Java - Script

→ It is introduced in the year 1995, Brendeon Eich, he is worked in Netscape company.

→ Java-script is a interpreted Programming language.

→ It is used to connect static-page into Dynamic Page.

→ Java-script
↓
live-script
↓
mocha
↓
ECMA script

These are the names for Java-script.

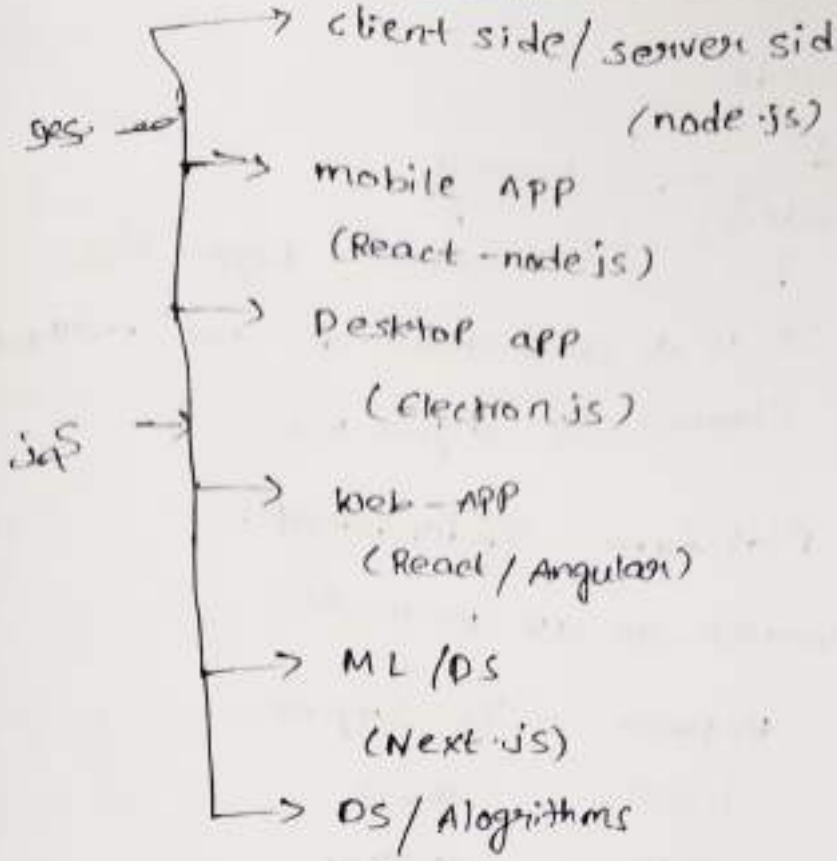
→ ECMA (European computer Manufacturing Association). It is a ~~com.~~ organization formed by Netscape company.

→ It is used to take care of Java-script.

→ Every year On June new updates are released.

→ At ES-6 version onwards Java-script start booming in the year 2015.

→ It is a client side and server-side (node-js)



features / characteristics of java-script :-

→ object-based and oops oriented Programming language. → It objects have built-in methods.

→ It is a Interpreted PL.

→ just-in-time-compiler.
→ In-buildt methods.

→ synchronous in nature / single-threated →
Asynchronous → Promise → fetching API.
(only-one call stack)

→ case-sensitive (keywords) lowercase

→ client-side & server-side.

client-side :- Any PL that works on Browser.

server-side :- Any PL that runs / works outside a Program.

For server-side

↓

node.js → library

→ Discovered by Rayan Dhal

↓

It is a combination of c++ method and chrome js-engine v.8.

→ It is a Plat form - Independent.

(It works on all browsers)

Browser → JS-Engine

chrome → v8.

IE → chekra

safari → JS-core

MFF → spider-monkey.

→ Java-script can written in two ways :-

1. Internal JS.
2. External JS.

1. Internal JS way :-

It should be written in a same-file inside a script tag. It is a Paired tag.

ex:-
```
<html>
<head></head>
<body>
- html tags -
<script>
- JS-code -
</script>
</body>
</html>
```

2. External JS :-

step 1 :- link .js file to .html file create a .js file.

step 2 :- link .js to .html file with the help of script tag.

<script src = "Path" , </script>

write in Bottom of body

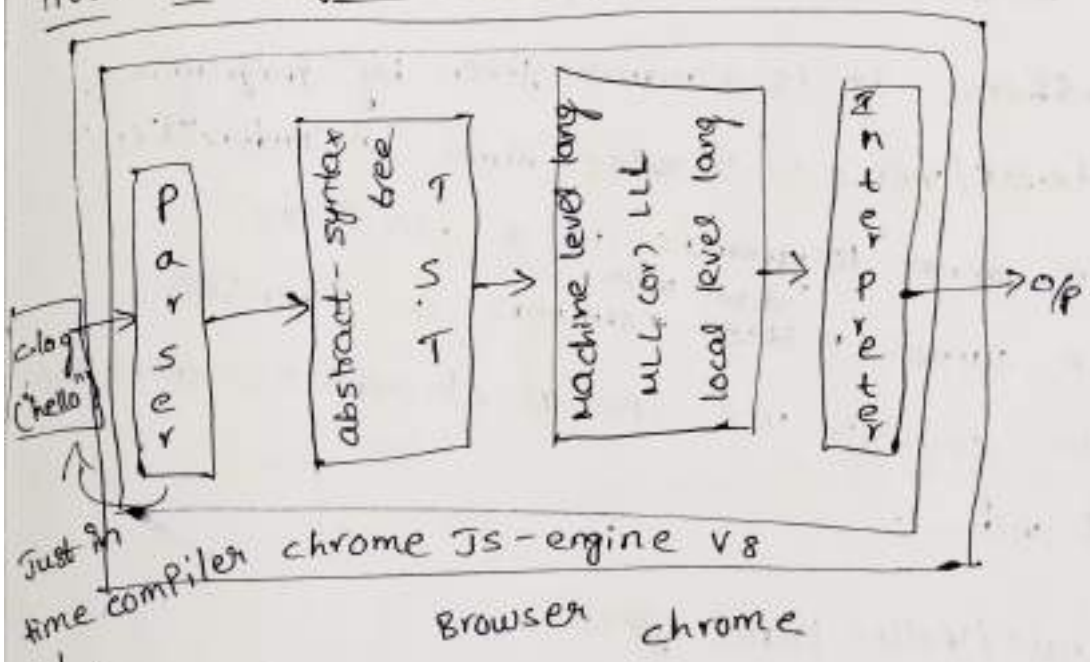
, async and defer are the keywords.

, It is used to write script tag anywhere in a Html file.

⇒ when Async keywords both html and js file works Parallelly.

⇒ Defer keyword not works like that.

How JS engine works :-



↳ checks the syntax

4 errors in JS :-

Reference, syntax, range, type.

Printing statements in J.S :-

console.log("JS");

console.warn(" i am warning message") ⚠ symbol

console.error(" i am error message") ⊗ error } when we get the flag it will gen

console.table(obj)

console.group( )

console.dir (window)

AST

Abstract syntax tree

→ code will arranged in a tree kind of structure.

GEC (Global Execution Context)

variable state → function/execution state

Tokens in JS :-

→ Tokens are smallest-unit in Programming language.

→ 3 types are there :-

1. keywords :- It is a Pre-defined reserved words.

eg's :- var, let, const, console, function, return, if, else, while, do, break, continue etc.

2. Identifiers :- It is a name given by Programmer.

~~3. Literals/value :- 3 rules~~ Rules of Indentifiers :-

1. we cannot keywords as a identifers.
2. we cannot ~~a use~~ start with numbers as identifiers.
3. we cannot use special characters otherthan $ and _ .

3. Literals/value (Data types)

→ It is a value used in Programming language.

2 types :- 1. Primitive-Data type / Immutable Data-type

2. Non-Primitive-Data type / Mutable data-type

1. Primitive Data type :-

→ values cannot be changed.

* Number
* string
* null
* un-defined
* boolean
* Bigint
* symbol

2. Non-Primitive Data type :

→ values can be changed easily.

* object
* class
* function
* Array
* set
* map

1) Number data type :-

→ Any Positive values, negative values, decimal will be treated has number-Data-type.

→ 64 bits binary storage (memory).

→ Range :- $-2^{53} - 1$ to $+2^{53} - 1$

Eg :- clg (type of 10)
console.log

clg (type of -10)

clg (type of 0.5)

2) string data type :-

→ Any sequence of character which is enclosed with ' ', " " and ` ` (backticks)

Eg :- clg (type of 'Hello')

clg (type of "Hello")

clg (type of `Hello`)

Ex1:- let a = 'I am a developer'

clg(a)

Ex2:- let b = "I'm a developer"

clg(b)

Backtick(``)

→ It is used to Print string in a multiple lines.

→ string which is generated using backticks is a templet string (used for interpolution ${ })

3) Un-defined Data type :-

→ will declare a variable but will not assign any value for it.

Eg :- var a;

clg(a) //UD → un-defined

Ex:- let a;

clg(a) //UD

4) Null Data type :-

→ It is a keyword. It is nothing but empty.

→ clg (type of null) // object

Eg:- let b = null;

clg (b) //null

5) boolean :-

→ It is a keyword. It will accept two values.

1. true (1)

2. false (0)

Big int :-

→ It is used to cross the range of number-data-type.

eg :- clg (type of 10n) // big-int

* 1n
  10n
  100n
  1000n

symbol :-

→ It will return's unique value..

eg :- let a = symbol ('Hello')

clg (a) // symbol

let b = symbol ('Hello')

clg (b)

clg (a===b) // false

→ strictly equal to

Diw == & and ===

ex :-

s == s // true

s == 's' // true

s === s // true

s === 's' // false

== & ===

== → equal to

=== → strictly equal to

Java script keywords :-

⇒ The following are reserved words in javascript.

⇒ They cannot be used as javascript variables, functions, methods, loop labels, or any object names.

abstract	else	instance of	switch
boolean	enum	interface	synchronized
break	export	long	this
byte	extends	native	throw
case	false	new	throws
catch	final	nkull	transient
char	finally	Package	true
class	float	Private	try
const	for	Protected	typeof
continue	function	Public	var
debugg er	goto	return	void
default	if	short	volatile
delete	impleme nts	static	while
do	import	super	with
double	in		

Type of operator :-

It is a unary operator. (It accepts only one operator). Type of operator will tells to the Programmer which type of datatype Programmer is using. Ex :- Not, Type operator

D/w Primitive & non-Primitive data type:-

Primitive DT

```
var a = 10;
clg(a)//10
```

0x1 — object referring 0x1 → 10

```
var b = a
clg(b)//10
var a = 'Hello'
clg(a)//Hello
clg(a)//Hello
clg(b)// 10
var b = a
clg(b)//Hello
```

'Hello' — 0x2

→ values cannot changed.

Non-Primitive DT

```
var obj = {
    name : 'abhi',
    id : 1
}
clg(obj)// { }
var obj1 = obj
clg(obj1)// { }
obj1. id = 10;
clg(obj)
{name : 'abhi', id = 10}
clg(obj1)
{name : 'abhi', id = 10}
```

0x1 — object referring 0x1 → { name: 'abhi', id = 1 }

→ values changed easily.

Type casting:-

→ converting of one-type of data type into another type is called type-casting.

→ Two types are there:-

1. Implict type-casting:-

→ converting of one-type of data-type to another data-type by a js-engine is called ITC.

Eg:- clg(5 + 5)//10

clg(5 + '5')//55 → number into string

clg(5-'5') // 0 → string into number

clg(5+'a') // 5a

clg(5-'a') // NaN → Not a number

clg(type of NaN) / number

⇒ In js undefined
0
null
NaN
false
defaultly consider as false.

Explict-type casting :-

→ converting of one-type of data type to another type by a Programmer is called ETC.

→ In-built methods

Number( )

Boollean( )

String( )

Eg:- clg(5+Number('5')) // 10

clg(String(5)+5) // 55

clg(Boolean(0)) // false

Variables in js :-

→ Block of memory used to store values stored in the form of bits.

→ Dynamic in nature (no-need of mentioning Data-type).

→ Scope's :-

* Global-scope.
* local-scope / script-scope.
* Block-scope { } → block, 3

3-types of variables

1. var
2. let

Diff var, let, const variables:

	var	let	const

1. Scope	Global-scope (Bec it Present inside global window object)
	eg:- var a = 10;
	clg(10) //10
	clg(window)
	a:10;
		local-scope (Bec it not-Present inside global window obj)	local-Scope (Bec it not-Present inside global window obj)
			eg:- const c=30;
			clg(c) //30
			clg(window)

		eg:- ~~var~~ let b=20;
		clg(b) //20
		clg(window)	

2. Declaration	var can be declared.
	Eg:- var a;
	clg(a) //UD
	un-defined
		let can be declared.
		eg:- let a;
		clg(a) //UD
			const cannot be declared.
			eg:- const c;
			clg(c) //syntax error.
			(Bec in const declaration & initialization happen at same time)

3. Declaration and initialization	var can be declared & initialized.
	eg:- var a=10;
	clg(a) //10
		let can be declared & initialized
		eg:- let b=1;
		clg(b) //1
			const can be declared & initialized.
			eg:- const c=11;
			clg(c) //11

4. Re-initialization	var can be Re-initialized.
	eg:- var a=10;
	clg(a) //10
	a='Hello'
	clg(a) //Hello
		let can be Re-initialized.
		eg:- let b=20;
		clg(b) //20
		b='Hi'
		clg(b) // Hi
			const can't be Re-initialized.
			eg:- const c=30;
			clg(c) //30
			c='Hello'
			clg(c) //syntax error

5. Re-Declaration and Re-initialization	var can be re-declared & re-initialized.	let can't be re-declared & Re-initialized.	const can't be re-declared & re-initialized
	Eg:- var a=10;	Eg:- let b=10;	Eg:- const c=10;
	clg (a)//10	clg(b)//10	clg(c)//10
	var a = 'Hello'	let b= 100	let const c=100
	clg(a)// Hello	clg(b)// reference error	clg(c)// reference error

<u>Use • strict</u> :- whenever we use usestrict we should declare identifiers with the help of variable names, if we don't use variable name it leads to reference error. It should be written inside single quote (' ') (or) double quote (" ") at the top of the Page. usestrict is a keyword.

<u>Hoisting in js</u> :-

⇒ JS engine allows Programmer to declare members (variables) before variable declaration.

→ variables
→ functions } Hoisting.

<u>GEC</u> (<u>Global</u> <u>Execution</u> <u>context</u>) :-

GEC
↓ variable State
↓ execution / functional - state.

⇒ Two steps :-

1. step 1 :- memory will be allowated in that one default ~~memory~~ ~~vari~~. value will be stored

(un-defined).

def:- Instructions are executed from top to bottom.

ex:- 1. var a = 10;
2. clg (a) // 10
3. clg (b)
4. var b = 20; // UD

UD
ox1

UD
ox2

ex:- 1. let a = 10;
2. clg (a) // 10
3. clg (d)
4. let d = 20; // RE } Hoisting

reference error

~~Rec~~ In let reference error will come bec TDZ.

Temporal Dead zone (TDZ):-
The time b/w variable initialization and variable declaration is called TDZ.

Hoisting:

var	let	const
var can be hoisted will get o/p as a undefined.	let can be hoisted ~~with~~ it ~~get o/p~~ leads to reference error, bec of TDZ.	const can be hoisted but it leads to reference error, bec of TDZ.
Ex:- clg (a)	Ex:- clg (b)	Ex:- clg (c)
~~var (a) // 10~~	let b = 20 // RE	const c = 30 // RE
var a = 10; // UD		

Operators in JS :-

It is a Pre-defined symbols used to Perform specific task.

1. Arithmetic operator :- +, -, ÷, *, /, ++, --

2. Assignment :- :=, +=, -=, *=, ÷=, /=

3. Relational :- <, <=, >, >=, ==, ===
   - == → equal to
   - === → strictly equal to

4. logical :- AND, OR, NOT
   (&&) (||) !

5. conditional operator / :- (condition) ? true statement : false statement
   Ternary operator

functions :-

* function - Declaration - statement / General functions.
* function - Expression.
* function - Programming.
  - → HOF (Higher-order-function)
  - → CBF (call-back function)
* Arrow-function.
* Nested - function
  - → lexical - scope / scope-chain
  - → Closer
* Immediate invoking function Expression (IIFE)

treated as objects

1. function - Declaration statement / General functions

> Block of code / set of instructions used to perform specific task.

identifiers

syntax:- function function-name (Parameters 1, Par 2 ...)

```
{
  - code -
}
```

value / expression (data-types)

function-name (arguments 1, arg 2 ...)

Advantage:- code - Re-usuability.

return-keyword :- It will stop's execution of function and controller will be given to caller.

Eg1:- function demo( )

```
{
clg('Hello-world')
}
clg(demo) // Body of the fn
clg( ) // Hello-world.
```

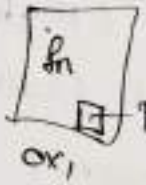
Eg2:- function add (a, b)

```
{
let c = a+b;
return c
clg ('Hello')
}
add (10, 10) // 20
add (100, 100) // 200
add (100-100, 200) // 200
```

clg add(20, 20) // 40
clg (add)

Eg3:- demo( ) // Hello world
clg (demo) // fn of the body } It is hoisting

function demo( )
{
clg ('Hello-world')
}
clg(demo) // fn of the body
demo( ) // Hello world



Prototype

or,

It is a copy of an function / object.

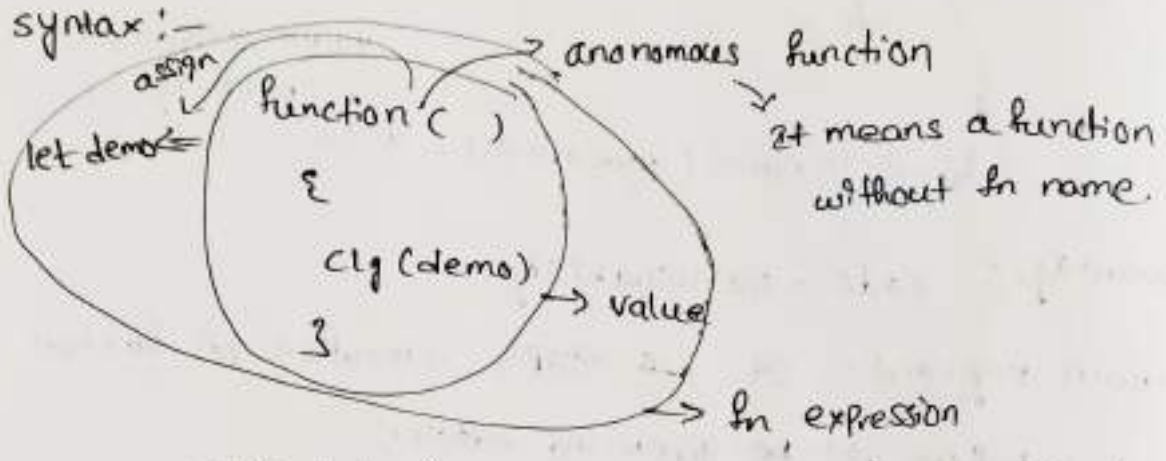
2. function - Expression :-

⇒ Assigning function has a value to one variable is called fn-expression.

⇒ It is used to perform specific task.

Note :- Here fn-exp can't be hoisted.

syntax :-



Ex :- add1(300, 300) // reference error

```
function (m,n)
{
let o = m+n
return o
}
add1(10,10) //20
add1 (100,100) //200
```

3. function - Programming :-

⇒ It is used to perform Generic - task (multiple task).

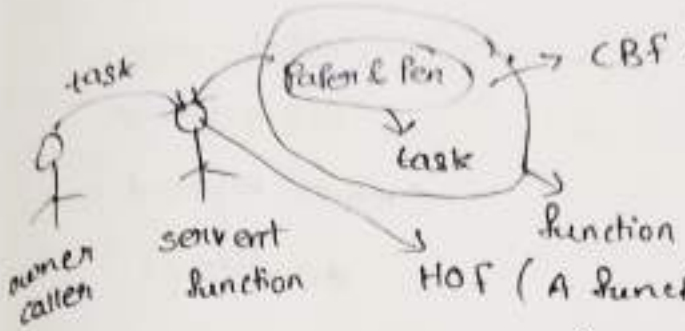
⇒ It have 2 types :-

1. HOF (Higher-order-function)
2. CBF (call-back-function)

⇒ It doesn't any syntax. It can be created by using function declaration statement's, function expression and Arrow - function.

function Programming (Generic-task)

(HOF, CBF)



CBF (A function which is passed has an argument to HOF)

HOF (A function which will accepts function has an arguments.)

(ex)

```
function operation (a, b, task)
{
let c = task (a, b)
return c
}
```

HOF CBF

```
operation(10, 10, function(a, b)
{
return a+b
}
operation (100, 100, function(a,b)
{
return a-b
}
```

features of ES-6 version:-

1. let and const.
2. Arrow - function.
3. Promise.
4. Asyne and await
5. De-structuring.
6. Rest-Parameter and spread operator.

4. Arrow-function:-

→ It is introduced in ES-6 (2015).

→ It is used to reduce syntax (to ~~du~~ reduce no's of lines in a code).

Two types of return

Implict - return Arrow function
(No need of using return keyword)

Explict - return Arrow function
(return keyword is mandatory).

Eg:-

```
function demo()
{
clg('Hello')
return 'Hello'
}
demo()
```

Arrow fn ⟹ fat arrow

```
let demo = ( ) => clg('Hello')
```

Parameters

fat arrow

Anonomous fn

value

fn - expression

Implict - return - Arrow - fn

```
demo() // Hello
```

Eg2:-

```
clg(demo1()) // RE
let demo1 = ( ) => { return 'Hello' }
```

fat arrow

value

fn - expression

Explict - return - Arrow - function

```
clg(demo1()) // Hello
```

Disadvantages of Arrow function:

1. we cannot create constructor function using arrow function because arrow function will not support for new keyword
2. we cannot use arrow function in call, apply, bind method.
3. Arrow function doesn't have its own Prototype.

This keyword :-

→ It is a keyword.
→ It holds memory-address of window object. we access members present inside window object.

eg:- clg (window)
clg (this)

```
var a = 10;
function demo( )
{
var a = 'Hello'
clg (a) // Hello
clg (this.a)
}
demo( ) // Hello
        // 10
```

```
let b = 20;
function demo1( )
{
let b = 100;
clg (b)
clg (this.b)
}
demo1( ) // 100
         // undefined
```

Nested function :-

A function inside another function is called Nested function.

In java script we will achive lexical scope (or) ~~Effec~~ scope chain & closer in Nested functions.

Ex:- function Parent ( )

```
{
  function child ( )
  {
    clg ('I am child-function')
  }
  return child ( )
}
```

Parent local scope

Parent( ) // I am child-function

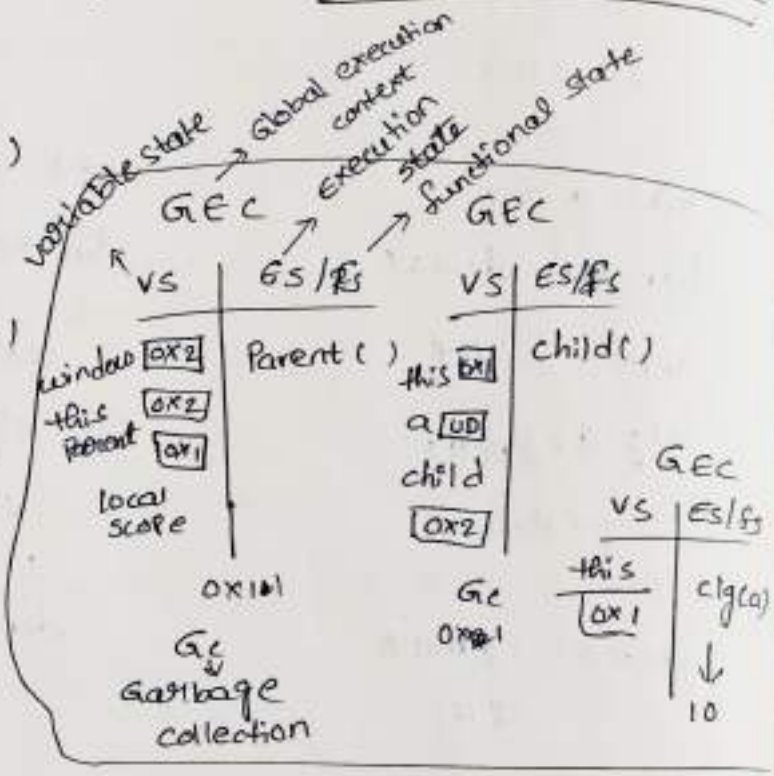
13 The ability of JS engine search for the variable in local scope if it is there then it search in global scope then it is called as lexical scope (or) scope chain.

Ex:- function Parent ( )

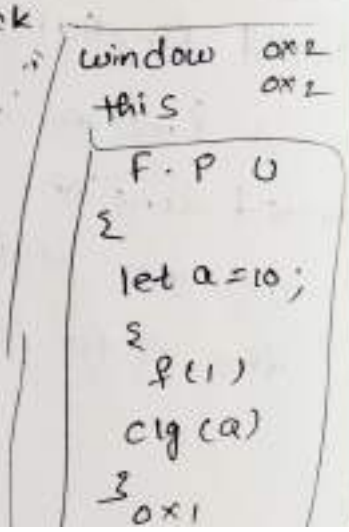
```
{
  let a=10;
  function child ( )
  {
    clg (a)
  }
  return child ( )
}
Parent ( ) // 10
```

global scope

local scope



call-stack



Clouser :-

The binding of child function lexical scope to Parent function is called clouser.

child fn $\xrightarrow{\text{reference address}}$ clouser (copy of Parent fn)

Disadvantage of clouser :-

→ Memory wastage. Bec how many times calls the child fn that many times copy of Parent fn is present in heap area.

Ex:-

```
function Parent ()
{
let a=10;
function child()
{
clg(a)
}
return child
}
Parent()()  //10
        P()()  //10
        P()()  //10
```

child

Note :- In nested function we achieve inheritance. (where child fn access the Properties of Parent fn).

Ex :-

```
function Parent ()
{
  let a=10;
  function child ()
  {
    clg(a)
    let b=20;
```

```
function child1 ()
{
clg (b)
}
return child1
}
return child
}
Parent ()()()
```

Ex2:-

```
function Parent ( )
{
function child1 ( )
{
clg ('I am child1')
chid 1.1 ( )
return child1.1 ( )
}
function child2 ( )
{
clg ('I am child2')
}
return [child1, child2]
let res = Parent ( )
res [0] ( ) ( )
res [1] ( )
```

~~secan~~ ~~first~~ Scenario1:-

```
function Parent ( )
{
let a = 10;
function child ( )
{
clg (a)
let b = 20;
function child1 ( )
{
clg (b)
}
return child1
}
return child
}
Parent ()()()
```

Scenario 2:-

```
function Parent ( )
{
function child1 ( )
{
clg ('I am child1')
}
function child2 ( )
{
clg ('I am child2')
}
return [child1, child2]
}
let res = Parent ( )
res [0] ( )
res [1] ( )
```

IIFE (Immediate, Involving function expression)

A function is called immediately as known as function object is created.

syntax:-

(↓) ( )
expression
function

eg) (function ( )
{
clg ('Hello')
} ( )

(( ) =>
{
return 'Hello'
} ( )

```
(function Demo(a, b)
  {
  let c = a+b;
    return c
  } (10, 10)
```

Advantage:-

→ It is used to Prevent global Pollute name space (In is we should avoide using global scope, because it Pollutes JS-engine).

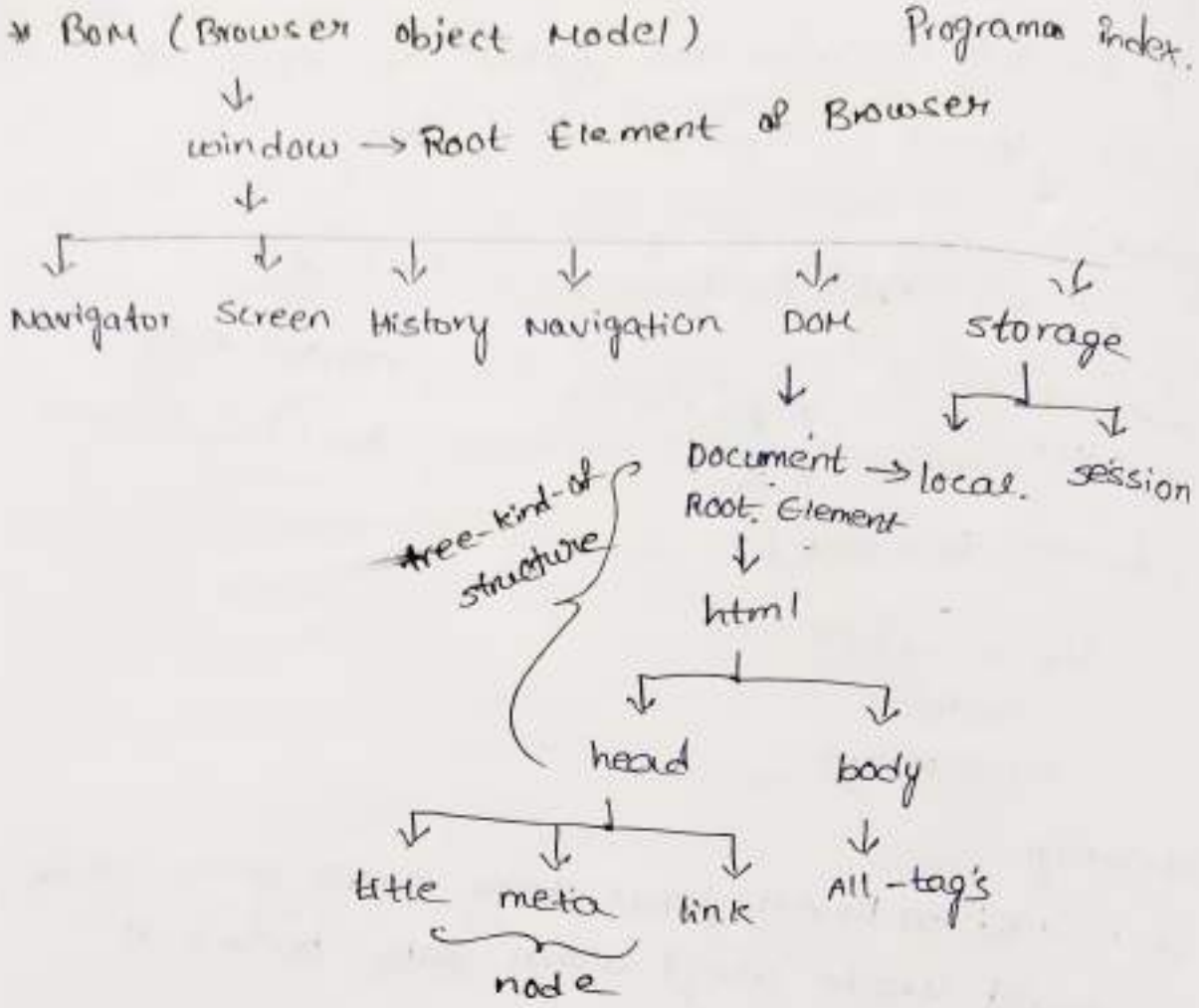
→ IIFE will be used in jquery.

eg:-

```
( function ( )
  {
  var a = 'vamsu'
  function demo( )
  {
  clg ("my name is ${a}")
  }
  demo( )
  let b = 'sin'
  function demo1 ( )
  {
  clg ('my name is ${b}')
  }
  demo1( )
  } ( )
```

local scope

* DOM (document object Model) API → Application Programa Index.
* BOM (Browser object Model)



DOM (Document object Model) :-

⇒ It is a "object".

⇒ DOM is an Browser API Provided by "window".

⇒ Dom is having a Root Element that is "document".

⇒ life-span (until the Page Reloads).

⇒ DOM is used camelcase.

Uses :-

⇒ Using DOM we can create, update, Delete element in a DOMtree.

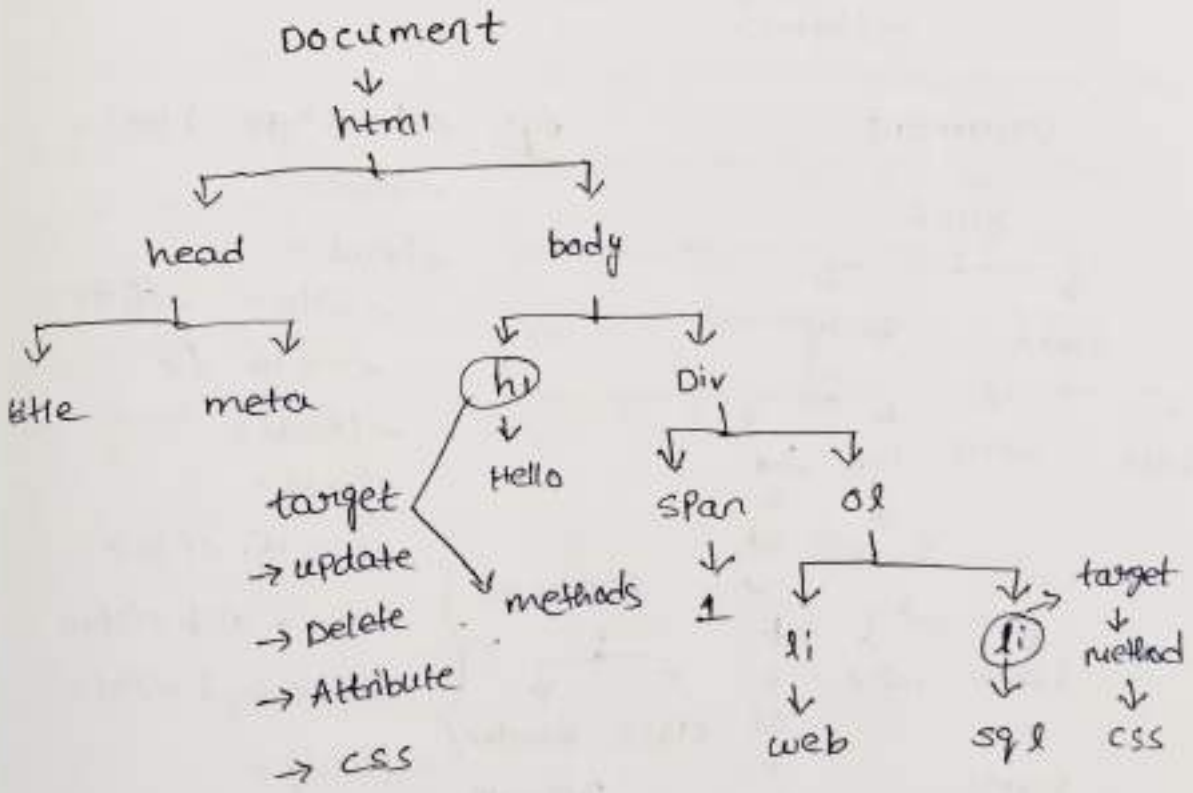
⇒ Using DOM we can add attributes and stage in DOMtree.

How to Convert html str into DOM tree :-

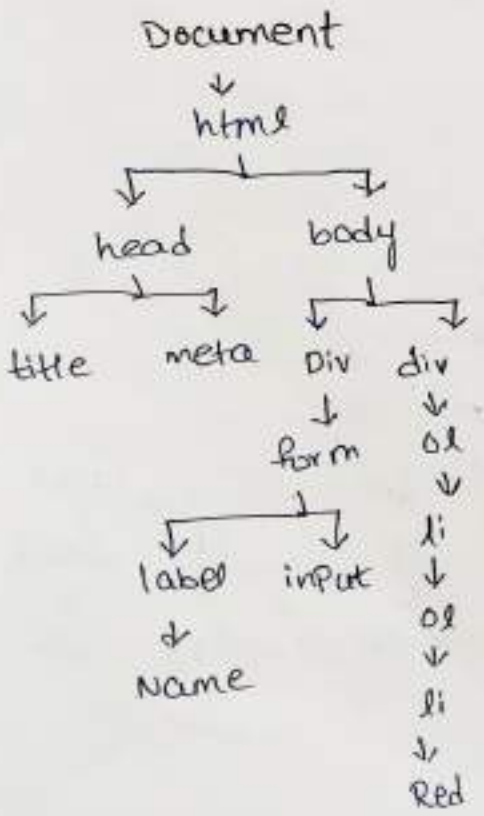
↓

structure

```
<!doctype html>
<html>
<head>
  <title> </title>
  <meta />
</head>
  <body>
    <h1> Hello </h1>
    <Div>
      <span> 1 </span>
      <ol>
        <li> web </li>
        <li> sql </li>
      </ol>
    </Div>
  </body>
</html>
```



```
<!doctype html>
<html>
  <head>
      <title>  </title>
     <meta />
     </head>
       <body>
        <form>
        <label> Name </ label >
        <input type = 'text' />
        </form>
        </div>
           <div>
              <ol>
                 <li>
                    <ol>
                       <li>Red </li>
                    </ol>
                 </li>
              </ol>
           </div>
        </body>
     </html>
```

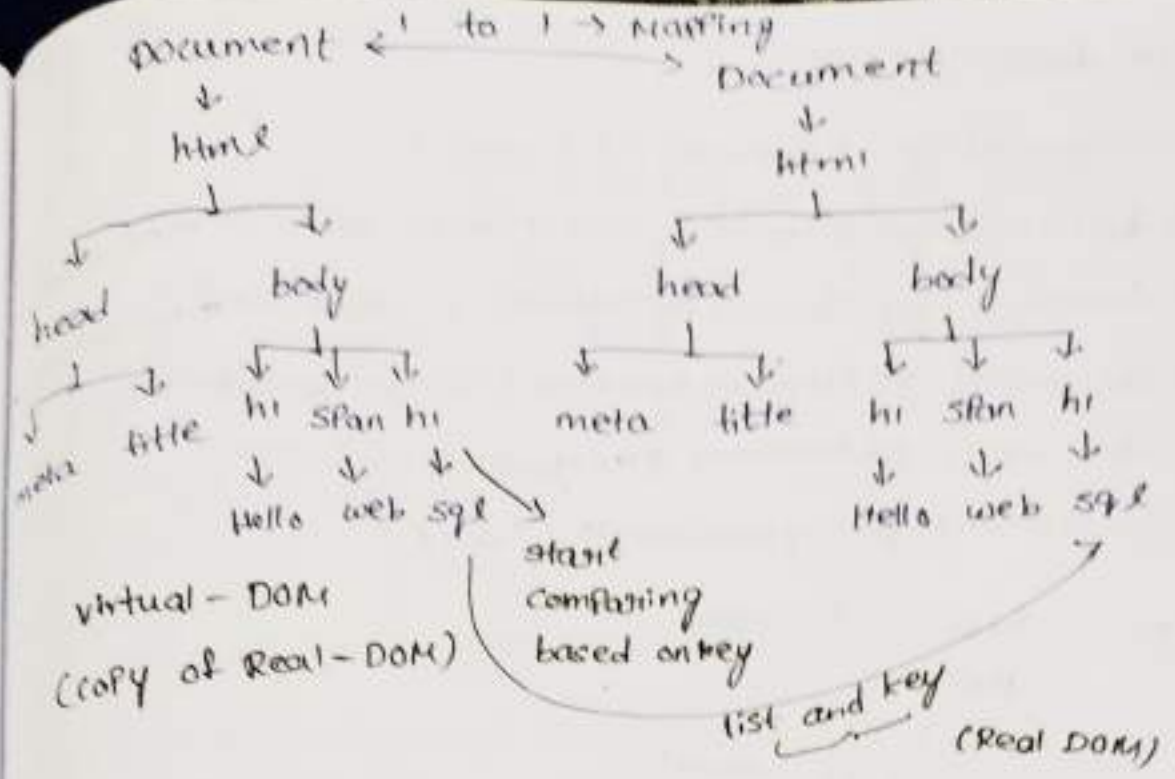


Eg:-

```
<!doctype html>
<html>
<head>
  <title>  </title>
  <meta />
</head>
<body>
<h1> Hi </ h1>
<span> web </span
<h1> sql </h1>
</body>
</html>
```

Component → { <h1> Hi </ h1>, web </span, <h1> sql </h1> }

Component → Class, function/ Presentation



Methods of DOM

Direct - Access Method

- document.doctype
- document.head
- document.title
- document.body
- document.links
- document.Images
- document.forms

Indirect - Access - Method

- document.getElementById ("value")
- doc.getElementByclassName ("value")
- doc.getElementBytagName ("value")
- doc.getElementByName ("value")
- document.querySelector ("value")
- doc.querySelectorAll ("value")

In-direct access:-

1. document.getElementById("value")
2. document.getElementByClassName("value") → HTML collection[]
3. document.getElementByTagName("value") → HTML collection[]
4. document.getElementByName("value") → Node list[]
5. document.~~getElement~~ querySelector("value")
6. document.querySelectorAll("value")

Ex1:- <h1> → id = 'demo'
Hello
</h1>
 → id = 'demo' Hello ✗

```
let a = document.getElementById("demo")
clg(a) // <h1> Hello </h1>
a.style.backgroundColor = "Red"
clg(a.textContext) // Hello
a.textcontext = "web-tech"
```

Ex2:- <h1> → class = 'demo1' web-Tech </h1>
 → class = 'demo1' sql

```
let b = document.getElementByClassName("demo1")
clg(b) // HTML collection [h1, p]
clg(b[0]) // <h1> Hello </h1>
b[0].style.color = 'Green'
b[0].textcontext = 'React'
clg(b[1]) // <p> sql </p>
```

```
<div> class = "demo2"
  Hi
</div>

let c = document.getElementBycLassName("demo2")
  clg (c) // HTML collection [div]
  clg (c[0]) // Hi
```

Ex3:-
```
<h1> Hello </h1>
<h1> web-Tec </h1>
let c = document.getElementByTagname ("h1")
  clg (c) // HTML collection [h1, h1]
  clg (c[1])
  c[1].style.color = "blue"
```

Ex4:-
```
<input type = 'text' id = " " name = "demo3" />
<div> name = 'demo3'
</div>
let d = document.getElementByName ("demo3")
  clg (d) // node.list [input, div]
  clg (d[1])
```

Note:- The difference b/w HTML collection and Node list is in HTML collection we can get only elements but in Node list we can get element as well as plain text.

Ex:-
```
<body>
  Hi
  <h1> Hello </h1>
    sql
  <p> Java </p>
</body>
```

o/p:- Hi

document.queryselector
("value")

1. value should be passed with CSS symbol.

2. will give reference of first-element.

eg:- <h1> id='demo' Hi </h1>

let a = doc.queryselector
("#demo")

clg (a)// <h1>Hi</h1>

<h2> class='demo1' </h2>

 class='demo1'

let b = doc.queryselector("demo1")

clg (b)// <h2> </h2>

document.queryselectorAll
("value")

1. value should be passed with CSS symbol.

2. will give reference in a nodelist [ ].

eg:- <h1> id='demo' Hello </h1>

let m = doc.qquery.selectorAll
("#demo")

clg (m)// nodelist [h1]

clg (m[0])

 class='demo1'

 class='demo1'

clg (n)// nodelist [p, span]

clg (n[1])

→ to avoid methods we are using Properties.

Properties / Traversal of DOM

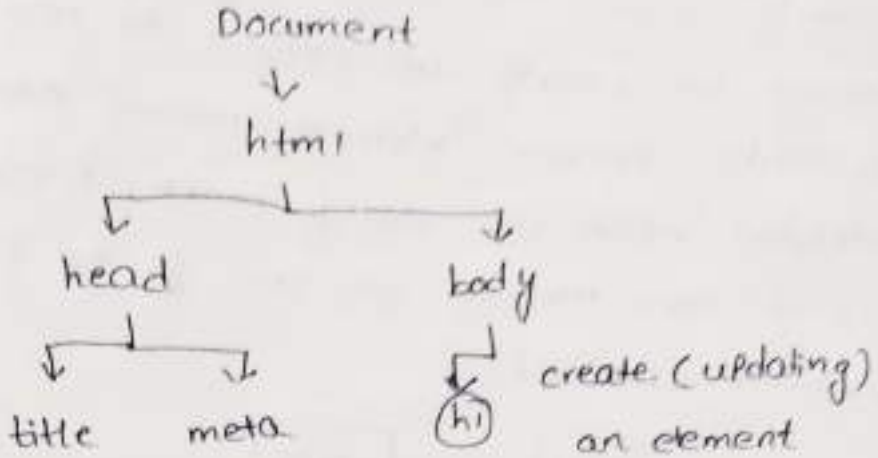
first child
first Element child
last child
last Element child
children // html collection[ ]
child nodes // node list [ ]
Parent Node
Parent Element
next sibling
next element sibling

→ targets ← Previous sibling
Previous Element sibling

→ only Plaintext / comment / link

Modification / Manipulation of DOM

```
<html>
<head>
</head>
<body>
</body>
</html>
```



⇒ updating / modifying DOM tree.

Two-ways :-

1. Inner HTML (Property)
2. create element (Method)

1. Inner HTML :-

/ create / update element in DOM.

syntax :-

document . body . inner HTML = " ele - name "

→ Property

eg :-

```
let a = document . body
clg (a) //<body> </body>
a.innerHTML + = "<h1> React </h1>"
a.innerHTML + = "<ol>
                 <li> web </li>
                 <li> sql </li>
                 </ol>
```

Dis-advantage :-

1. It reduces efficiency of Browser (whenever we create an element the old dom tree destory and new dom tree will create that time it reduces

efficiency of Browser).

2. Security issues.

(whenever we create an element the new element will override previous element means then we take concatenation mean we should kept information to overcome this method we will go for create an Element (method).

react - - webgl

3. Create Element :-

It is a method used to create / update element in a DOM.

Two - steps :-

step 1 :- create an element using create element method.

eg :- let h1 = document. create Element ("h1")

clg (h1) //<h1> </h1>

h1. text Content = "Hello-world"

<h1> Hello-world </h1>

step 2 :- Append an element to Parent using append child ( ) (or) append method document.

body. appendChild (h1).

ex :-

// create an ol tag.

let ol = document. create Element ("ol")

clg (ol) //

```
// create an li tag.
let li1 = document.createElement("li")
  clg(li1) // <li> </li>
    li1.textContent = "web"
    <li> web </li>
// append li to ol.
ol.appendChild(li1)
let liR = document.createElement("li")
  clg(liR) // <li> </li>
  liR.text content = "sql"
          <li> sql </li>
  ol.appendchild (liR)
// append ol to ParentElement (Body)
  document.body.appendChild(ol)
    ol.setAttribute("type", "A")
    ol.style.backgroundColor = "Red"

      <ol>
      <li> web </li>
      <li> sql </li>
      </ol>
```

web sql

Event's in JS

→ Event is an action performed by end-user on a web-Page.

Types :-

✓ Keyboard event ✓ form event

✓ Mouse event ✓ Pointer event

1. Keyboard event → key up, key down, key Press

2. Mouse event → on mouse enter, on mouse leave, double click

3. form event → input, focus, Blur, on submit

4. Pointer event → click

Event Handler / Event listner :-

→ It is a function, which type of event is given by end-user.

→ 3-Phase

* capturing
* target
* Bubbling

ways of Providing event's for element :-

→ 3 ways are there.

1st way :-

```
< button onclick ="demo()"> submit </button>

function demo ( )
{
  clg ('Hello-world')
}
```

2nd way :-

```
< button id='demo' > Resume </button>

let a = document.querySelector("#demo")

clg (a) // <button>Resume </button>
```

```
a.onclick = function( )
{
clg ("Hello-world");
}
```

(or)

```
a.onclick = ( ) =>
{
clg ("Hello-world");
}
```

2nd way :-

By using addeventlistner method we do.

add Eventlistner ("event", call Back fn, boolean)

method — name

→ In boolean default one 'false'.

Event - Propogation :-

→ flow of event.

Ex :- <button> submit </button>

3-phase

* capturing
* target (event)
* Bubbling

```
let btn = doc.queryselector ("button")
clg (btn)
btn.onclick = ( ) =>
{ clg ('Hello')
}
```

capturing :- It starts from the top of the dom tree and reaches until to the bottom of the dom tree (until target).

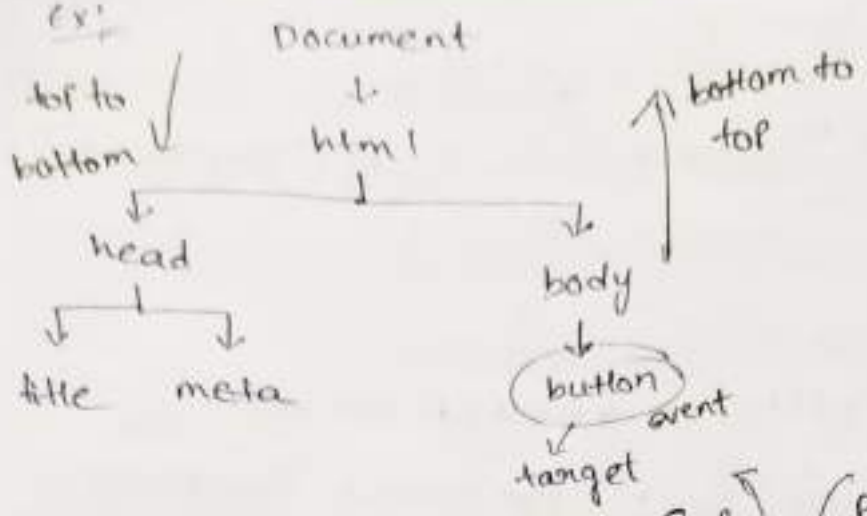
→ It's a top to bottom approach.

Target :- Target is nothing but a element which is having a event.

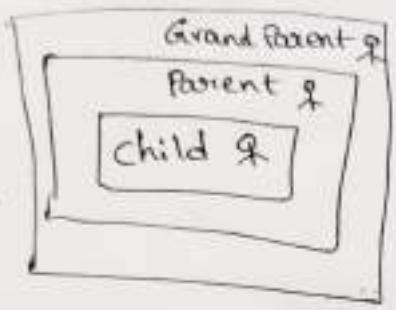
Bubbling :- It starts from the bottom of the dom tree and reaches until to the target top of the dom tree.

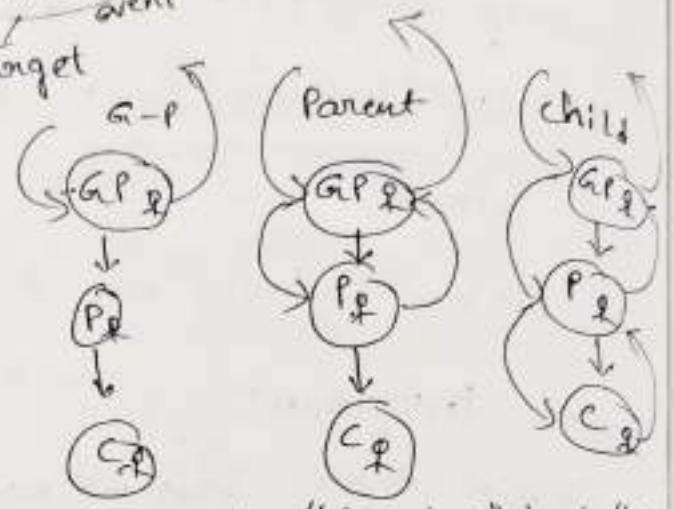
⇒ It's a bottom to top approach.

Ex:



Ex:





add EventListener ("event" CBF, Boolean)

true (capturing) false (Bubbling)

```
let G-P = doc.queryselector("#G-P")
    clg(G-P)
    G-P.addeventlistner("click", (e) =>
    {
        e.stopPropagation()
            (or)
        e.stopImmediatePropagation()
    }, false)
let Parent = doc.querySelector("#P")
    clg(Parent)
```

```
parent.addEventListner("click", (e) =>
{
  e.stopPropagation()
  (or)
  e.stopImmediatePropagation()
  }, false)
```

Asynchronous in JS

→ making a way for other function to execute.

- setTimeOut (cbf, delay-time)
- set Interval (cbf, delay-time)

} method → present in window obj

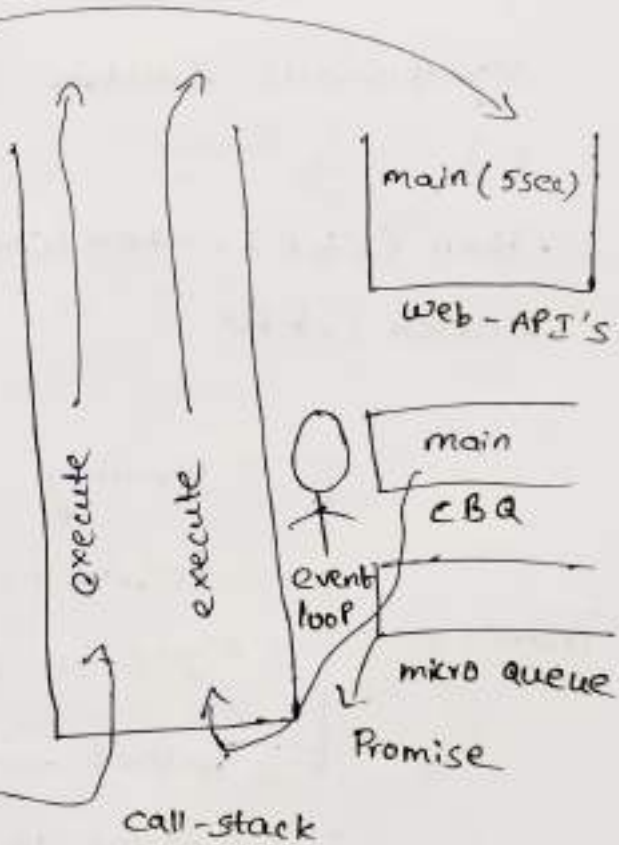
cbf → call-back-function

Eventloop :- It is a designed pattern mechanism. used to ~~the~~ control the call stack.

Ex:-

```
function main(m,n)
{
setTimeout(() =>
{
for (let i=m; i<=n; i++)
{
  clg(i);
}, 5000)
}
main(1, 10)
function demo()
{
clg('numbers Printed');
}
demo() // numbers Printed
```

Asynchronously



Promise :-

→ It is a object.

→ used to look after asynchronous function.

→ 3-Phases are there :-

* Pending phase :- Either Promise will be resolved (or) reject.
* resolve phase :- Asynchronous is successfull ~~with~~ Promise will be resolved → method .then (cbf)
* reject phase :- Asynchronous is not-successfull Promise will be rejected → method .catch (cbf)

Syntax :-

constructor

```
new Promise ((resolve, reject) =>
  {
  Asynchronous function
  })
  .then (cbf) . then (cbf)
  . catch (cbf)
```

Promise chaining
↓
To avoid this we can use async & await

Promise
- → Pending
- → resolve → .then( ) → .then( )
  (Async is successful)
- → reject → . catch ( ).
  (Async is not successful)

Ex:-

```
function main(m,n)
{
new Promise ((resolve, reject) => {
 setTimeOut (() =>
  {
  if (isNaN (m) || is NaN(n)
   {
    return reject( )
   }
  for (let i=m; i<=n; i++)
   { clg (i)
   }
  return resolve()
  }, 5000)
   .then (() => { clg ("Async is success"))
   .catch (() => { clg ("error occured"))
   }) }
     main (1, 10)
function demo( )
 {
  clg ('no's Printed')
 }
  demo( )
```

Event Delegation :- Event delegation in javascript is a Pattern that efficiently handles events. Events can be added to a Parent element instead of adding to every single element.

String and String Method's

→ Any sequence of character's which is enclosed with ' ', " ", and back ticks (` `).

→ It is a Primitive data type.

* 2-way's

1) Literal - way

```
let str = 'Hello'
clg(str) //Hello
```

2) by using new keyword

```
let str1 = new String('Hello')
clg(str1) //Hello
```

String - method's :-

```
let str = 'Hello-world'
clg(str) //Hello-world
```

1) length Property

```
clg(str.length) //11
```

2) index of ( )

```
clg(str.indexof('H') //0
```

3) to uppercase ( )

```
clg(str.touppercase()) //HELLO-WORLD
```

4) to lowercase ( )

```
clg(str.tolowercase()) //hello-world
```

5) charAt ( )

```
clg(str.charAt(2)) //l
```

6) charcode At ( )

(returns UTF (Uni code Transfer format) of an Particular character. (0 - 65535)

```
clg(str.charcode At(0))
```

7) Concat ( )

```
let str1 = 'Java-script'
clg(str1) ——→ clg(str.concat(str1))
```

let str2 = "Java script"

clg (str2) // Java • script

7) slice (start Index, end index)
It is used to create new string

clg (str2. slice (0, 4) // Java

clg (str2 · slice(-11, -7)) // java

8) sub string (start Index, end index)
It is used to create new-string.

clg (str2 · sub string (0, 2)) // Ja

10) sub- str ( ) used to create newstring.
It is a deplicative one.

clg (str2 · substr (0, 4)) Java

11) split ( ) (converts string to Array)

clg (str2 · split ("") // ['J', 'a', 'v', 'a', 's', 'c', 'r', 'i', 'p', 't']

clg (str2 split (' ') // ['Java', 'script']

clg (str2 split ( ) // ['Java-script']

14) startswith ( )

clg (str2 · startswith ('J')) // true

15) Endswith ( )

clg (str2 · endswith ('o')) // false

let str4 = "sql React"

clg (str4)

16) replace ( )

clg (str4 · replace ("sql", "JS")

12) repeat ( )

clg (str2 · repeat (2))

// Java script Java script

13) trim ( )

let str3 = " sql "

clg (str3)

clg (str3 · trim ( ))

It is used to remove spaces start and end Point.

17) Includes ( )

clg (str4 · includes ('J'))

// false.

18) at ( )

clg (str4 · at ('s'))

// 0, s

Array and Array method's

→ Array is used to store the data of an different data-types.

→ Heterogenous (its store any kind of data-types)

1) By using literal-way

```
let arr = [10, 'Hello', true, null, undefined]
  clg (arr)
```

2) By using new keyword

```
let arr1 = new Array (10, 20, 30, 40)
  clg (arr1)
```

Array Methods

```
let arr = [10, 20, 30, 40]
  clg (arr) // [10, 20, 30, 40]
```

1. length

```
clg (arr.length) // 4
```

2. index of ( )

```
clg (arr.index of (40)) // 3
```

3. Push ( )

(used to add an item of the end of an array)

```
clg (arr.Push (50, 60)) // 6
clg (arr) // [10, 20, 30, 40, 50, 60]
```

4. Pop ( )

→ used to ~~add~~ remove an item's from the end of array.

```
clg (arr.Pop()) // 60
clg (arr) // [10, 20, 30, 40, 50]
```

5. un-shift ( )

→ used to add an item at the starting of an array.

```
clg (arr.unshift (1, 5)) // 7
clg (arr) // [5, 10, 20, 30, 40, 50]
```

1) shift ( )

→ used to remove item's from the starting of array.

clg(arr. shift ( )) // 1

clg (arr) // [5, 10, 20, 30, 40, 50]

2) let arr = [10, 20, 30, 40]
0 1 2 3

clg (arr)

3) includes ( )

clg (arr. includes (50)) // ~~false~~ true

4) splice (start-Index, Delete-Count, Add-item)

→ used to modify an array.

let arr3 = [10, 20, 30, 40, 50]
0 1 2 3 4

clg (arr3. splice (0, 3, 'Hello'))
// [10, 20, 30]

clg (arr3) // ['Hello', 40, 50]

11) reverse ( )

→ clg (arr. reverse ( )) // [40, 30, 20, 10, 5]

15) Sort ( )

let arr 5 = [50, 20, 10, 40, 30]

clg (arr)

clg (arr5 . Sort ( )) // [10, 20, 30, 40, 50]

let arr 6 = [1000, 1, 20, 200, 100]

clg (arr6 . Sort ((a, b) =>

o/p:- [1, 20, 100, 200, 1000] ← return a-b // gives in ascending order

o/p:- [1000, 200, 100, 20, 1] ← return b-a // gives in descending order

8) concat ( )

let arr1 = [50, 60]

clg (arr. Concat (arr1))
// [5, 10, 20, 30, 40, 50, 50, 60]

9) slice (start Index, End Index)

→ It is used to create new-array.

clg (arr. slice (0, 3)) //
[5, 10, 20] ~~[10, 20, 30]~~

12) Join ( )

clg (arr. Join ('+')) //
~~10+20+30+40~~
50+40+30+20+10+5

13) isArray ( )

clg (Array. isArray (arr)) //
true

14) from ( )

let str = 'Hello'

clg (str)

clg (Array. from (str))

// ['H', 'e', 'l', 'l', 'o']

16) let arr7 = [10, 20, 30, 40]

16) for-in (itterates only index)

```
for (let ele in arr7)
  {
  clg(ele)
  }
```

o/p:- 0
1
2
3

17) for-of (itterates only values)

```
for (let ele1 of arr7)
  {
  clg(ele1)
  }
```

o/p:- 10
20
30
40

18) forEach( )

→ It is used to iterate values and index.

```
let a = arr7.forEach((values, index) =>
        {
        return `${value} => ${index}`
        })
clg(a);
```

o/p:- 10 => 0
20 => 1
30 => 2
40 => 3

19) map( )

→ It is used to modify an array.

→ returns new array.

```
let b = arr7.map((ele) =>
        {
        return ele + 10
        })
clg(b); // [20, 30, 40, 50]
```

20) filter( )

→ used to filter an array based on condition.

→ returns new array.

```
let c = arr7.filter((ele) =>
        {
        return ele > 10
        })
```

clg(c)// [20, 30, 40]

21) reduce( )

→ used to find total sum of an array.

let arr7 . reduce ((accumulator, last index) =>

{

return acc + li

3) 3), 10)

clg(d) // 100

clg(d) // 110

acc	li	
10	10 = 10	20
20	10 = 30	40
30	30 = 60	70
40	60 = 100	110

22) flat( )

→ It is used to take ele/item from depth of an array.

arr7 . flat (infinite)

Eg1:- [5, [20], 30, 40]

Eg2:- [10, [20, 10], 30]

let arr8 = [10, [20, 30], 40]

clg (arr8 . flate (infinite)) // [10, 20, 30, 40]

23) every ( )

→ every condition must be satisfy then only o/p will be true.

24) some ( )

→ ~~every~~ Anyone condition is satisfy o/p will be true.

25) fill ( )

→ To ~~fa~~ fill empty spaces in the array.

Object and object methods

→ used to store data in the form of key's and value's pair.

key → identifiers
value → Datatypes

→ 2 - ways

1) literal - way :

```
let obj = {
        key   : value
        name  : "abhi",
        id    : 1,
        arr   : [10,20,30],
        demo  : () => clg ('Hello')
        boolean : True,
        null    : null,
    undefined : undefined
}
```

identifiers

Data types

clg (obj)
clg (obj.arr[1])
clg (obj.demo())

2) Constructor function :- → used to create an object

```
function obj1 (name, id)
{
    this.name = name;
    this.id   = id;
}
let P1 = new obj1 ('anil', 1)
clg (P1)
```

identifier

this → it targets current object

Object - methods

```
let obj = {
        name : "abhi",
        id   : 1
}
clg (obj)
```

1) // Print values

```
clg (obj.name) // abhi
```

2) // add new key & value-Pair

```
obj.age = 20;
clg (obj)
```

3) // update value

```
obj.id = 100;
clg (obj)
```

4) // delete key & value Pair

```
delete obj.age
clg (obj)
```

8) assign ( )

(used to concat more than two object).

```
let obj2 = {
designation : "developer";
}
clg (obj2)
let d = Object.assign (obj, obj2)
clg (d)
```

5) // Print all values in the object

```
let a = Object.values (obj)
clg (a)
```

6) // Print all key's

```
let b = Object.keys (obj)
clg (b)
```

7) // entries

(used to convert object → Array type)

```
let c = Object.entries (obj)
clg (c) // [ [name: "abhi"] [id: 1] ]
```

9) seal ( )

→ we can't add new key & value, but existing value key's will be updated.

```
Object.seal (obj)
clg (object.isSealed (obj))
obj.name = 'ramu'
clg (obj)
obj.phno = 1234 ✗
clg (obj) → Not gona work
```

10) freeze ( )

→ we can't add new key and value Pair as well as we can't update existing key values also. O/P gives in boolean format.

```
Object.freeze (obj)
clg (object.is frozen(obj))
obj.name = "Java-script" ✗
clg (obj)
obj.address = "Punjagutta" ✗
clg (obj)
```

JSON (Java-script object Notation)

→ It is a plain-text-file which is used is used to store data in the form of key and value pair, both key and value should be enclosed with "".

→ two methods

* Stringify (convert object → JSON)
* Parser (convert JSON → object)

Step 1:- create a .json file.

Step 2:- we can use any data-type to create json otherthan undefined, function, bigint data-type.

Ex:-

Data.json

```
{ {
    "name" : "abhi",
    "id"   : 1 ,
    "boolean" : true,
    "null"    : null,
    "arr"     : [10, 20, 30],
    "obj"     : {
                designation : "developer"
                }
  },
  {
    "name" : "anil",
    "id"   : 2
  }]
```

0 index

Array of object

1 index

* fetch( )
* asyne & await } To fetch JSON
* AJAX

Browser Storage :-

1. Local Storage :-

→ Data in a local storage store Permenantly until someone delete's it.

→ It will store upto 5MB of data.

Methods in local storage :-

i) localstorage.setItem ("key", "value")

→ It used to create data in a local-storage.

ii) localstorage.getItem ("key")

→ It used to fetch data from local-storage.

iii) localstorage.clear( )

→ It will erase data in local-storage.

2. Section-storage

→ Data in a section-storage deletes, ones the section end.

→ It stores upto 5 MB of data.

Methods in section-storage :-

i) sectionstorage.setItem ("key", value")

→ It used to create data in a session-storage.

ii) sessionstorage.getItem ("key")

→ It used to fetch data from session-storage.

iii) sessionstorage.clear( )

→ It will erase data in session-storage.