not a array of characters → const *gameName* = new *String* ("bhavesh")

if you run this on console of browser then you will find key value pairs of string which look like

0:“b”

1: "h"

2: "a"

3: "v"

4: "e"

5: "s"

6: "h"

length: 7

<primitive value>: "bhavesh"

17. String methods

//string

const *name* = "Bhavesh"

const *repoCount* = 3

*console.log*(name+repoCount)

*console.log*(`Hello my name is ${name} and i am having ${repoCount} repos on github`)

const *gameName* = new *String* ("My name is bhavesh gadling")

*console.log*(gameName[0])

*console.log*(*gameName.*\_\_proto\_\_) // for the whole string

*console.log*(*gameName.charAt*(0))// gives the character at index and if you entered index > strlength or index < 0 then it will result nothing

*console.log*(*gameName.indexOf*('h')) //gives the index of the first matching character

*console.log*(*gameName.lastIndexOf*(gameName))

const *newString* = *gameName.substring*(0, 5) //divide whole sting into parts and return it negative indexing is not handled

*console.log*(newString);

const *anotherstring* = *gameName.slice*(0,4)// divide whole string from 0 to 4 negative indexing is handled

*console.log*(anotherstring)

const *newStringone* = " Bhavesh "

*console.log*(*newStringone.trim*()) //trim the string from end and start and only remove ' ' , '\n' chars from string

const *url* = "https://bhavesh.com/bhavesh%20Gadling"

*console.log*(*url.replace*('%20', '-')) // replace only first matching characters of string and other than first char the string remains unchanged

*console.log*(*url.includes*('h'))// returns true or false if 'h' is in string or not.

*console.log*(*gameName.split*(' '))// split string on the basis of space or any other character u had given in ''

18. for defining explicitly of type of a variable

use → const <var name> = *new* <type>(<value>)

19. toFixed() is the method in JS which will give you how many digits should be print after the decimal

console*.*log(score*.*toFixed(6)) → x*.*000000 whatever values stored in the score*.*

20. toprecision() is the method which is give you precise value

console*.*log(otherNumber*.*toPrecision(3)) → give you value if number is contain 3 digit before then it will round of the decimal value if it is contain less than 3 digit then it will round of the digit after decimal if number is greater than 3 digit then it will give in exponential form

21. toLocalString() is method which add the commas to the number and give the output

const *hundreds* = 1000000000

*console.log*(*hundreds.toLocaleString*())

output :- 1,000,000,000

for indian standards *console.log*(*hundreds.toLocaleString*('en-IN')) this will give you string according to the indian standards…*.*

22. In Javascript Math is the object in Javascript which has a lot of method and fixed values…

for generating random number we can use the → *console.log*(*Math.random*()) it always give you value between the 0 to 1.

if you want to define range of values then

code :-

const *max* = 6

const *min* = 1

*console.log*(*Math.floor*(*Math.random*() \* (max - min + 1)) + 1)

it will generate the number including max and min both*.*

23. There is object in Javascript called Date which has a lots of methods….

let mydate = new *Date*()

*console.log*(mydate)

*console.log*(*mydate.toString*());

*console.log*(*mydate.toDateString*());

*console.log*(*mydate.toISOString*())

*console.log*(*mydate.toJSON*())

*console.log*(*mydate.toLocaleDateString*())

*console.log*(*mydate.toLocaleTimeString*());

this are the major methods of date object….

24. if you want to create the custom dates in JS then u can declare like this

let mycreatedDate = new *Date*(2024, 5, 20, hr, min)

*console.log*(*mycreatedDate.toString*())

and month counting is starts from 0 not from 1.

25. when you want a specific format…. Then you can do it like

let myCretedDate = new *Date*("2024-01-20")

*console.log*(*myCretedDate.toLocaleString*())

but in this case the month indexing starts from the 1 not 0.

26. for more customization of date you can use It

let newDate = new *Date*()

*newDate.toLocaleString*('default', {

weekday : "long"

}

)

27. Arrays in Javascript are resizable. It can also contain mixed type of elements like numbers, string or character. Indexing is zero based always starts with zero….

28. copying objects create shallow copies(copies whose property share the same references) of objects. And deep copies are copies whose property does not share the same reference.

29. there are two important methods in Javascript .slice() and .splice()

slice(small, large) → this operates on values in Array, large value is not included and does not affect the original array. You can also say

slice(x, y) → starting from index ‘x’ add elements excluding index ‘y’….

splice(small, large) → it include large value And large value is number of elements that is to be add in the spliced array and remove the copied element from original array. You can also say

splice(x, y) → starting from index ‘x’ add ‘y’ no. of elements from the original array and delete the added elements from the original array….

30. when you want to combine two arrays if you are using

const *marvel* = ["Iron Man", "Thor", "Spiderman"]

const *dc* = ["Superman", "Flash", "Batman"]

method 1:-

*marvel.push*(dc)

*console.log*(marvel)

push method the array marvel will look like this :-

[ 'Iron Man', 'Thor', 'Spiderman', [ 'Superman', 'Flash', 'Batman' ] ] push method use mutate the array so if you want to access Superman, Flash, Batman then you have to do like   
*console.log*(marvel[3][0])

this is the syntax

method 2 :-

const *all\_heros* = *marvel\_heros.concat*(dc\_heros)

*console.log*(all\_heros)

concat method combine two arrays and give the new array and does not mutate original array….

It can’t mutate the array….

Method 3 :-

using spread

const *allNewHeros* = [...marvel\_heros, ...dc\_heros]

*console.log*(allNewHeros);

**… → spread out the array**

**this will combine the both arrays.**

31. if you have give the situation like

const *another\_array* = [1,2,3, [4, 5, 6], 7, [6, 7, [4, 5]]]

this then you can spread out the array using flat() function

const *usable\_array* = *another\_array.flat*(*Infinity*)

*console.log*(usable\_array);

flat() → argument is depth, At which depth you want to spread arrays of arrays of arrays….

Argument → Infinity is a hack for this….

32. if you want to check if the input Is array or not

*console.log*(*Array.isArray*("Bhavesh"))

*for objects you can’t make a array using object it will create a empty array*

*console.log*(*Array.from*({name : "Bhavesh"}))

let score1 = 100

let score2 = 200

let score3 = 300

*console.log*(*Array.of*(score1, score2, score3))

this will combine all elements in an same array

33.when you make a object from a constructor then it will be an singleton

for defining a symbol in object which is used as a symbol you have to define it like

const *mySym* = *Symbol*("MyKey1")

const *JsUser* = {

*name*: "Bhavesh",

*age* : 17,

[mySym]: "MyKey1", ← this is the symbol

*location*: "Amravati",

*email* : "bhavesh@google.com",

*isloggedIn* : *false*,

*lastLoginDate*: ["monday", "Saturday"]

}

//methods to access value from an Object

*console.log*(JsUser["email"])

*console.log*(typeof JsUser[mySym])

*Object.freeze*(JsUser) ← it used to lock the object and it can not apply to one element of object it is only apply to object so after this the Object JsUser cannot change further.

34.

*JsUser.greeting* = function() {

*console.log*("Hello User!")

}

*JsUser.greetingTwo* = function() {

*console.log*(`Hello I am ${this*.*name}`)

}

so the function greeting Is function defined in JsUser object

this keyword is the keyword which is used for the accessing the current object elements.

35. you can define object like this

const *regularUser* = {

*name*:"Bhavesh",

*fullname*:{

*UserFullName*: {

*firstName*:"Bhavesh",

*surName*: "Gadling"

}

}

}

// console.log(regularUser.fullname.UserFullName.firstName)

const *obj1* = {1:"a", 2:"b"}

const *obj2* = {3:"c", 4:"d"}

const *obj3* = *Object.assign*({}, obj1, obj2)

*console.log*(obj3);

In this case the this assign method store all the combined so always use first this curly braces to ensure that obj1 should not change…. Unless the obj1 will be changed after this statement.

const *obj3* = {...obj1, ...obj2}

this is one more syntax which says spread obj1 and obj2 and copy it in the obj3

36. if you want to fetch all keys from an object then you will use

const *arr* = *Object.keys*(tinderuser1)

*console.log*(typeof arr)

it fetches all the keys from the tinderuser1 and store it in an array you can use Object.values() and Object.entries() to fetch values and values & keys both. But after using Object.entries() you will get an array of array.

37. if you want to check whether a value exist in dataset you can use

*console.log*(*tinderuser1.hasOwnProperty*('isloggedin'))

hasOwnProperty() is a method which returns a boolean if the value exist in tinderuser1 or not if yes return true if no return false and it is case sensitive.

38. JSON is called as Javascript object notations which is used for making the api

39. The delete operator cuts off a tentacle from such an octopus. It is a unary

operator that, when applied to an object property, will remove the named

property from the object.

let anObject = {left: 1, right: 2};

console.log("left" in anObject);

this is the notation for checking whether anObject contain the value left.

40. There’s an Object.assign function that copies all properties from one object

into another:

let objectA = {a: 1, b: 2};

Object.assign(objectA, {b: 3, c: 4});

console.log(objectA);

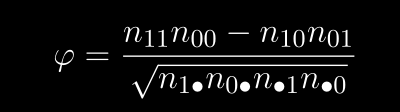
// → {a: 1, b: 3, c: 4}

41. When you compare objects with JavaScript’s == operator, it compares by iden-

tity: it will produce true only if both objects are precisely the same value.

Comparing different objects will return false.

42. this is the formula to find co-relation between two boolean values.



The notation n01 indicates the number of measurements where the first variable is false (0) and the second variable (pizza) is true (1). The value n1• refers to the sum of all measurements where the first variable is true. Likewise, n•0 refers to the sum of the measurements where the second variable is false.

43. To search from the end instead of the start, there’s a similar method called lastIndexOf()

44. One difference is that a string’s indexOf can search for a string containing more than one character, whereas the corresponding array method looks only for a single element:

console.log("one two three".indexOf("ee"));

// → 11

45. It can be useful for a function to accept any number of arguments. For example,

Math.max computes the maximum of all the arguments it is given. To write

such a function, you put three dots before the function’s last parameter, like

this:

function max(...numbers) {

let result = -Infinity;

for (let number of numbers) {

if (number > result) result = number;

}

return result;

}

console.log(max(4, 1, 9, -2));

// → 9

46. Note that if you try to destructure null or undefined, you get an error, much

as you would if you directly try to access a property of those values.

Serialize means that object to string.

deSerialize means string to object.

47. JavaScript gives us the functions JSON.stringify and JSON.parse to convert

data to and from this format.

The first takes a JavaScript value and returns

a JSON-encoded string. The second takes such a string and converts it to the

value it encodes:

let string = JSON.stringify({squirrel: false,

events: ["weekend"]});

console.log(string);

// → {"squirrel":false,"events":["weekend"]}

console.log(JSON.parse(string).events);

// → ["weekend"]

48. function is defined in Javascript like this   
 function *sayMyName*() {

*console.log*("Bhavesh")

}

*sayMyName*()

if you type only sayMyName and didn’t type () then it is the reference to the function sayMyName() and it will not execute further.

If you want to set a default value then u can use

function *loginUserMessage*(username = "Bhavesh") {

if (!username) {

*console.log*("Please enter a user name")

*return*

} else

*return* `${username} is just logged in!.`

}

this will set the username default value “Bhavesh” if empty string is provided in a function argument then it will set username to “Bhavesh” and if the non-empty string is provided then the string username string will be overide.

49. there is a rest and spread operator in js. rest is said when we are creating function and want that the function should take infinite number of argument and spread when we want to combine an Array. Then we say rest.

Operator look like … this the operator

function *calculateCartPrice*(...num1) {

*return* num1

}

*console.log*(*calculateCartPrice*(200, 300, 500, 600))

it will give you an array.

50. Each local scope can also see all the local scopes that

contain it, and all scopes can see the global scope. This approach to binding

visibility is called lexical scoping.

51. let launchMissiles = function() {

missileSystem.launch("now");

};

if (safeMode) {

launchMissiles = function() {/\* do nothing \*/};

}

it is the code in which we are giving a value to the function.

This is the notations of defining functions in Javascript

const *roundTo* = (n, step) => {

let remainder = n % step;

*return* n - remainder + (remainder < step / 2 ? 0 : step);

};

const *square1* = (x) => { *return* x \* x; };

const *square2* = x => x \* x;

it is also the notation of defining functions.

52. The place where the computer stores this context is the call stack. Every

time a function is called, the current context is stored on top of this stack.

When a function returns, it removes the top context from the stack and uses

that context to continue execution.

53. JavaScript is extremely broad-minded about the number of arguments you

can pass to a function. If you pass too many, the extra ones are ignored. If

you pass too few, the missing parameters are assigned the value undefined.

54. function multiplier(factor) {

function *multiplier*(factor) {

*return* number => number \* factor;

}

let twice = *multiplier*(2);

*console.log*(*twice*(5));

// → 10

In this code the multiplier(2) → number => number \* 2 (after that this function is stored in twice as value and then it is called twice(5) which → 5 => 5 \* 2 and output it to the 10.

55. the implementation of recursive function has one problem: in typical JavaScript implementations, it’s about three times slower than a version using a for loop.

Running through a simple loop is generally cheaper than calling a function

multiple times.

56. In this code

addtwo(2)

const addtwo = function(x) {

return x + 2

}

we can’t access the addtwo() before declaration.

This is called as hoisting.

57. In JavaScript, const creates an immutable binding to a value, not an immutable value itself. Like if u define like

const user = {

username : "Bhavesh",

price: 999,

welcome: function() {

console.log(`Hi ${this.username}, welcome to website`)

}

}

user.welcome()

user.username = "Anay"

user.welcome()

then you can only change the inner value of object user not the user it self.   
Ex:- user = {/\* another object\*/ } it is not possible.

58. this keyword always talk about current context. In browser the global object is window object

function chai() {

let username = "hitesh"

console.log(this.username)

}

chai()

if you run this code you will get undefined. Because it only works on the objects not In side function.

const addtwoNum = (num1, num2) => num1 + num2

this we call implicit declaration of arrow function.

59. Immediate invoked function expression is the expression which will run exactly after starting the expression. This is defined like this

*//*Immeadiate invoked function expression

(

function one() {

console.log("this is Immediate invoked function expression");

})()

(wrapping) (calling of function)

reason for using the IIFE is used from decrease the problem which occurred from global scope pollution.

*//*Immeadiate invoked function expression

(

function one() {

*//*named IIFE

console.log("this is Immediate invoked function expression");

})();

(() => {

*//*Normal IIFE

console.log(`DB connected two`) }) ();

in this code the ; is must for it to execute the next II function expression.

60. predicate function is the function which return only boolean value.

61.

function speak(line) {

console.log(`The ${this.type} rabbit says '${line}'`);

}

let whiteRabbit = {type: "white", speak};

let hungryRabbit = {type: "hungry", speak};

whiteRabbit.speak("Oh my fur and whiskers");

// → The white rabbit says 'Oh my fur and whiskers'

hungryRabbit.speak("Got any carrots?");

// → The hungry rabbit says 'Got any carrots?'

in this case the this.type is object → (type : “”)

\*In this code the the speak function is assigned as method to the object whiteRabbit and hungryRabbit.

62.

let finder = {

find(array) {

return array.some(v => v == this.value);

},

value: 5

};

console.log(finder.find([4, 5]));

**Arrow functions do not have their own** this.  
They **use the** this **value of the context where they were defined**.

* this.value inside the arrow function refers to finder.value because the arrow function was defined inside the find() method of finder.

#### ****With Regular Function:****

**let finder1 = {**

find(array) {

return array.some(function(v) {

return v == this.value; // <-- This is the key

});

},

value: 5

}

**this does not refer to finder. It is refering to the global value not finder.value.**

**It refers to the global object or undefined (in strict mode).**

**So this.value is undefined, and comparison fails.**

**Arrow functions inherit** this **from where they are defined.**

* **Regular functions have their own** this, which can lead to unexpected behavior if not bound.

**One way to create a rabbit object type with a speak method would be to create**

**a helper function that has a rabbit type as its parameter and returns an object**

**holding that as its type property and our speak function in its speak property.**

**All rabbits share that same method. Plain old objects created with {} notation are linked to an object called Object.prototype.**

****Object Oriented Programming in Javascript****

****classes :-** A class defines the shape of a type of object—**

**what methods and properties it has. Such an object is called an instance of**

**the class.**

**Constructor :- In JavaScript, a constructor is a special function used to create and initialize objects. It's typically used with classes or as a function constructor to create multiple objects with the same structure and behavior.**

**Constructors, in**

**JavaScript, are called by putting the keyword new in front of them. Doing so**

**creates a fresh instance object whose prototype is the object from the function’s**

**prototype property, then runs the function with this bound to the new object,**

**and finally returns the object.**

**class Person {**

constructor(name, age) {

this.name = name;

this.age = age;

}

//any number of methods you can define in this

greet() {

console.log(`Hello ${this.name} you are age is ${this.age}`)

}

}

const person = new Person("Bhavesh", 17)

person.greet()

console.log(person.age)

Private Properties :- To declare a private method, put a # sign in front of its name. Such methods

can be called only from inside the class declaration that defines them.

****Overiding the prototypes of Object** :- function Rabbit(type) {**

**this.type = type;**

**}**

**Rabbit.prototype.teeth = "small";**

**let killerRabbit = new Rabbit("killer");**

**console.log(killerRabbit.teeth); // → small**

**killerRabbit.teeth = "long, sharp, and bloody";**

**console.log(killerRabbit.teeth); // → long, sharp, and bloody**

**console.log((new Rabbit("basic")).teeth); // → small**

**console.log(Rabbit.prototype.teeth); // → small**

**When you assign a property directly to an object that already exists on its prototype, you’re not changing the prototype’s property—you’re creating a new one on the object. This hides (or overrides) the prototype's property only for that object.**

****Maps :-****

**let ages = {**

**Boris: 39,**

**Liang: 22,**

**Júlia: 62**

**};**

**console.log(`Júlia is ${ages["Júlia"]}`);**

**// → Júlia is 62**

**console.log("Is Jack's age known?", "Jack" in ages);**

**// → Is Jack's age known? false**

**console.log("Is toString's age known?", "toString" in ages);**

**// → Is toString's age known? True**

**in this code the toString object you didn’t add but it is giving true because by default in object.prototype there is the key value pair called toString but u didn’t add it. So how you will fix this**

**If you pass null to Object.create, the resulting object will not**

**derive from Object.prototype and can be safely used as a map.**

**console.log("toString" in Object.create(null));**

**// → false**

**Object property names must be strings.**

**if you need a map whose keys can’t easily be converted to strings—such as objects—you cannot use an object as your map. Fortunately, JavaScript comes with a class called Map that is written for this exact purpose. It stores a mapping and allows any type of keys.**

**let ages = new Map();**

**ages.set("Boris", 39);**

**ages.set("Liang", 22);**

**ages.set("Júlia", 62);**

**console.log(`Júlia is ${ages.get("Júlia")}`);**

**// → Júlia is 62**

**console.log("Is Jack's age known?", ages.has("Jack"));**

**// → Is Jack's age known? false**

**console.log(ages.has("toString"));**

**// → false**

**The methods set, get, and has are part of the interface of the Map object.**

**Interface :- interface is like a set of rules or expectations that an object or function must follow. It tells us what properties or methods an object should have, without worrying about how they’re implemented.**

**Polymorphism :-**

**Code can work with many types of objects, as long as they behave in a certain way (i.e., follow the same "interface").**

🧠 1. **Getters**: Reading a Property That Runs Code

let varyingSize = {

get size() {

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

}

};

console.log(varyingSize.size); // e.g. → 73

console.log(varyingSize.size); // e.g. → 49

this is test code for getters which only reads the properties of Objects.

🧊 2. **Setters**: Writing to a Property That Runs Code

class Temperature {

constructor(celsius) {

this.celsius = celsius;

}

get fahrenheit() {

return this.celsius \* 1.8 + 32;

}

set fahrenheit(value) {

this.celsius = (value - 32) / 1.8;

}

}

this is the ex of getters & setters.

Static method is the methods which define on the class itself not on the interface.

An object is **iterable** if it can be used in a for...of loop or with the spread operator like this:

for (let item of something) { ... }

let arr = [...something];

To be iterable, the object must have a **special method**:

something[Symbol.iterator]. ← This method must return an **iterator**

### ****🔑 Key Concepts****

1. **Iterable Object**:  
   Has a method named [Symbol.iterator]() that returns an iterator.
2. **Iterator Object**:  
   Has a .next() method that returns an object with:
   * value: the next item
   * done: true if iteration is complete
3. for...of **Loop**:  
   Automatically calls [Symbol.iterator]() and uses .next() to iterate values.
4. **Custom Iterable Example**:  
   You can make your own data structure (like a linked list) iterable by:
   * Creating a separate iterator class
   * Adding [Symbol.iterator] to the prototype of the main class
5. **Spread Operator Works**:  
   The ... operator can also be used to consume iterable values.

### ****🔑 Key Points****

1. extends **keyword**:
   * Used to create a subclass (e.g., LengthList) based on a superclass (e.g., List).
2. super() **in constructor**:
   * Calls the superclass’s constructor to set up inherited properties.
3. **Private Fields (**#length**)**:
   * Used to safely store internal data, like the precomputed length.
4. **Avoiding Redundant Computation**:
   * The length getter returns the stored value instead of traversing the list.
5. **Using** super.length:
   * Accesses the original getter from the superclass to compute length once.
6. **Encapsulation & Inheritance**:
   * Inheritance allows reuse of structure and behavior but can increase complexity and coupling between classes.
7. **Inheritance Caution**:
   * Powerful but should be used sparingly to avoid tightly coupled, tangled code.

**It is occasionally useful to know whether an object was derived from a specific**

**class. For this, JavaScript provides a binary operator called instanceof.**

**console.log(**

**new LengthList(1, null) instanceof LengthList);**

**64. JavaScript has 3 execution context**

**I) Global execution context**

**ii) Function Execution context**

**ii) Eval Execution context**

**javascript has been run in two phases**

**I) memory creation phase → only memory is allocated to variables. All variables assigned undefined In this phase. For function memory is allocated to defination of function.**

**ii) Execution phase → actual value will be assigned to variables.**

**If every single function call will create a new execution context and also memory phase will also repeat for the variables in function and for function parameters. And also a new thread is also created. A function returning a value will return it to the global excution context.**

**65. there is a null**

**for nullish coalescing operator(??) : null or undefined**

**which checks whether the first value is null if yes then assign it second value.**

**66. for of loops**

**syntax :- for (const element of array) {**

console.log(element);

}

**67. The** Map **object holds key-value pairs and remembers the original insertion order of the keys. Any value (both objects and** [primitive values](https://developer.mozilla.org/en-US/docs/Glossary/Primitive)**) may be used as either a key or a value.**

**68. Objects are not iterabable, so you cannot iterate over there**

**myobj = {**

'game1': "NFS",

'game2': "PUBG",

'game3': "FIFA",

'game4': "COD",

}

for (let [key, value] of myobj) {

console.log(key, ' :- ', value)

}

**as it is demonstrated in this.**

**//for in loop is used for the iterate over the object**

// for (const key in myobj) {

// console.log(` key of the myobj is ${key} value of it is ${myobj[key]}`);

// }

// let programming = ["js", "py", "cpp", "java"]

// console.log(`programmer must know languages:- `)

// for (key in programming) {

// console.log(`${programming[key]}`)

// }

//for in loop is not work for the maps()

//foreach loop

let coding = ["js", "cpp", "python", "ruby", "java"]

// coding.forEach( function(item) {

// console.log(item)

// });

// coding.forEach( (item) => {

// console.log(item)

// })

// function printme(item) {

// console.log(item)

// }

// coding.forEach(printme)

// coding.forEach( (item, index, arr) => {

// console.log(item, index, arr)

// })

let myCoding = [

{

lang : "Javascript",

langFileName: "js"

},

{

lang : "Java",

langFileName : "java"

},

{

lang : "Python",

langFileName : "py"

},

]

myCoding.forEach( (item) => {

console.log(item.langFileName)

})

**69. filter() is the method which takes the predicate function and returns the elements which satisfy the condition**

**const myNums = [1, 2, 3, 4, 5, 6, 7, 8, 9,10]**

let newnums = myNums.filter( (x) => (x >= 4)) // it accepts a function.

console.log(newnums)

let newnums1 = myNums.filter( (x) => {

return x >= 4

})

**this is the code for it.**

**70. .map is method which modify the whole function of it.**

**const myNums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]**

// const newNums = myNums.map( (x) => x + 10)

let newNums = myNums

.map( (nums) => nums \* 10)

.map( (nums) => nums + 1)

.filter( nums => nums > 40)

like that it is also called as chaining.

71. There is .reduce method which takes arguments like accumulater and currentvalue

const myNums = [1, 2, 3]

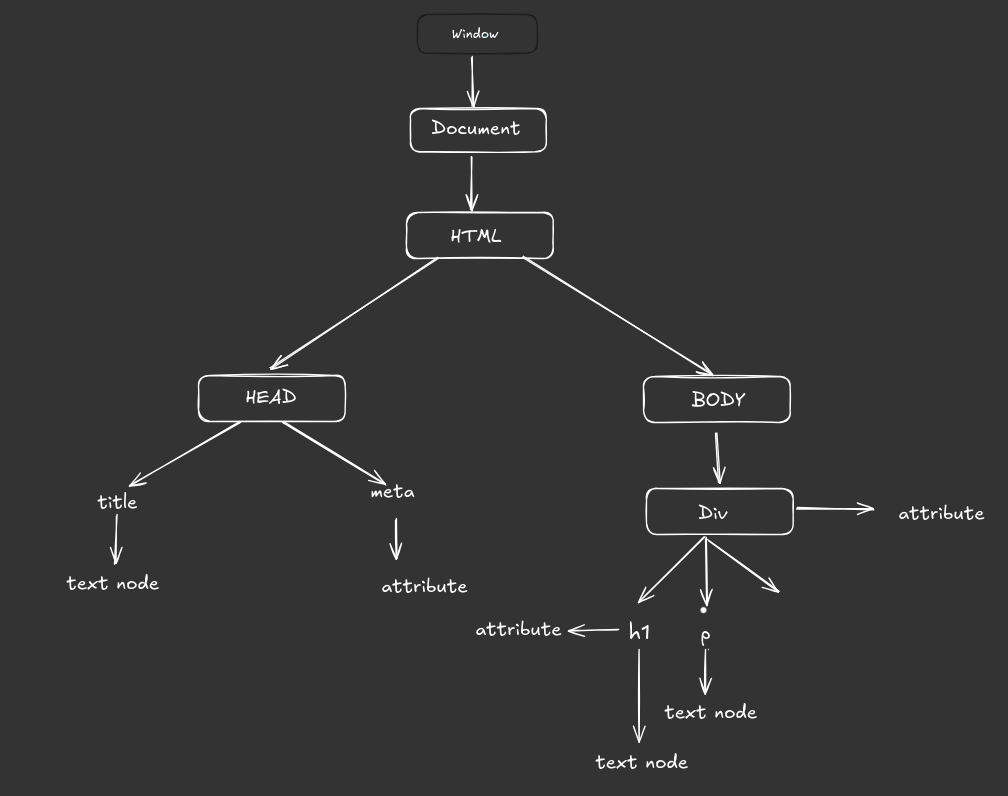
const total = myNums.reduce( (acc, currval) => {

console.log(`acc is ${acc} currval is ${currval}`)

return acc + currval}, *0* )

and this work like that set accumulator to 0 and the add currval to it then set accumulator to currval + accumulator as like that…..

**DOM(Document Object Model)**



this is the diagram of the how the DOM is exist in this.

IN dom the return value of the getElementById() Is a whole html element.

Now in every html page the elements are objects only.

Now how to get values from it.

There are 3 method to get the value.

1. innerHTML → it shows the whole value of it.

2. innerText → it does not show hidden text.

3. textcontent → it shows the hidden text also.

When you typed document.querySelector() then you will get an node list not an array,

when you typed getElementByClassName() then you will get HTML collection which is different from the node list.

If you have to convert HTML collection to node list then u can use

Array.from(nodelist)

this will return a array of node list.

If you want to go from node to upper side like if you selected a inner query and u Want to select a upper query of that then u can do

console.log(day1.parentElement)

child nodes are counted like /n and comment but it will count only first ‘/n’ as node of tree

### Event Propagation in JavaScript

Event propagation is how events flow through the DOM tree when triggered on an element. It has **two main phases**:

#### 1. ****Capturing Phase (Event Capturing)****

* The event starts from the **top (window/document)** and travels **down** to the target element.
* Listeners set with capturing mode (useCapture = true) catch the event during this phase.
* Rarely used but important in some scenarios.

#### 2. ****Target Phase****

* The event reaches the **target element** where it was originally triggered.
* Event listeners on the target element run here.

#### 3. ****Bubbling Phase (Event Bubbling)****

* The event bubbles **upwards** from the target element back to the root.
* Listeners set with bubbling mode (useCapture = false or omitted) catch the event here.
* This is the default behavior in JavaScript.

### Visual Flow:

Window → Document → ... → Parent Element → Target Element ← Parent Element ← ... ← Document ← Window

(Capturing) (Target) (Bubbling)

### Controlling Propagation

* **event.stopPropagation()**: Stops the event from bubbling or capturing further.
* **event.preventDefault()**: Prevents the default browser action (not propagation).
* You can specify capturing or bubbling when adding listeners:

javascript

element.addEventListener('click', handler, true); *// Capturing phase*

element.addEventListener('click', handler, false); *// Bubbling phase (default)*

Blocking code

→ block the flow of program

→ such as read file synchronouse

non-Blocking code

→ does not block the flow of program.

→ Such as Read file Asynchronouse

Promises

the Promise object is represents the eventual completion (or failure) of an asynchronous operation and its resulting value.

States of promises:

pending:- initial state neither fulfilled nor rejected.

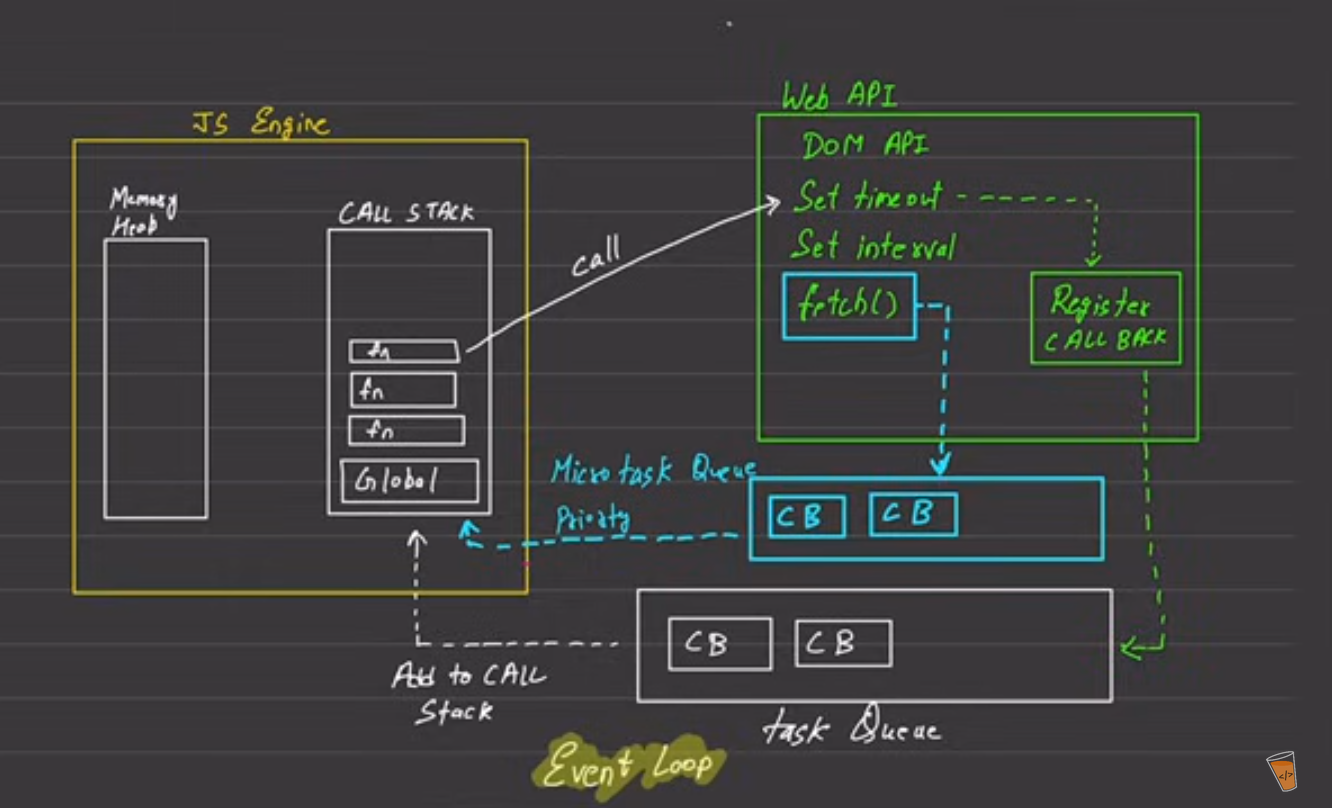
Fulfilled :- operation is completed successfully.

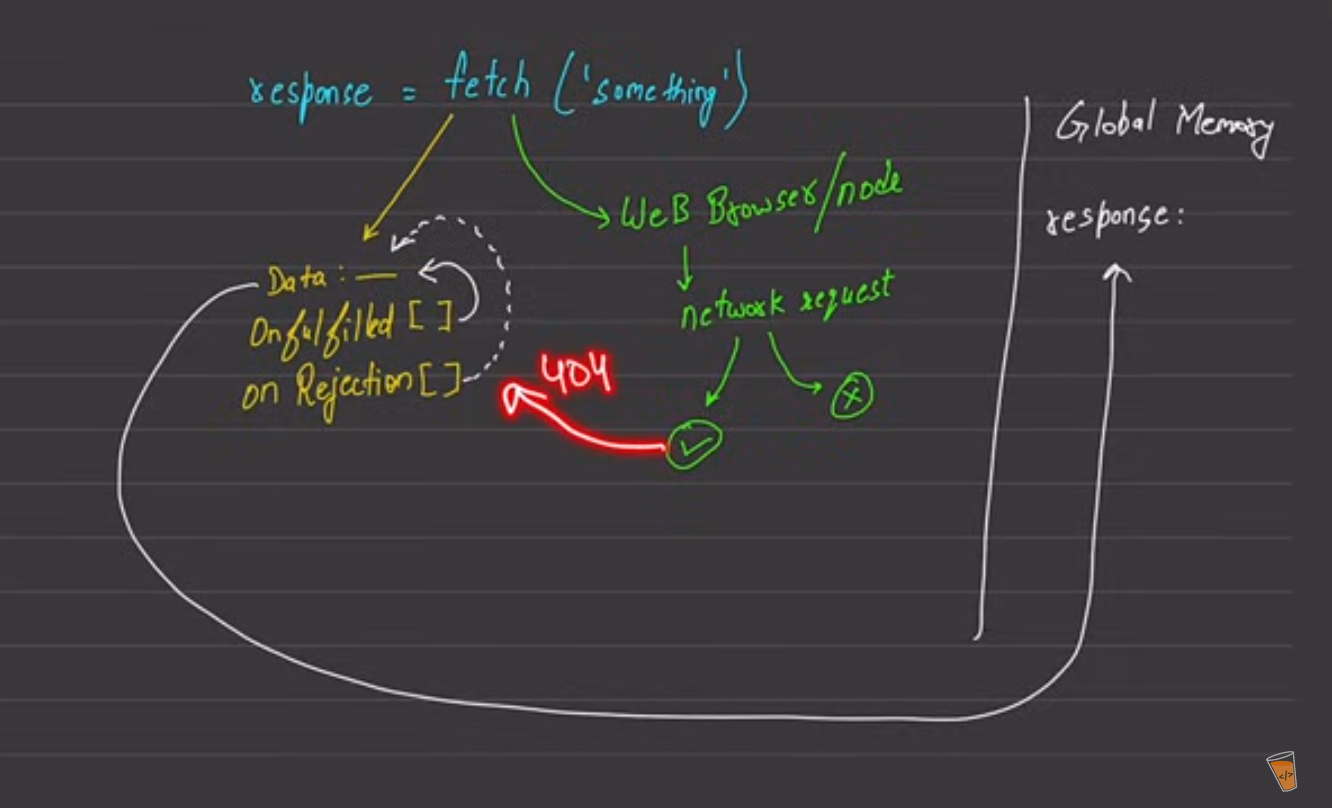
Rejected :- operation is failed.

**Async/await** is syntactic sugar built on top of Promises, introduced to make asynchronous code look and behave more like synchronous code, improving readability.

* async: Declares an asynchronous function that returns a Promise.
* await: Pauses the execution of the async function until the Promise is resolved or rejected

you requeested for a promise and he gave an error code of 404 then we will get this as an response it’s not an error.





This is insight of how the execution works in javascript.