JavaScript

Variables:

⇒ they are three types of variable datatypes
1) var
2) let
3) const

⇒ 'var' and 'let' datatypes are similar but 'const' is different because 'const' is only used for Constants values.

ex① var value1 = 10;
console.log(value1);
output: 10

var value1;
console.log(value1)
output: undefined

② let value = 10
console.log(value)
output: 10

let value;
console.log(value)
output: undefined.

⇒ ex: console.log(value) // output: undefined
// statements
"
"
var value = 100
console.log(value); // output: 100

⇒ in Javascript any where we can declare and and initialize the variables. within the script tags in the html page.

⇒ only by using 'var' keyword we can do above operation

⇒ but 'let' keyword doesn't support that functionality

⇒ using 'var' datatype we can declare more than one variable declarations.

ex: var i = 10 ⇒ console.log(i) // 10
var i = 20 ⇒ console.log(i) // 20
var i = 30
console.log(i) // 30

let :

⇒ let has the block level scope

ex: {
let name = 'vamsi';
console.log(name) // vamsi
}
console.log(name); // undefined

⇒ 'var' don't have any block level scope

Rules)

⇒ variables names are can use symbols like '_', $, alphabets, numbers. ex: vamsi_4

⇒ variables names should start with alphabets and underscore

ex: _vamsi;
let vamsi;

⇒ variables names are case sensitive
var name; // valid
var NAME; // valid

⇒ variables can write if as camel case
ex: var firstName;
var secondName;

Datatypes:

⇒ Types of datatypes ①

① primitive
* Number
* string
* Boolean

② Special
* undefined
* Null

③ Compusite
* Array
* object

⇒ the values we providing based on that value it binds automatically to that name.

=) to know the type of variable we use 'typeof()'

primitive:

```
var x = 87
console.log(x) // Number
var x = 87.01
console.log(x) // Number

var y = 'xyz' (or) var y = "xyz"
console.log(y) // String

var y = 'String "is"'
console.log(y) // string "is"

var b = true
console.log(b) // true
console.log(typeof(b)) // boolean
```

Special:

⇒ If we declare the variable but it will not initialized so it is called as undefined value

```
var d;
console.log(d) // undefined
console.log(typeof(d)) // undefined
```

=) undefined data type will take single value not group of values.

null

⇒ 'null' data type refers to the variable don't have any value it is not similar to undefined.

```
var b = null
console.log(b) // null
console.log(typeof(b)) // null
```

Composite:

⇒ to store multiple values in one single variable is called as arrays.

```
var num = [9, 10, 30, 20]
console.log(num) // [9, 10, 30, 20]

var num = [9, 'num', true, null, 'A']
console.log(num) // [9, 'num', true, null, 'A']
```

object: It is used to store key-value pairs like Hashmaps in java, dictionary in python

```
var ob = {'name': "vamsi", 'name2': "vamsi2"}
console.log(ob) // {'name': "vamsi", 'name2': "vamsi2"}
```

⇒ Key is only in string type it can be enclosed in '', "".

=) values should be any type.

=) Key and values should be seperated by colon and complete pair seperated by comma ',':

```
var students = [
{'name': "vamsi", 'age': 22}
{'name': "vamu", 'age': 21}
]
```

⇒ we can store multiple object elements in an array.

Template Literal

⇒ To print a complete string in different lines

```
var name = `hello world`
console.log(name) // hello world

var name = `hello \n world`
console.log(name) // hello
                  world
```

⇒ To print variable value in blw string

```
var x = "Java Script"
console.log("hello" + x + "world") // hello JavaScript world
```

⇒ To print variable value in multiple time in multiple lines we can enclose within back ticks (`)

```
var x = "Java Script"
console.log(`hello ${x} world`) // hello JavaScript world

var a = 10
var b = 20
console.log(`sum of ${a} and ${b} : ${a+b}`)
// sum of 10 and 20 : 30
```

⇒ instead using `\n` we can use like this

```
console.log(`hello
world`)          // hello
                 // world
```

⇒ we can provide operators for strings

```
console.log('hello'+'world') // helloworld
console.log("hello"+"world") // helloworld
```

Operators

Types of operators :-

1. Arithmetic operators
2. Comparision operators
3. Logical operators
4. Assignment operators
5. Type operators
6. Bitwise operators
7. Ternary operators

1) Arithmetic operators

- Add → 2+4 → 6
- Subtract → 2-4 → -2
- multiply → 2*4 → 8
- Divide → 2/4 → 0.5
- modulus → 2%2 → 1
- Negate → -(5) → -5

- increment (++) → pre / post
- decrement (--) → pre / post

} unary operator, it will take one value

(**) = power of

```
var a = "Java"
var b = "Script"
console.log(a+b) // JavaScript

var a = 30
var b = 20
console.log(a+b) // 50

var a = "Java"
var b = 9
console.log(a+b) // Java9

var a = 5
var b = 6
var c = "Java"
console.log(a+b+c) // 11Java
```

} this done based on assosicativity and providence

Comparision operators

→ compare only value it will neglect about type.

a==b	true if a and b are the same
a!=b	true if a and b are not the same
a>b	true if a is greater than b
a>=b	true if a is greater than or equal to b
a<b	true if a is less than b
a<=b	true if a is less than or equal to b
a===b	true if a is strictly equals to b if their type values are same.

Logical operators:

a && b AND true if both are true

a || b OR true if either or both are true

!a NOT true if a is false

ex:
```
var a=6
var b=true
console.log(a && b) // true

var a=false
var b=true
console.log(a && b) // false
```

Assignment operators

=	⇒	x=y	⇒ x=y
+=	⇒	x+=y	⇒ x=x+y
-=	⇒	x-=y	⇒ x=x-y
*=	⇒	x*=y	⇒ x=x*y
/=	⇒	x/=y	⇒ x=x/y
%=	⇒	x%=y	⇒ x=x%y
<<=	⇒	x<<=y	⇒ x=x<<y
>>=	⇒	x>>=y	⇒ x=x>>y
>>>=	⇒	x>>>=y	⇒ x=x>>>y
&=	⇒	x&=y	⇒ x=x&y
^=	⇒	x^=y	⇒ x=x^y
!=	⇒	x!=y	⇒ x=x!y
**=	⇒	x**=y	⇒ x=x**y

Bitwise operators

operators	description	example	Same as	Result	decimal
&	AND	5&1	0101 & 0001	0001	1
|	OR	5|1	0101 | 0001	0101	5
~	NOT	~5	~0101	1010	10
^	XOR	5^1	0101^0001	0100	4
<<	zero fill left shift	5<<1	0101<<1	1010	10
>>	signed right shift	5>>1	0101>>1	0010	2
>>>	zero fill right shift	5>>>1	0101>>>1	0010	2

Ternary operator:

condition ? expression_1 : expression_2;

ex: big = a>b ? a : b;

Conditional statements

1) if statement
2) if-else statement
3) if-else if-else statement
4) switch statement

⇒ if block treats '0' as false value and also empty string

* if(0) // false
* if("") // false
* if("string") // true
* if(9) // true

ex:-
```
var balance = 10000
var amount = prompt("enter amount to transfer")
if (amount < balance){
  alert("before deduction balance: " + balance)
  balance = balance - amount
  alert("after deduction successfully")
  alert("after deduction balance :" + balance)
}
else{
  alert("your balance is: " + balance + " enter amount less than your balance")
}
```

ex:-

```
var score = prompt("enter marks")
var grade;
if(score>=90){
  grade="A"
}
else if(score>=80){
  grade="B"
}
else if(score>=70){
  grade="C"
}
else if(score>=60){
  grade="D"
}
else{
  grade="F"
}
document.write("your marks:" + score + "grade:" + grade)
```

Switch

Syntax:-

```
switch(expression/value){
case choice1:
    run this code
    break;
case choice2:
    run this code instead
    break;
default:
    no case matches run this code
}
```

⇒ without break keyword next immediate statement will execute so to maintain code as correct mention break key mandatory. and as well as default at end of the all statements.

Type Conversion

1) implicit type Conversion
2) explicit type Conversion

1) implicit type Conversion

1) document.write(2+"4") // 24

⇒ javascript automatically converts integer to string

2) document.write(2-"4") // -2

⇒ javascript automatically converts string to integer

3) document.write(2-"s") // NaN

⇒ because string is an character it cannot convert into integer.

4) document.write(12+true) // 13 (or) document.write(12+false) // 12

because boolean value is true it means '1' it automatically converts it into integer. where as false is '0'

2) explicit type Conversion

⇒ this type conversion is only for integer, string, boolean

ex:- var x="6"

document.write(x+" : "+typeof(x)) // string

document.write("
")

var y = Number(x)

document.write(typeof(y)) // Number

⇒ from above example 'x' is already a string so that way it will convert into Number but instead of '6' there will be some text present in the sense it will not convert because it is already a string type of value.

⇒ document.write(parseInt("123java")) // 123

* it will convert number part only and returns the number
* also if convert from different data type

⇒ document.write(parseInt("I123java")) // NaN

the string starts with any char than it will return NaN

⇒ it will return the value as Number data type.

=> document.write(parseFloat("10.5Java")) // 10.5

* Conversion into string type

```
var x = 89
document.write(typeof(x)) // Number
x = String(x)
document.write("<br>")
document.write(typeof(x)) // String
```

'x' is shortcut

=> Strings will convert any thing into string type it automatically encloses within the double cots.

=> any data type elements can convert into string type

Conversion into boolean

=> Boolean() will returns 'true' when elements are non-zero.

```
var y = Boolean(6)
document.write(y) // true
var y = Boolean(0)
document.write(y) // false
var y = Boolean(-1)
document.write(y) // true
var y = Boolean("")
document.write(y) // false
var y = Boolean("Java")
document.write(y) // true
var y = Boolean(null)
document.write(y) // False
```

Loops in Javascript:-

types:- 1) for loop
2) while loop
3) do-while loop
4) for-in loop
5) for-of loop

1) for (we know the iterations)

```
for(initialization; condition; increment){
    // code to be executed
}
```

2) while (we don't know the iterations)

```
while(condition){
    //code to be executed
}
```

ex:-

```
var num = +prompt("Enter a number")
while(num!=0){
    document.write("entered number:"+num+"<br>")
    num = +prompt("entered a number")
}
```

output

entered number: 1
entered number: 2
entered number: 3
entered number: 0 (until 0 provided)

3) do-while

=> It will execute minimum one time
=> it will check condition in while executes statements in do.

```
do{
    // code to executed
}
while(condition);
```

ex:-
```
var count = 5
do {
    document.write("entered number: " + num + "<br>")
    num = prompt("enter a number")
    count = count - 1
}
while(count > 0);
```

for in (vs) for of

⇒ this is used for iterable data types like strings, objects, arrays

⇒ it means group of elements assigned to single variable

⇒ this loop points only index.

```
var fruits = ['mango', 'apple', 'banana', 'kiwi']

for(index in fruits){
    document.write(index + "<br>" + fruits[index])
}
```

output

0 : mango
1 : apple
2 : banana
3 : kiwi

* in object print of it will prints only keys

```
for(index of fruits){
    document.write(index + " : ")
}
```

output

mango : apple : banana : kiwi :

⇒ this loop will prints only element in an array

⇒ this both loops are used to iterate strings, array, objects etc.

⇒ ~~for in~~:

```
var students = {'100': "xyz", '101': "pqr"}
for(key in students)
    document.write(key + students[key])
```

output

100 : xyz
101 : pqr

ex:

```
var str = "indra"
for(i in str){
    document.write(i + " : " + str[i] + "<br>")
}
```

output

0 : I
1 : n
2 : d
3 : r
4 : a

⇒ for of loop only for arrays. and also for group of values.

Arrays

1) why arrays
2) how to declare arrays
3) how to access array elements
4) how to update/add array elements
5) method on arrays

how to declare arrays:-

1) `var movies = ['Bahubali', 'Acharya', 'RRR'];`
2) `var movies = new Array('Bahubali', 'Acharya');`

* `var movies = [];`
* `var movies = new Array();`

how to access array elements:-

⇒ we can access elements through their indexes

⇒ we try to access the elements in an array those are out of array index than it will show you an error called 'undefined'.

update / add of elements

① var movies = ['RRR', 'Bahubali']
console.log(movies)
movies[2] = "Maharshi"
console.log(movies)

② var movies = ['RRR', 'Bahubhali']
console.log(movies)
movies[100] = "maharshi"
console.log(movies)

output
'RRR', 'Bahubhali', undefined, undefined. . . . . 'maharshi'
0 1 2 3 100

⇒ except values contained indexes remaining all indexes are undefined if the values are ranges from start to end

⇒ arrays can store heterogenous elements of different data types
ex: var mo = [67, 0.2, 'A', "vamsi"]

methods on arrays:

* push()
* pop()
* unshift()
* shift()
* indexOf()
* slice()

push()

⇒ this function will decrease the undefined values in array
⇒ because it will add elements at the end of the array

var fruit = ['apple', 'banana', 'kiwi']
fruit.push('mango')
console.log(fruit) // apple, banana, kiwi, mango

* pop()

⇒ this function will removes last indexed element from an array and it returns the removed element

var fruit = ['apple', 'banana', 'kiwi']
fruit.pop()
console.log(fruit) // 'apple', 'banana'

* unshift()

⇒ this function will add element at the beginning of the array
⇒ it is quite opposite to the push().

var fruit = ['apple', 'banana']
fruit.unshift('mango')
console.log(fruit) // mango, apple, banana

* shift()

⇒ this function will removes an element at beginning of the array and it returns the removed element
⇒ it is quite opposite to the pop()

var fruit = ['app', 'banana']
fruit.shift()
console.log(fruit) // banana

* indexOf()

⇒ this function will returns the position of selected element

var fruit = ['apple', 'banana']
fruit.indexof('apple')
console.log(fruit.indexof('apple')) // 0

⇒ if the element is not present in the array than it will return '-1'.
⇒ if the array elements are duplicated than it will return first occurance of an an array.

* slice()

⇒ this function will returns the ↑elements from an array. some portion of

⇒ it will work from start index to end index

⇒ the elements are at start index are included but the elements are at ~~the~~ before end index are included.

```
var fruit=['apple', 'banana', 'mango']
console.log(fruit.slice(1,3)) // banana, mango
```

Syntax: slice(start index, end index)

Functions in JS

* what is function
* why functions
* how to create functions
* How to call functions
* what is function expression

what is function:

⇒ A block of code written to do a particular task

⇒ anything that enclosed between {}

ex:
```
{
    statements
    ;;
    ;;
}
```

⇒ these are used to reuse the code anywhere in the program.

how to create function / function definition:

⇒ every function should have same variable name rules

Syntax:
```
function functionName(parameters){
    body
    return value;
}
```

⇒ inputs are takes in the form of parameters

⇒ return & parameters are optional

ex:
```
function sum(num1, num2){
    var result = num1+num2
    return result;
}
```

call functions
```
functionName(parameters);
```

```
function sum(){
    body
}
```

```
sum(9);
```

⇒ there is no return statement than it will be return undefined

~~what~~ is

function expression:

⇒ if a function assigned to a variable is called function expression.

ex:
```
add = function sum(num1, num2){
    var result = num1+num2
    return result;
}
```

ex:
```
var add = function sum(num1, num2){
    var result = num1+num2
    return result;
}
var res = add(10,7)
document.write(res) // 17
```

⇒ if you call sum from above program it will show some error. "not defined".

Types of function

* Named function
* Anonymous function
* immediately invoked function expression (IIFE)
* Arrow function

Named function

=> it is an normal function

Syntax: function functionName(parameters){
body
return value;
}

ex: function sum(num1, num2){
var result = num1 + num2
return result;
}

Anonymous function

=> this function don't have the function name
=> the function start with function keyword.

function(parameters){
body
return value;
}

=> anonymous function used with function expression
=> without function expression it shows an error

ex: Sum = function(num1, num2){
body
return value;
}

Sum(value1, value2)

IIFE

=> when we want to execute a function immediately where they created, IIFE used.

Syntax: (functiondefinition)();

ex: (
function product(num1, num2){
var result = num1 * num2
return result;
}
)(6,8);

=> this function execute only once in a program
=> nobody will access this program (or) function because this is private (or) secure function.

ex2: var Sum = (function add(num1, num2){
var res = num1+num2
return res;
})(6,8);

Consle.log(Sum)// 14

=> without Sum variable we can not access above function

Arrow functions

=> this function is nothing but if don't have function keyword and function name.
=> this function only will have the open and closed parantheses followed by arrow symbol (=>) and not follows open and closed curly braces.

var variable name = (parameters) => {
body
return value;
}

ex: var product = (num1, num2) => {
var res = num1 + num2
return res;
}
product(5,8) // 13

Arrow function forms:

1) if only one statement in function body / without keyword return
var product = (num1, num2) => num1 * num2

2) if only one parameter
var cube = num1 => num1 * num1 * num1
(or)
var cube = (num1) => num1 * num1 * num1
(or)
var cube = (num1) => { return num1 * num1 * num1 }

calling cube(9);

3) if no argument
var greet = () => console.log("good morning");
(or)
var greet = _ => console.log("good morning")
or
~~var greet = => console.log("good morning")~~

Scope in Javascript:

Types of scopes:
1) global scope
2) local scope
3) Block level scope

local scope:
=> the variable declared inside the function those variables are called local scope variables.

ex: function test() {
var B = 67
console.log(B) // 67
}
console.log(B) // error

global scope:
=> the variables declared out side the block and as well as outside the functions they are called global variable
=> we can use those variables in any where in the program.
* we can use inside a function
* we can use inside a block

ex: function test() {
console.log(y)
}
var y = 10
test() // 10
{
console.log(y) // 10
}

ex: function test() {
var x = 10
console.log(x)
}
var x = 101
test() // 10

=> because function firstly checks for it's scope and then checks global scope if value is present for the variable inside function then blindly it will use that variable.
=> this will happens when both local and global scope variable names are same.

Block Scope

⇒ the variables declared inside the block can be accessible by only where in the program including functions also

ex:
```
function test(){
    console.log(x)
    console.log(y)
}
var y=70
{
    var x=100
}
console.log(x)
test() // 100
      // 100
      // 70
```

⇒ with the keyword 'var' we can access any where in the program. except in the function.

⇒ but we can not access variables with keywords like 'var', 'let', 'const' inside a function.

⇒ when we create variable with keyword 'let' inside a block can not use outside of the block. Similarly 'const'

default parameters & And Rest parameters

default parameters

⇒
```
function product(num1, num2 = 1){
    var result = num1*num2
    return result;
}
product(8);
```

⇒ when we call this function by passing same arguments. than it will execute normally

⇒ in case without any arguments we just calling a function than the default parameters where calculated as a result.

ex:
```
function product(num1, num2=1)
{
    console.log(num1, num2, num1*num2)
}
product(8) // 8, 1, 9
```

Rest parameters

⇒ we can make function to accept unspecified number of parameters

```
function product(num1=1, num=1, ...arr)
{
    var result = num1*num2
    return result;
}
```

ex:
```
function product(num1=5, num2=1, ...rest){
    var result = num1*num2
    for(num of rest){
        result = result*num
    }
    console.log(result)
}
product(8,7,8)
```

⇒ the javascript Rest parameter allows a function to accept an indefinite number of arguments as an array. it is represented by three dots (...) followed by the parameter name and must be the last parameter in the function enabling flexible and dynamic argument handling.

⇒ using loops we can access rest parameter array elements

⇒ by default in every function there is a 'arguments' object. all the arguments (or) parameters are stored in the arguments object.

ex: function product (num1=5, num2=1, ...rest){
console.log (arguments)
Console.log (arguments[0]) // 7
}
product (7,8,9,2,10)
present in argument object

foreach method

=> for every element present inside array the specified function will be executed.

Syntax: arr.forEach (function fnName(currentItem, index, arr){
// code here
});

=> currentItem : Required
=> index: optional
=> arr: optional

=> we can write any function inside this foreach.
ex: anonymous, arrow, named, ~~[illegible]~~ functions.

ex:
```
var Salaries = [50000, 45000, 55000, 60000]
Salaries.forEach ((Salary, index, arr)=>{
    console.log (Salary)
    Console.log (index)
    console.log(arr)
});
```

output
```
50000
0
arr
45000
1
arr
55000
2
arr
60000
3
arr
```

ex:
```
var Salaries = [50000, 45000, 55000, 60000]
console.log (Salaries)
Salaries.forEach ((salary, index) =>{
    let increment = salary/10
    Salaries[index] = salary + increment
});
console.log (Salaries)
```

ex: function definition seperated

```
var Salaries = [50000, 45000, 55000, 60000]
console.log (Salaries)
Salaries.forEach (increment)

function increment (Salary, index) {
    let increment = Salary/10
    Salaries[index] = Salary + increment
}
Console.log ( Salaries)
```

Set in Javascript

=> Set is a collection of unique values
=> it doesn't allow duplicate values
=> Set is a class in Javascript
=> first we have to create an object for the set() class

```
var ob1 = new Set();
ob1.add(9); // adding single element
console.log (ob1) // 9,
ob1.add(60).add(70).add(80) // add multiple elements
Console.log (ob1) // 9, 60, 70, 80
```

Convert array into set

```
var a1 = [50,70,60]
console.log(a1) // 50,70,60
var s1 = new Set(a1)
console.log(s1) // 50,70,60
```

convert String into set

```
var s1 = "hyderbad"
var s2 = new Set(s1)
console.log(s2) // 'h','y','d','e','r','b','a',
```

deletion

```
s2.delete('d')
console.log(s2) // hyerba
```

⇒ If the element present in the set it will delete the element and returns the boolean value

checking elements in Set

```
s2.has('d')
console.log(s2.has('d')) // true
```

length

```
s2.size()
console.log(s2) // 7
```

empty set

```
s2.clear()
console.log(s2) // 0
```

Iterating

1) forEach function
2) for-of loop

entries (iterating process)

```
let itr = s2.entries()
console.log(itr.next())
console.log(itr.next())
```

Objects in JavaScript

⇒ object is collection of elements in the form of properties and methods

⇒ property is a key - value pair.

ex:

```
var user1 = {
    name: 'xyz',
    email: xyz@gmail.com,
    mobile: 9e9000000
    login: function(){
        console.log("login success")
    }
};
```

ways to create object

1) object literal
2) using a new keyword with object constructor
3) using a new keyword with a constructor function
4) object.create() method
5) classes

using object Literal :

```
let movie1 = {}
```

ex:

```
let movie = {
    name: 'RRR',
    release: 2023,
    director: "Rajamouli"
}
```

accessing object values

⇒ we can access values by using key in 2 ways

1) obj["key"] → quotes are mandatory

ex: movie["name"] → valid

movie[name] → invalid

2) obj.key

ex: movie.name

add new properties to Object

⇒ we can add properties by using keys in 2 ways

1) obj["key"] = value

ex: movie["budget"] = "400 crores"

2) obj.key = value

ex: movie.budget = "400 crores"

⇒ we can update values in object in same way

2) using new keyword with object Constructor

```
let movie1 = new object();
movie1.name = "bahubali";
movie1.director = "Rajamouli";
```

3) using new operator with Constructor function

Step1: create Constructor function:

```
function user(name, age, place){
    this.name = name;
    this.age = age;
    this.place = place
}
```

Step2: create object with constructor function call:

```
let user1 = new user("abc", 25, "Hyderabad");
```

4) object.create method:

```
let movie2 = object.create(movie);

let movie3 = object.create(movie, {
        name: {value: "RRR"},
        music: {value: "Keeravani"}
})
```

⇒ this approach is used when object is already existed and we do modifications on that object properties by using this approach.

method creating in an object:

```
function user(n, a, p){
    this.name = n;
    this.age = a;
    this.place = p;
    this.login = function(){
        console.log("Hello" + " " + this.name + "loggged success")
    }
}

let user1 = new user("abc", 25, "Hyderabad")
user1.login()
```

output:

Hello abc loggged success

Iterating object

⇒ we can iterate object using for-in loop

ex:

```
for(key in user1){
    console.log(key, ":", user1[key])
}
```

=> we can access all the values at once from an object

```
console.log(Object.keys(user1))
```

=> 'keys()' it is function which gives all the keys in a object in the form of array

```
console.log(object.values(user1))
```

=> 'values()' it is function which gives all the values in a object in the form of array.

```
console.log(object.entries(user1))
```

=> 'entries()' is a function which gives all the key and value pairs in a object in the form of array.

Console methods in JS :-

- => console.log()
- => console.info()
- => console.warn()
- => console.error()
- => console.table()
- => console.clear()
- => console.assert()
- => console.count()
- => console.countreset()
- => console.time()
- => console.Timelog()
- => console.timeend()
- => console.group()
- => console.groupend()

DOM in JS

=> DOM : document object model

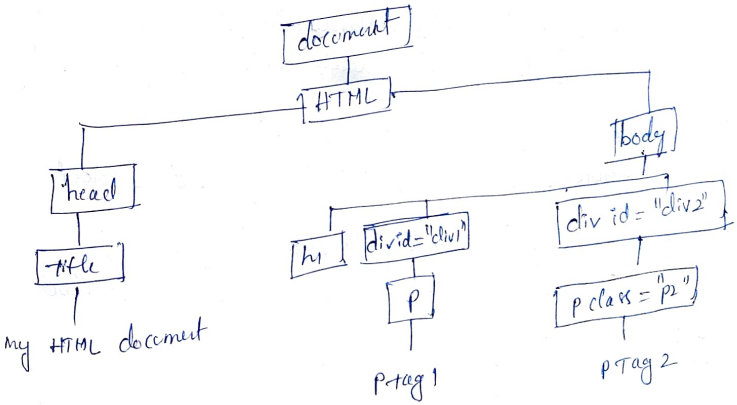
=> by using HTML Dom Javascript can access, change or remove any elements of HTML document and also can create new elements at any position.

=> when a webpage loaded, browser create DOM of webpage.

HTML document

```
<html>
    <head>
        <title> my HTML Document </title>
    </head>
    <body>
        <h1> Heading </h1>
        <div id="div1">
            <p> P Tag 1 </p>
        </div>
        <div id="div2">
            <p class="p2"> P Tag 2 </p>
        </div>
    </body>
</html>
```

document object model (DOM)



=> here is main object is 'document'

=> remaining all are also objects but not main object

⇒ Document object has * properties
* methods

⇒ using this document object properties & methods we can
* select HTML elements
* modify HTML elements
* Remove / delete HTML elements
* create HTML elements
* Add / Remove / change styles to HTML elements.

methods to select HTML elements :-

1) document.getElementById("idname")
returns element with the specified id

2) document.getElementByClassName("classname")
Returns list of all elements belogs to the specified class

3) document.getElementByTagName("tagname")
Returns list of all elements with the specified tag

4) document.querySelector(".class/#id/tagname")
returns the first object matching css style selector

5) document.querySelectorAll(".class/#id/tagname")
returns all objects matches the css style selector

ex:-
```
<body>
    <div class="special">
        <h2>this is heading</h2>
        <p>this is paragraph</p>
    </div>
    <div class="special">
        <h2>this is heading2</h2>
        <p>this is paragraph2</p>
    </div>
    <script>
    let x=document.querySelectorAll("h2")
    console.log(x)
</script>
```

DOM properties to select HTML elements:

⇒ document.body
⇒ document.head
⇒ document.title
⇒ document.anchors
⇒ document.forms
⇒ document.images
⇒ document.scripts

DOM manupulation

⇒ how to create HTML elements
⇒ how to set content to element
⇒ how to append a new element
⇒ how to insert an element before another element
⇒ how to remove an element
⇒ how to remove child element
⇒ how to replace an element

ex:-
```
<div id="second">
    <p>this is first paragraph</p>
    <p>this is second paragraph</p>
</div>
<script>
    var n=document.createElement('p')
    n.innerText="this is third paragraph"
    var parent=document.getElementById("second")
    parent.appendChild(n)
    console.log(parent.innerText)
    console.log(parent.innerHTML)
</script>
```

ex:- add element before an element

```
<Script>
    var n = document.createElement('p')
    n.innerText = "this is special paragraph"
    var parent = document.getElementById("second")
    var last = document.getElementById("last")
    parent.insertBefore(n, last)
</Script>
```

ex:- how to remove an element

```
<Script>
    var n = document.getElementById("last")
    last.remove()
</Script>
```

ex:- how to remove an childelement

```
<Script>
    var parent = document.getElementById("second")
    var last = document.getElementById("first")
    parent.removeChild(last)
</Script>
```

ex:- how to replace an element

```
<Script>
    var n = document.createElement('h1')
    n.innerText = "this is special heading"
    var last = document.getElementById("last")
    last.replaceChildren(n)
</Script>
```

add styles in JS

=> how to change style of element
=> how to get attribute value of an element
=> how to set/change attribute of an element

① how to change style of element

ex:-

```
<body>
    <div id="special">
        This is division one
    </div>
    <Script>
        var x = document.getElementById("special")
        x.style.color = "pink"
        x.style.border = "black 3px solid"
        console.log(x.style)
    </Script>
</body>
```

② how to get attribute value of an element:-

```
var x = document.getElementById("special")
console.log(x.getAttribute('class'))
```

③ how to set/change attribute of an element

```
var x = document.getElementById("special")
x.setAttribute('class', 'one')
console.log(x.getAttribute('class'))
```

ex:-

```
<div id="special">
    This is division one
</div> <br>
<img src="nature.png" alt="" id="img1">
<Script>
    var x = document.getElementById("img1")
    x.style.width = "100px"
    x.style.height = "100px"
    console.log(x.getAttribute('src'))
</Script>
```

In case change attribute use
↓
x.setAttribute('src', 'cake.png')

create an attribute

```
Name: <input type="text" id="fn">
<Script>
    var x = document.getElementById("fn")
    console.log(x.getAttribute('value'))
    x.setAttribute('value', "xyz")
</Script>
```

adding an class attribute to element

```
var x = document.getElementById("special")
x.classList.add("two")
```

=> this function will not remove the already existed class
- it will just add an new attribute to the element.

remove an class attribute in the element

```
x.classList.remove('one')
```

Model window

=> Alert - to show message to user
=> prompt - To take some input from user
=> confirm - to get confirmation from user

```
ex: alert("this is alert")
    prompt("enter your name")
```
=> user clicked on Cancel it will return null

```
prompt("enter your name", "abc")
```
=> it will take string format as an input

```
confirm("are you sure want to delete")
```
=> It will return true when user click on 'OK' otherwise false.

Events in Javascript

=> what is event & types of events in JS
=> what is event handler
=> ways to handle events in JS

Event

=> Event is nothing but an action
=> In webpage also what ever user do everything is a event

who generates events?

=> user → keypress, Scroll, focus, ...
=> System → load, error, abort, ...

Types of user generated events:

=> Browser Specific events → Scroll up/down resize browser...
=> Dom / web page specific events → click, hover, focus...

Commonly used events in Javascript

mouse events
1) click
2) double click
3) mouse over
4) mouse out
5) mouse move

Keyboard events
- Key down
- Key up
- Key press

focus events
- focus
- blur
- focusin
- focusout

form events
- Submit
- reset
- change

ways to handle events;

① ⇒ inline event handlers (using event attributes in HTML)

ex: <button onclick = "javascript code">

⇒ every event listner can be written with 'on'

ex: onclick
onmouseover

ex:

```
<body>
  <div id="special">
  Name: <input id="sp" type="text" /> <br><br>
  <button id="b1" onclick="change()">change bg</button>
  </div>
  <script>
    function change(){
      document.body.style.backgroundColor="green"
    }
  </script>
</body>
```

② using event properties in JavaScript

ex: let btn = document.getElementById('b1')
button.onclick = function name / anonymous function

ex:

```
let btn = document.getElementById('b1')
btn.onclick = change;
function change(){
  document.body.style.backgroundColor = "pink"
}
```

⇒ from above program the function change should not written as like 'change()' this. must and should write like this 'change;'

⇒ because if mention parantheses automatically browser runs entire code.

ex2:

```
let btn = document.getElementById('b1')
btn.onclick = function(){ document.body.style.backgr
                          oundColor = "grey");
```

③ using addEventListener() method in JS

ex: let btn = document.getElementById('b1')
button.addEventListener('eventname', function name/anonymous fn)

ex:

```
let btn = document.getElementById('b1')
btn.addEventListener('click', change)
                          ↳ avoid paranthesis
function change(){
  document.body.style.backgroundColor="pink"
}
```

ex2:

```
let btn = document.getElementById("b1")
btn.addEventListener('click', function(){
  document.body.style.backgroundColor="pink"
})
```

ex3:

```
let fn= document.getElementById("fullname")
fn.addEventListener('focus', change)
function change(){
  console.log(this)
  fn.style.backgroundColor="pink"
}
```

⇒ 'this' is like a object. we can know the event done on which element.

⇒ 'e' is an object we can put it within a paranthesis of a function.

Ex: let fn = document.getElementById("fullname")
fn.addEventListener('keyup', change)

function change(event){
console.log(event)
if(event.key == 'a'){
alert("a is not allowed")
}
fn.style.backgroundcolor = "pink"
}

Ex: Name: <input id='sp' type='text'/>

<button id='b1' onclick="change();">change bg</button>

<script>
function change(){
var letters = '0123456789ABCDEF';
var color = '#';
for(var i=0; i<6; i++){
var x = Math.floor(Math.random()*16);
color = color + letters[x]
}
document.body.style.backgroundcolor = color
}

var s = document.getElementById('sp')
s.onfocus = change2
s.onblur = change3

function change2(){
s.style.backgroundcolor = "grey"
}
function change3(){
s.style.backgroundcolor = "white"
}
</script>

Get timeout function / Set interval function

Set timeout function

⇒ <script>
setTimeout(function(name){ (function, argument)
alert("Hello"+name)
}, 3000, "xyz"); (time)
console.log("hello")
</script>

⇒ It will take three arguments, 1) function 2) time 3) any argument outs to the function

⇒ this function execution will pause for some time and then executes.

clearing timer interval

ex: var x = setTimeout(function(name){
alert("Hello"+name)
}, 3000, "xyz");
console.log(x)
clearInterval(x)
console.log("hello")

⇒ 'x' is an variable it holds the setTimeout function id.
⇒ after function there will be an another function called 'clearInterval(x)' it will removes setTime function execution

ex: var x = setTimeout(test, 3000, "xyz");
console.log(x)
console.log("hello")
function test(name){
alert("Hello"+name)
}

Set interval function

⇒ for very time interval provided in the function definition the function should calls.
⇒ ex: the function 'test' should called for every 3000 milise secs.

```
var counter=1
var x = setInterval (test, 3000, "nag");
console.log(x)
console.log("hello")
function test (name){
    alert ("called!" + counter)
    counter++
    if (counter>5){
        clearInterval(x)
    }
}
```

Map()

⇒ It is used to modify arrays data and to create new array
⇒ first the map function will take values from existed array and take values from it and modify insert into new array.
⇒ at last the map function returns the modified value to the new array.

```
var birthyear = [2015, 2016, 2001, 2002]
var ages = birthyear.map(function (ele, index, arr){
    let age = 2021 - ele
    return age;
})
console.log(ages)
```

element = required
index = optional
arr = optional

⇒ without return statement function will return undefined.
⇒ for each method do not create new array only map method will create new array

filter()

⇒ filter method used to get values from array
⇒ the filtered values which are filtered based on some condition it will get.

ex:-
```
var nums = [6, 9, 89, 60, 10]
nums.filter(function(ele))
```

⇒ this method will checks entire array for condition satisfy and return that element to the new array

ex:
```
var nums = [6, 9, 89, 60, 10]
var even = nums.filter(function(element)){
    return element%2==0
})
console.log(even)
```

⇒ inside function will takes two arguments like map function element is mandatory.

Reduce()

⇒ this function have atleast two parameters as default.
one is accumulator, element or, index, arr.
⇒ one the function returns the value that value passed as an input to the next iteration accumulator value.
⇒ accumulator default value defined at the beginning

Syntax:
```
array.reduce(function(accumulator, element){}, default value)
```

⇒ here accumulator is different type variable name.

⇒ arrname.reduce(function, accumulator default value)

⇒ function can take 4 arguments

⇒ function (acc value., ele, index, array)

⇒ arr.reduce.(function (acc ele, index, arr){},8)

⇒ this method returns a single value

Ex1:-
```
var arr =[7,9, 8, 9, 10, 78]
arr. reduce (function (acc, ele){
    if (acc> ele){
        return acc
    }
    else{
      return ele
    }
}, arr[0])
```

Ex2:-
```
var arr =[7,90, 8]
var sum=arr. reduce (function (acc, ele){
    return acc+ele
},0)
```

find, any, every methods in Javascript:-
```
let places =["hyderbad", vijayawada", "Delhi", "vizag"]
var places = places.find (function(ele){
    return ele.startwith("v")
})
console.log(place)
```

⇒ this function returns first occurance with starting letter, and it doesn't checks for other occurances. only returns first occurance.

⇒ if it finds the selection returns the world otherwise returns 'undefined'.

⇒ this function similar to the finder but there is one difference instead of returning the caught value it will returns the boolean value.
```
let places =["hyderbad", "vijayawada", "Delhi", "vizag"]
var place = places.Some(function (ele){
    return ele.startwith ("v")
})
console.log (place)
```

every():-
```
marks = [7, 89, 87, 88, 54, 64]
var pass = marks. every(function (mark){
    return mark>35
})
console.log (pass) //false
```

⇒ this method returns 'true' when entire array will have the true values it means given condition should be satisfied.

⇒ one the element met the false element then method stops it's iterating through another elements.

Spread operator

⇒ this will take input as a iterable object like
1. string
2. array
3. object

⇒ Spread operator denoted as "(...arrayname)"

ex1:-
```
var arr =[78,89,90,67]
console.log(arr) // 78,89,90,67
Console.log(...arr) // 78,89,90,67
```

ex2:-
```
var arr2 =[2,4,arr]
console.log(arr2) // 2, 4, Array of first example
~~console.log~~
```

ex3:-
```
var arr2=[2,4,...arr]
console.log(arr2) // 2,4,78,89,90,67
```

ex4:-
```
var str="Java"
console.log(...str) // J a v a
console.log(str) // Java
```

ex5:-
```
var arr1=[1,2,3]
var arr2=[4,5,6]
var arr3=[...arr1,...arr2]
console.log(arr3) // [1,2,3,4,5,6]
~~console~~ var arr3=[...arr1,6,7,...arr2,67,87]
```

Call back function in javascript:-

ex:-
```
function test(a){
    a()
}
test(function(){
    console.log("call back function")
})
```

output
call back function

ex5
```
function first(test){
    test()
    console.log("this is first function")
}
function second(){
    console.log("this is second function")
}
first(second)
```

output
this is second function
this is first function

ex:-
```
function add(a,b){
    console.log(a+b)
}
function product(a,b){
    console.log(a*b)
}
function calculator(val1,val2,operation){
    operation(val1,val2)
}
calculator(4,5,product)
calculator(4,5,add)
```

output
20
9

⇒ if a function passcle a function wants to execute then put it in the parantheses like function(),

Higher order functions in javascript:-

⇒ higher order function is a function that receives another function as argement or return another function or both

Example of receiving another function as argument:

```
function first(fn){
    fn()
}
function Second(){
    console.log("this is Second function")
}
first(Second)
```

ex:

```
var arr = [56, 8, 7, 10, 11]
function Square(num){
    return num*num
}
function cube(num){
    return num*num*num
}
function anyoperation(arr, operation){
    let result = []
    for(ele of arr) result.push(operation(ele))
    return result
}
let x = anyoperation(arr, cube)
console.log(x)
```

output

arr will be cubed

Returning a function:

```
function first(){
    return Second
}
function Second(){
    console.log("this is Second function")
}
first()()
```

output

this is Second function

ex2:

```
function first(){
    return () => { console.log("this arr fn") }
}
function Second(){
    console.log("this is Second fn")
}
first()()
```

output

this arr fn

=> we can able to return any type of functions like arrow, anonymous, named.

Closure in JS

=> accessing variables of a function even its execution completes or out of its scope

ex:

```
function outer(){
    var x = 10;
    function inner(){
        var y = 100
        console.log(x)
    }
    return inner
}
var res = outer()
res() // 10
```

ex:

```
<input type="button" onclick="updatecount()" value="click">
<p id="count"> clicked :0</p>
<script>
    function updateCountSecure(){
        let count = 0;
        function inner(){
            count = count+1
            let p = document.getElementById("count")
            p.innerText = `clicked : ${count}`
        }
        return inner
```

```
let updatedCount = updateCountSecond()
console.log(updatedCount);
```

This Keyword in JavaScript

⇒ value of this keyword outside a function / object
⇒ value of this keyword inside a regular function
⇒ value of this keyword inside a method
⇒ value of this keyword inside event handlers
⇒ value of this keyword inside a function in strict mode

① console.log(this) ⇒ window object
if we use this keyword outside of function or object

ex:
```
var a = 10
console.log(this)
function add() {
}
console.log(window.a)
console.log(a)
console.log(this.a)
var obj = {
    name: "xyz"
}
console.log(this.x)
```
⇒ it will represent window object

②
```
var a = 10
// console.log(this)
function add() {
    console.log(this.a)
}
```
⇒ it will represent window object

③
```
var movie = {
    name: "RRR",
    getname: function() {
        console.log(this)
        console.log(this.name) // RRR
    }
}
movie.getName()
```
⇒ in this case 'this' keyword Represents movies object not window object.

④
```
<input type="button" value="click" onclick="console.log(this)" />
```
~~onclick~~ onclick = "this.style.background = 'green'; this.style.color = 'white'"

⇒ this will also represents the input element

⑤
```
"use strict"
var x = 10; // No error
    or
x = 10 ; // error
```

⇒
```
"use strict"
function() {
    console.log(this) // undefined
}
```
⇒ with 'use strict' key word the function returns undefined
⇒ without 'use strict' key word the function returns window object.

call, apply, bind in JS

ex:

```
var salary = 20000;
function getSalary(){
    console.log(this.salary)
    console.log(window.salary)
    console.log(salary)
}
employee = {
    salary: 30000
}
getSalary.call(employee) // 30000
```

⇒ Actually this will points to the window when it is used in a function. But now in this case we can change the this key word pointer to the any other object by using some methods like call, apply, bind.

Syntax: call(objectname) Similar to the apply, bind

ex:

```
var salary = 20000
function getSalary(a,b){
    console.log(this.salary)
    console.log(a,b)
}
employee = {
    salary: 30000
}
getSalary.call(employee, 67, 690)
```

output

```
30000
67
690 //
```

apply()

⇒ apply() is similar to the call() but call function takes comma seperated arguments where as apply() takes as in array format.

```
getSalary.apply(employee, [67, 78]) // 30000
                                    67
                                    78
```

bind()

```
var f1 = getSalary.bind(employee, 67, 690)
f1()
```

output

```
30000
67
690
```

By using user user defined object

```
var myobject = {
    salary: 40000,
    getSalary: function(a,b){
        console.log(this.salary)
        console.log(a,b)
    }
}
myobject.getSalary(65,89)
employee1 = {
    salary: 30000
}
myobject.getSalary.call(employee1, 67, 690) // 30000
                                            67
                                            690
```

Optional chaining

```
let employee = {
    id = 151;
    email: 'xyz@gmail.com',
    personalInfo: {
        name: 'xyz'
        mobile: 999999999
        address: {
            line1: 'Near Big Bazar',
            line2: 'Ameerpet Road',
            pincode: 500073,
            city : 'Hyderbad',
            state : 'Telangana'
        }
    }
}
console.log(employee.personal?.mobile) // undefined
console.log("hello") // hello
```

⇒ '?' this symbol represents as optional chaining symbol

⇒ when an property of an object is not present in the object than at execution time it returns an error without mentioning the optional chaining symbol.

⇒ it will directly stops execution at first console statement so to overcome this problem using '?' is good practice it will avoids the executions and execute the next console.

Date object

```
var d1 = new Date()
console.log(d1) // todays date
```

```
let d2 = new Date(20000)
console.log(d2) // Jan 10 1970 05:30:20 => 05:30:20
```

⇒ one day after

```
let d2 = new Date(24*60*60*1000)
console.log(d2) // Jan 02 1970 05:30:00
```

⇒ one day before

```
let d2 = new Date(-24*60*60*1000)
console.log(d2) // Dec 31 1969 05:30:00
```

```
⇒ let d4 = new Date('2020-2-08')
  console.log(d4)

⇒ let d5 = new Date('08-2-2020')
  console.log(d5)

⇒ let d6 = new Date(2019,10,28)
  console.log(d6) // Nov 28
```

⇒ because the months are starting from 0 -> Jan 11 dec

```
⇒ let d6 = new Date(2019,10,28,5,30,10)
  console.log(d6) // Nov 28 2019 5:30:10

⇒ let d6 = new Date(2019,1)
  console.log(d6) // feb 2019
```

Math object

⇒ it is predefined object

String method in String object :-

- => charAt()
- => indexOf()
- => lastIndex()
- => toLowerCase()
- => toUpperCase()
- => concat()
- => trim()
- => split()
- => startsWith
- => endsWith
- => slice()
- => substring()
- => substr()
- => includes()
- => charCodeAt()
- => repeat()
- => replace()

=> var str = new String('Java')
console.log(str) // Java

* **indexOf()**
=> the character present in a String than it returns the that character index otherwise -1

* **lastIndexOf()**
=> returns last occurance index.

* **startsWith()**
=> returns boolean value

* **endsWith()**
=> returns boolean value

* **Slice()**
=> returns a substring in a String

* **charCodeAt()**
=> returns the Ascii code

* **Repeat()**
=> returns the repeated String

Window (Open and Close)

window.open(URL, name/target, Specifications)

- blank
- self
- parent
- top

Specifications:
- => width
- => height
- => left
- => Top
- => Resizable
- => Scrollable

Ex:-

```
<div>
  <h1> this is main window </h1>
  <button onclick = "openNewWindow()">
      open a window
  </button>
</div>
<script>
  var ref;
  function openNewWindow(){
    ref = window.open("", "", "width=100px,height=200px,left=100px,top=200px")
  }

  function closeNewWindow(){
    ref.close()
  }
```

=> moveTo()
=> moveBy()
=> resizeTo()
=> resizeBy()

① moveTo()

```
function MoveWindow(){
    ref.moveTo(500,100)
}
```

② moveBy()

```
function moveWindow(){
    ref.moveBy(200,100)
}
```

③ resizeTo()

```
function resizeToWindow(){
    ref.resizeTo(800,400)
}
```

④ resizeBy()

```
function resizeByWindow(){
    ref.resizeBy(300,200)
}
```

AJAX – Asynchronous Javascript and XML

=) Ajax helps in fetching data asynchronously from a remote web server

=) Data loaded by AJAX call is done asynchronously without page reload

=) the server will send response which contains data that we have requested

=) data can be of any format like JSON, XML.

=) Initially by servers used to send data in XML format

=) In Javascript we have to use XMLHttpRequest object to make AJAX call to exchange data from webserver

*) AJAX request can be send to server in with 3steps

1) Create XMLHttpRequest object

ex) let xhr = new XMLHttpRequest()

2) Create request with that object open() method

Syntax: open(method, url, async, username, password)

ex:

```
<body>
    <input type="submit" class="button" id="special" value="Get data">
    <div id="data"></div>

<script>
    let btn = document.getElementById("special")
    let datadiv = document.getElementById("data")
    btn.addEventListener('click', loadData)

    function loadData(){
        let xhr = new XMLHttpRequest()
        xhr.open('GET', 'test.txt', true)
        xhr.send()
        xhr.onprogress = function(){
            datadiv.innerText = "Loading....."
        }
        xhr.onload = function(){
            datadiv.innerText = xhr.responseText;
        }
    }
</script>
```

method	description
new XMLHttpRequest()	Creates a new XMLHttpRequest object
open(method, url, async, user, psw)	Specifies the request method: the request type GET or post url: the file location async: true (asynchronous) or false user: optional user name psw: optional password
send()	Sends the request to the server used for GET requests
send(string)	Sends the requests to the server used for POST requests
setRequestHeader()	Adds a label/value pair to the header to be sent

abort() ⟶ cancels the current request

getAllResponseHeaders() ⟶ Returns header information

getResponseHeader() ⟶ Return specific header information

property	description
onload	⟶ when the request is complete and the response is fully downloaded.
onreadystatechange	⟶ defines a function to be called when the readyState property changes
onprogress	⟶ triggers periodically while the response is being downloaded, reports how much has been downloaded
readyState	⟶ Holds the status of the XMLHttpRequest 0: request not initialized 1: Server connection established 2: request received 3: processing request 4: request finished and response is ready.
responseText	⟶ Returns the response data as a string
responseXML	⟶ Returns the response data as XML data
status	⟶ Returns the status-number of a request 200: "ok" 403: "forbidden" 404: "Not found"
statusText	⟶ Returns the status-text ("ok" or "Not found")

Call Back Hell

⇒ the order of execution of functions statements one by one as per the function calls.

ex)
```
function f(){
  statement1
}
function f1(){
  statement2
}
f()
f1()
```

output
statement2
statement1

> To avoid this use callback hell

ex)
```
function step1(next_task){
  setTimeout(()=>{
    console.log("step1 done")
    next_task()
  },2000)
}
function step2(){
  console.log("step2 done")
}
step1(step2)
```

output
step1 done
step2 done

ex:
```
function step1(){
  setTimeout(()=>{
    console.log("step1 done")
  },2000)setTimeout(()=>{
    console.log("step2 done")
    setTimeout(()=>{
      console.log("step3 done")
      setTimeout(()=>{
        console.log("step4 done")
        setTimeout(()=>{
          console.log("step5 done")
        },1000)
      },1000)
    },2000)
  },2000)
```

⇒ this is callback hell

promises in JavaScript

⇒ promises is an object in JS

⇒ To overcome the situation of callbackhell use this promises

⇒ promise is an object

```
let p = new promise()
```

⇒ promise object takes a callback function

```
let p = new promise(function(){})
```

⇒ This object returns some data either success data or error information.

```
let p = new promise(function(resolve, reject){
})
```

(resolve, reject — callback functions)

⇒ states of promise 1) pending
2) fulfilled
3) rejected

ex:

```
let p = new Promise{
    function(success, error){
        let datacome = true
        if(datacame){
            success([1,2,3])
        }
        else
            error("promises not done")
    }
}
console.log(p)
p.then(function(data){ ⟹ promise is success
    console.log(data)
})
p.catch(function(errordata){ ⟹ promise is reject
    console.log(errordata)
})
```

or

```
let btn = document.getElementById("special")
let datadiv = document.getElementById("data")

btn.onclick = function(){
    let p = new promise(function(resolve, reject){
        let xhr = new XMLHttpRequest()
        xhr.open("GET", "test.txt", true)
        xhr.send()
        xhr.onload = function(){
            if(xhr.status == 200){
                resolve(xhr.responseText)
            }
            else
                reject("No data found")
        }
    })
    p.then(function(data){
        console.log(data)
    })
    p.catch(function(errordata){
        console.log(errordata)
    })
}
```

Fetch API

⇒ fetch is a method of browser window object which helps to make AJAX call

Syntax: fetch(url,[options])

options: optional parameters like method, headers...

⇒ fetch method returns a promise

⇒ So to use the data from server we have to use then() method of promise return by fetch method

```
let promise = fetch("yourSomeUrl")
promise.then(function(response){
    console.log(response)
```

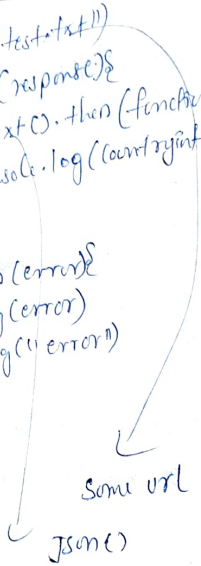
Ex:-

```
let p = fetch("test.txt")
p.then(function(response){
    response.text().then(function(countryinfo){
        console.log(countryinfo)

})

p.catch(function(error){
    console.log(error)
    console.log("error")

})
```



Second way of usage

```
fetch("test.txt")
.then(function(response){
    response.text().then(function(countryinfo){
        console.log(countryinfo)

})
```

Ex:-

```
fetch('URL',{
method: 'POST',
body: JSON.stringify({
        title:'xyz',
        body: 'test',
        userId: 2,

}),
headers:{
    'content-type': 'application/json; charset=UTF-8',
},
})
.then((response)=> response.json())
.then((json)=> console.log(json));
fetch("URL")
.then((response)=> response.json())
.then((json)=> console.log(json));
```

Async await in Javascript

⇒ Async and Await in Javascript is used to simplify handling asynchronous operations using promises. By enabling asynchronous code to appear synchronous, they enhance code readability and make it easier to manage complex asynchronous flows.

```
async function fetchData(){
    const response= await fetch("url");
    const data = await response.json();
    consonle.log(data);

}
fetchData();
```

ex:-

```
async function test(){
    x= await 5+7;
    console.log("inside function")
    return x

}
let res = test()
console.log("outside function")
```

output

```
outside function
inside function
12
```

Ex:-

```
easyn function test(){
    let res= await fetch("test.txt")
    let finalres = await res.text()
    console.log(finalres)

}
test()
```

output

output in the file.

⇒ 'await' key word is used before the returning promise functions or any other expressions.

⇒ 'async' key word is used in the before the function starting.

Error handling

ex:
```
console.log("line1")
console.log(line2)      error statement
console.log("line3")
```

⇒ from above program second statement is error statement so it stops execution on that line.

⇒ To avoid this situation we can use error handling like (try, catch blocks).

```
try
{
  console.log("line1")
  console.log(line2)
}
catch(error){
   console.log(error)
}
```

output

line1

⇒ the statements after the error statement are not executable.

⇒ 'name, message' these are properties of error parameter

⇒ we can known what type of error, and what is the error.

throw

⇒ predefined action by JS

ex:
```
let x=10
console.log("line1")
if(x<15){
   throw "x must be greater than 15"
   console.log("please provide number>15")
}                                  ↓ not executable
console.log("hello");
        ↓ not executable
```

output

line1

throws error

⇒ it will stops the execution after the error statement

prototype in JS

⇒ Javascript is a prototype-based, automatically adds a prototype property to functions upon creation. This prototype object allows attaching methods and properties, facilitating inheritance. for all objects created from the function.

ex:
```
function example(){
}
console.log(example)
```

output

then will be an prototype.

⇒ protype is an object inside a function

⇒ ~~__proto__ is an also object inside a object~~

⇒ __proto__ is an property in an prototype object

ex:
```
var obj = {}
console.log(obj)
```

output

prototype: object

__proto__: object

⇒ what are the functions or properties create under the prototype of a function will be shared among all the objects

```
function employee(name){
    this.name = name;
}
employee.prototype.id = "101";
var obj = new employee("Akhil")
var obj2 = new employee("vanssi")
console.log(obj.id) // 101
console.log(obj2.id) // 101
```

modification

```
employee.prototype.id = 102 // 102
obj.__proto__.id = 103 // 103
```

Destructuring in JS

```
let arr = [100,200,300,400]
let person = {
    name : "xyz",
    age : 36,
    address : {
        city : "Bangalore",
        State : "hyderabad"
    }
}
let a = arr[0]
let b = arr[1]
let c = arr[2]
```

} this approch is too timetaking and also memory consumption so to avoid this destructuring came in to the picture.

or

```
let [a,b,c] = arr
console.log(a,b,c) // 100, 200, 300

let [a,,c] = arr
console.log(a,c) // 100,300

let [a,...arr1] = arr
console.log(a, arr1) // 100, [200,300,400]
```

an object

⇒ accessing an property in object

```
console.log(person.name) // xyz
console.log(person.address.city) // Bangalore
```

To avoid this

```
let {name, age} = person
console.log(name, age) // xyz, 36

let {name, address:{city}} = person
console.log(name, city) // xyz, Bangalore

let {nam, age} = person
console.log(nam) // undefined
```

⇒ if in case (name) value is not there in the object so use

```
let {name="pqr"} = person => we can add new property
console.log(name) // pqr
```

⇒ if in case we want call "name" property with different name or alias name we can

```
let {name:nam} = person
console.log(nam) // xyz
console.log(name) // xyz
```

Strict mode

=> this mode will apply at beginning of the Script tag

=> we can use "use strict" any where in the program inside a function, outside a function, entire program

=> By using this mode we can overcome user mistakes

ex:
```
<Script>
    "use strict"
     x=10
     console.log(x) // error of type mentioning

     var x=10
     console.log(x) // 10
     function test(){
       "use strict"
        x=10  => var x=10
        console.log(x) // error
     }
     test()
</Script>
```

Regular Expressions in JS

=> Regular expression is a sequence of characters, helps to create a search pattern

ways to create regular expressions:

1) RegExp literal notation

Syntax: /pattern/[flags] → optional

ex: /hello/

/^[0-9]{10}$/

2) using 'RegExp' Constructor function

Syntax: new RegExp('pattern', [flags]) → optional

=> regular expressions are case sensitive

```
let re = /hello/

=> let res = re.test("Hello Javascript")
   console.log(res) // false

=> let res = re.test("hello Javascript")
   console.log(res) // true

   let re = /hello/i  -> flag  both cases (upper, lower)

   let res = re.test("Hello Javascript")
   console.log(res) // true
```

=> there are two methods in the RegExp 1) test()
2) exec()

By using object:
```
let reg = new RegExp("hello")
let res = re.test("thellowoh")
console.log(res) // True
```

=> this function will takes variables also

ex:
```
let x = 'hello'
let re = new RegExp(`${x}`)
let res = re.test("thellowoh")
console.log(res) // True
```

exec():

=> it will find the index position of a expression in given string and it will return it

```
let re = /hello/
let res = re.exec("hello javascript hello")
console.log(res) // array form
console.log(res.index) // 0
```

=> if it is not there it returns null

'g' => this is a flag and it is used to find occurances of a [illegible] and once found then leave that string and

expression	description	example
[.....]	Any one character blw the brackets.	[a&9] ⇒ a or & or 9
[^....]	Any one character not blw the brackets	[^a&9] ⇒ everything except a, & and 9
[0-9]	Any number from 0-9	1/1/2 ..... /9
[a-z]	Any lower case alphabet	a/b/c/..... /z
[A-Z]	Any upper case alphabet	A/B/..... /Z
[a-zA-Z]	Any lower case or upper case	a/b/c/..... /z, A/B/... /Z

Quantifiers: for specifying the number of occurrences of a character

Quantifier	description	example
*	⇒ match zero or more times	Bm* ⇒ "B", "Bm", "Bmm", "Bmmm", and so on..
+	⇒ match one or more times	j+ ⇒ "j", "jj", "jjj"...
?	⇒ match zero or one time	bs? ⇒ "b" or "bs", but not "bss"...
${n}	⇒ match exactly n times	jab${5} ⇒ "jabbbbb"
${n,}	⇒ match at least n times	mb${3,} ⇒ "mbbb", "mbbbb"...
${n,m}	⇒ match from n to m times	br${3,5}a ⇒ "brrra" or "brrrra" or "brrrrra"

⇒ '.' dot is one meta character it will allow all the characters except '\n'.

or Period

```
let reg = /he.llo/
console.log(reg.test("javascript  helnllo")) //false
console.log(reg.test("    "  hee llo")) True
console.log(reg.test("    "  hello")) false
```

⇒ '\d' backward slash 'd' tells about a digit ranges from 0-9

```
var reg = /\d/
console.log(reg.test("javascript hello")) //false
console.log(reg.test("javaScript 4hello")) // True
```

⇒ '\D' backward slash capital 'D' tell about except digits atleast one character will present in given string.

```
let reg = /\D/
console.log(reg.test("95")) // true
console.log(reg.test("9")) // false
```

⇒ '\w' backward slash 'w' tells about the given string should contains [0-9, a-z, A-Z, _). except these remaining all are not acceptable.

```
let reg = /\w/
console.log(reg.test("-")) //false
console.log(reg.test("OSA_")) True
```

⇒ "\W" backward slash 'W' tells about the given string should contains other than small 'w' characters.

```
let reg = /\W/
console.log(reg.test("-")) // true
console.log(reg.test("a-sA")) //false
```

⇒ "\."

```
let reg = /\.s/
console.log(reg.test("6s")) // false
console.log(reg.test("6.s")) //true
```

⇒ "\s" for space