-! JAVASCRIPT !-

-! Syllabus !-

1. Introduction
2. How Java Script Engine works -> Global Execution Context
3. Scopes
4. Types of Errors
5. Variables - Operators, typicasting
6. functions - 7 types
   1. function Decleration
   2. Generator function
   3. function Expression
   4. Arrow function
   5. Nested function
   6. functional Programming
   7. Immediate Invoking function Expression (IIFE)
7. DOM - Daument object model
   1. What is DOM
   2. DOM Methods
   3. DOM Properties (or) traversells.
   4. DOM Modification (or) Manipulation
   5. Events
   6. Event Propogation
   7. Event Delegation
8. Ansynchronous
9. Promises
10. Arrays , Array programs
11. Strings. , String Programs

12. Objects , Object programs.
13. Storage parts.
    local storage
    session storage
    cookies
14. API
    JSON
    AJAX
    fetching API
15. Shallow copy
16. Deep copy
17. Destructuring.
18. Dereferring. (changing of Address)
19. REST and spread parameters
20. call apply bind

Introduction:-

* Javascript was created by Beniden Eich in 1995.
* Beniden Eich worked at Netscape and implemented ~~javascript~~ Java script for their web browser, Netscape navigator.
* The idea was the major ~~part~~ of interactive part of the client web app.
* Initially Java script name changed several times

1st code name → Mocha
2nd code name → Live script
3rd code name → Java Script
4th code name & final code name - ECMA script

* In Netscape navigator 2.0 betas (September 1995) it was called as Live script.
* In Netscape Navigator 2.0 betas 3 (December 1995) it was named as Java Script.
* The ECMA in ECMA Script comes from the organization that hosts the primary standard
* The original name of that organization was ECMA - European Computer Manufacture Association.

** JavaScript and ECMA Script both are same **

Q. What is Java Script and why do we use Java Script?

* Java Script is used to convert static page into dynamic page.
* Java Script is a light weight programming Language
* It is Object based programming language as well as Object Oriented programming Language.
* It was used by several websites for scripting webpages
* Java Script is client side scripting language and one of the most efficient and commonly used scripting language.
* The term client side scripting language means that it runs at client side (client machine) inside the web browser.

Characterstics of Java Script:-

* Java Script is Synchronous (we can't go back) and interpretting (we can go back) Language (line by line)
* It is open source
* It is case sensitive in nature
* It is an Error free language
* It is platform Independent
* It is obp and oop language.
* It works on both server side (Node js) as well as client side.
* It is a single threaded language.

Note:-

* Java Script is HLL - High level Language

* Browser → It is an application which provides environment to run Java script instructions.

* Java script code can run both inside and outside the browser from High level language to Machine level language

Inside - Javascript Engine.

outside - Node Js.

Execution of Java Script on browsers.

* We can execute Java script instructions directly in the console provided by the browser.

* We can execute java script instruction by embedding in HTML page.

* There are two ways to embedded java script in HTML page.

1. Internal:-

With the help of script tag we can embedd java script instruction

Syntax:-

```
<html>
  <body>
    <script> // Js code </script>
  </body>
</html>
```

2. External :-

We can create a separate file for java script with an extension '.js' i.e., javascript.js

To link the javascript file with HTML Page using src="" attribute of ~~Scr~~ script tag.

```
<html>
  <~~body~~>
  <head>
    =
    <script src="">  </script>
  </head>
  <body>

  </body>
</html>
```

Relative path

Absolute path.

async :-

Whenever we use async attribute both java script code and HTML code executes Parallelly

```
<html>
  <head>
    =
    <script async src="" - "" > </script>
  </head>
  <body>

  </body>
</html>
```

Relative path

Absolute path

defer :-

Whenever we use defer attribute first it will execute the HTML code and then java script code will be executed.

```
<html>
  <head>
    <script defer src="_"></script>
  </head>
  <body>
  </body>
</html>
```

Relative Path
Absolute path

Note:- If you write javascript code in between the HTML code, whenever javascript engine sees defer in script tag, it will hold the execution of javascript code and first executes the HTML code later executes the js code (or) script tag.

Errors in Java script :-

There are 4 types of errors in javascript

1. Syntax error
2. Reference error
3. Type error
4. Rang error.

-: Tokens :-

Tokens are the smallest unit of any Programming Language

Tokens consists of :

1. keywords
2. Identifiers
3. Literals
4. Separators

1. Keywords :-

A pre defined reserved words which is understandable by the java script engine is known as keywords.

Note :-
Every keyword must be in the lower case and they should not be used used as identifiers

Eg :- if, for, let, const, var, while, do-while null, switch, break, continue, case, void function, instance of, throw, export, delete, catch Private, Package, true, debugger, extends, default interface, super, with, enum, return, try, let, yield typeof, Public, ~~for~~, static, new, else, finally, false, import, do, Protected, ~~void~~ in, implements, this, await, throws, volatile, Private, protected float, finally, ~~function~~, goto, eval, double, long native, new, Package, class, ~~enum~~, export, super, import, so on..

2. Identifiers !-

The name given by the programmer to the component of java script like : variables, functions, class, etc...

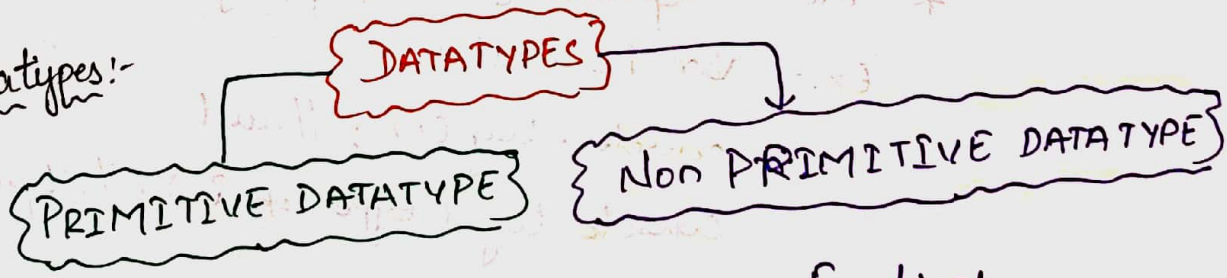
Rules !-

1. An identifier cannot start with number but it can contain numbers.
2. We cannot use keywords as Identifiers
3. In identifiers we cannot use any special symbols other than '$' and '_'

3. Literals (or) Data (or) Values !-

Data which is used in the javascript Program is called as literals (or) Data (or) values.

Datatypes !-



Numbers
Bigint
Null
Nan
Boolean
Undefined
String
Symbol

functions
Maps
Sets
Objects
Arrays
for of
for in

1. Number :-

* It represents the mathematical operations and all the numbers like +ve, -ve, float, long comes under num datatype.

* for number data type we have a range i.e., +ve $(2^{53}-1)$ to -ve $(2^{53}-1)$.

Ex:- let a = 105; b = -2, c = 10124 73;

2. Bigint :-

* We can use bigint type to store the number beyond the given range and also we have to suffix with 'n'.

Ex:- 10200 3000 40000n;

Ex:- let b = 10200300040000n;

3. Null :-

* Null represents empty object as a value.

Ex:- var i = null;

console.log(i); // null

console.log(type of i); // Object

4 NaN :-

* It will check whether the give number value is number or not.

5. Undefined :-

* A variable that has not been assigned a value i.e; undefined.

* The undefined property indicates that a variable has not the assigned value

Eg:- var a ; console.log(a); // undefined.

Boolean:

* It can hold two values that are true and false.
* It is useful for controlling program flow like if, switch.

String:

* In javascript strings can be represented using single quotes, double quotes (or) backticks
* The start and end of the string must be done with the help of same quote.
* If a text contains single quotes then it can be represented using double quotes and also backticks

Ex:- let a = 'I 'am a developer' ✗

let b = " I 'am a developer " ✓

Backtick:

A backtick can be used for multiple line strings

```
let a = ' my hobbies
        are
          Singing
          cooking
          Eating
          Mars Movies '

console.log(a);
```

* The string by using backticks is also known as templates.
* The advantage of it is we can substitute and express by using interpolation [${}]

Symbol :-

* It is used for uniqueness in console in red color
* We can see in unique color in our colosole.

Syntax: Symbol ( )

Ex:- let c = Symbol ("Hello Web");
clg (c); // Symbol ("Hello Web")

Type of operator :-

It is a keyword used as a unary operator (checks only one data type or values) to identify the type of data.

Syntax :-

i, typeof (value)

ii, type of value

Examples :-

i, let a = 10126491264132946139s
console.log (a);
console.log (typeof a); // number

ii, let b = "hey".
console.log (b);
console.log (typeof b); // string

iii, let c = 'hey'
console.log (c)
console.log (typeof c); // string

```
let d = `abcdef
ghijkl`

console.log(d);
console.log(typeof d); // string

let e = true
console.log(e)
console.log(typeof e) // boolean

let f = 100n
console.log(f);
console.log(typeof f); // Bigint

let g = null
console.log(g);
console.log(typeof g); // object

let h = Symbol("400");
```

:- Operators :-

What are operators in Java script ?

In java script an Operator is a special Symbol used to perform operation and operand (values and variables)

Here is a list of different operators.

1. Arithmetic operator
2. Assignment Operator
3. Comparision operator
4. Logical operator

1. Arithmetic Operator :-

Arithmetic operators are used to perform arithmetic calculations

Ex: Var a = 3+5 // 8

Operator	Name	Example
+	Addition	x + y
-	Substraction	x - y
*	Multiplication	x * y
/	Division	x / y
%	Modulator / remainder	x % y
**	Exponential / Power	x ** y
++	Increment by 1	x ++ / ++ x
--	Decrement by 1	x -- / -- x

-: Operators :-

What are operators in Java script ?

In java script an Operator is a special Symbol used to perform operation and operand (values and variables)

Here is a list of different operators.

1. Arithmetic operator
2. Assignment operator
3. Comparision operator
4. Logical operator

1. Arithmetic Operator :-

Arithmetic operators are used to perform arithmetic calculations

Ex: Var a = 3+5 // 8

operator	Name	Example
+	Addition	x + y
-	Substraction	x - y
*	Multiplication	x * y
/	Division	x / y
%	Modulator / remainder	x % y
**	Exponential / Power	x ** y
++	Increment by 1	x ++ / ++ x
--	Decrement by 1	x -- / -- x

Assignment Operator :-

operator	Name	Example
=	Assignment operator	a = 7
+=	Addition Assignment	a += 2
-=	Substraction Assignment	a -= 2
*=	Multiplication Assignment	a *= 2
/=	Division Assignment	a /= 2
%=	Modulus Assignment	a %= 2
**=	Exponential Assignment	a **= 2

Comparision Operator (or) Relational operator

Operator	Description	Example
==	equals to : returns true if the operand are equal	x == 1
!=	Not equals to : returns true if the operands are not equal.	x != 5
===	Strictly equals to : returns true if the operands are equal and has the same data type	x === 5
!==	not strictly equals to : returns true if the operands are equal but of different type and of the same type	x !== y

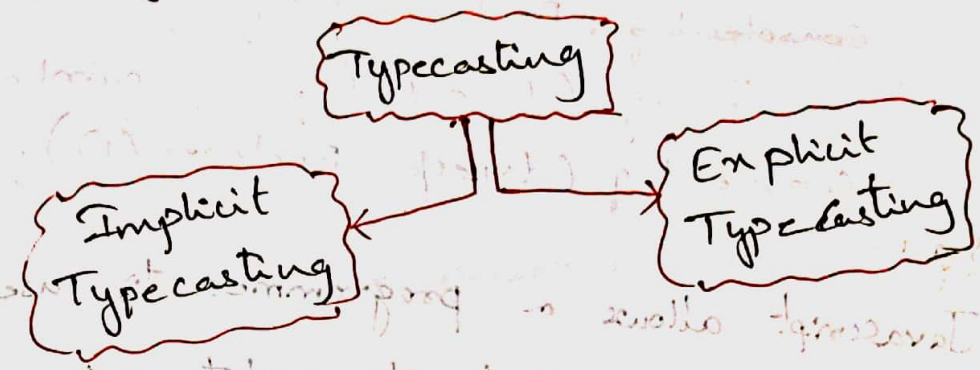
>	greater than: True if left operand is greater than right operand	x > y
>=	greater than equals: true if the left operand is greater than (or) equals to right operand	x >= y
<	less than: true if the left operand is less than right operand	x < y
<=	less than or equals: true if the left operand is ~~greater than~~ ~~or~~ lesser than equals to right operand	x <= y

4 Logical Operator:-

operator	Description	Example
&&	Logical AND: True if both operands are true else it returns false	x && y
||	Logical OR: True if any one of the operands are true. else return false	x || y
!	Logical NOT: True if the operand is false else-vice versa	! x

Typecasting :-

The process of converting one type to another type of data



Implicit Typecasting :-

The process of converting one type of data to another type of data by javascript engine implicitly (implicit typecasting), when a wrong type of data is used in the expression is known as Type corrosion or implicit typecasting

```
let a = 10, b = "20";
console.log (a+b) = 1020
console.log (a*b) = 200
console.log (a/b) = 0.5
console.log (a%.b) = 10
console.log (b-a) = 10
console.log (a+number(b)),=30
```

Explicit Typecasting :-

The process of converting one type of value to another type of value by developer is known as explicit typecasting.

Conversion of any number to number

i, String to number

ii, If a string is a valid numbers we get real number

iii, If the string consist any other characters then we get NaN as output

Ex:-

```
console.log (String (a) + b);          1020
console.log (Boolean ("I"));           true
Console.log (Boolean (0));             false
Console.log (typeof 1);                number
:Console.log (typeof Boolean (1));     boolean
```

Hoisting :-

* Javascript allows a programmer to use a member (variables) before the declaration statement.
* This characterstics is known as hoisting.

* Hoisting is nothing but calling a variable before declaration of initialization is called hoisting.

Temperol Dead Zone :-

* The time between the variables which are created during its initialization is known as TDZ i.e Temporal Dead Zone.

* "Let" and "const" belongs to temperol Dead zone.

Eg:-

```
Let a
clg (a) // undefined — because ! initialiged.
clg (b) // Reference Error — Temperol Dead Zone
let b = 10
clg (b) // 10
```

* The problem occurs only if the variables used before declaration. [Reference Error is the problem]

Scopes!-

There are three types of scopes

1. Global scope
2. Local scope / script scope
3. Block scope / functional scope

1. Global scope
   Which can be declared outside the block

2. Local scope / script scope
   which can be declared inside the block of function

3. Block scope / functional scope
   which can be declared inside the block(s) flower brackets.

Note!-
Global scope will be present inside the window object but local and block scope will not be present inside the window object.

Variables!-

Variable is a block memory used to store the values.

It is dynamic in nature (no need to maintain any data types) a name blok of memory used to store a value [Container to store data]

There are 3 types of variables

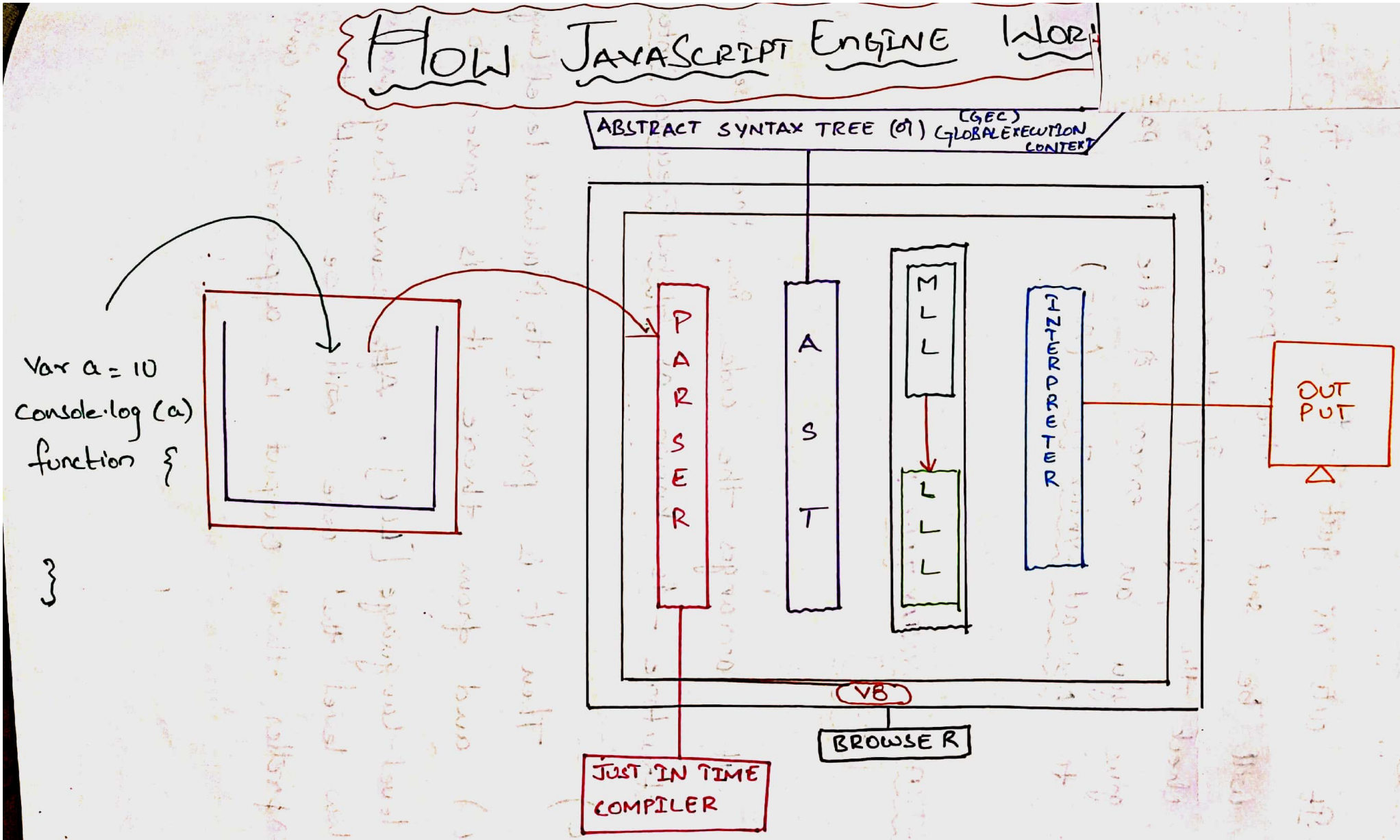
1. Var
2. let
3. const

Var	let	const
1. Var is global scope	let is local scope	const is local scope

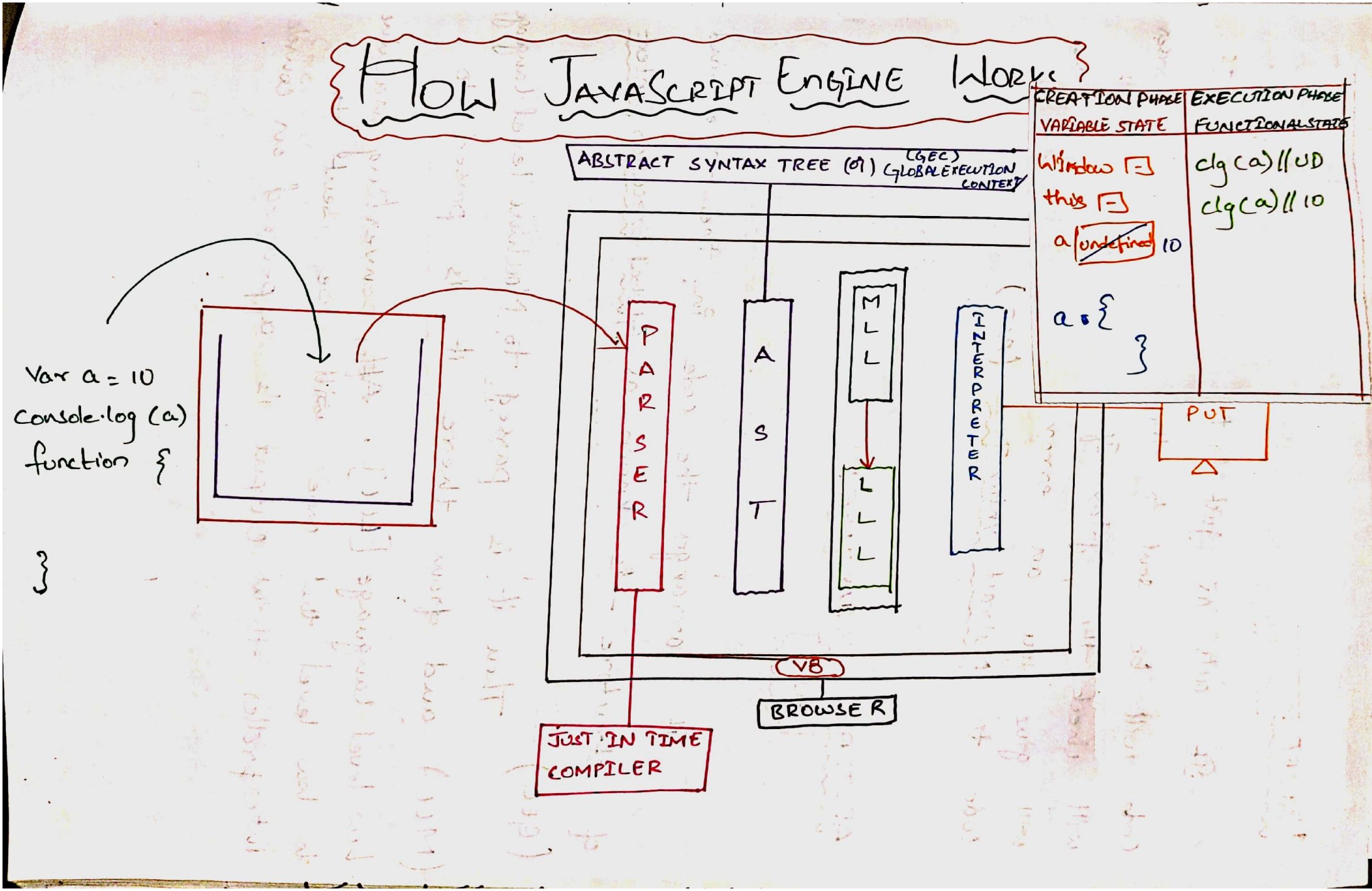
2. Hoisting is possible
Eg:-
clg (a) // undefined
var a = 10
clg (a) // 10
	Hoisting is not possible
	Eg:-
	clg (b) // RE
	let b = 30
		Hoisting is not possible
		Eg:-
		clg (c) // RE
		const c = 50

DECLARATION and INITIALIZATION		
var a = 10 clg (a)	let b = 30 clg (b)	const c = 50 clg (c)
REINITIALIZATION		
a = 50 clg (a)	b = 90 clg (b)	c = 80 clg (a)
REDECLARATION		
var a = 20 clg (a) var a a = 180 clg (a); // 180	let b = 40 clg (b) Type error let b b = 80 clg (b); // 180	const c = 10 clg (c) Type error const c c = 240 clg (c) // Error

HOW JAVASCRIPT ENGINE WORK



How JavaScript Engine Work?



PARSER :-

It acts as just in time compiler. Firstly JS code will be sent to the parser - then the Parser will check the syntax. If there is an error it will give you an error or else it parses the code to Abstract Syntax Tree (AST)

ABSTRACT SYNTAX TREE :-
[AST]

It arranges the code in the tree kind of structure - is known as Global Execution Context (GEC) -

Then it is parsed to Machine level Language (MLL) and from there it is parsed to Low level language [LLL]. After converting machine to low level the code will be sent to interpreter then Output is appeared on console.

Note!- It is a combination of C++ & V8 (Chrome JS Engine) with bundle of methods and rules

Node provides the environment JS outside the browser this invention helps the JS to gain its popularity in usage as backend language.

We can run JS both inside and outside the browser using Node.js

Global Execution Context!-

Creation Phase / Variable state	Execution phase / functional state

window [-]
this [-]
a ~~undefined~~ 10
add { }
	clog(a); //undefined
	clog(a); // 10

The creation of Execution context (Global Execution context) happens in 2 phases!-

1st phase :- Variable state / Creation Phase / Memory state

2nd phase :- Execution Phase / functional state / code state

I st :- Variable state / creation phase / Memory state :-

In the creation state, the execution context is first associated with an execution context object.

The execution context object stores lot of information (or) data which the code in the execution context will occur during runtime.

II nd :- Functional state / Execution phase / code state :-

The javascript engine reads the code in the current execution context one more then update variable object with actual variables of the variables

Then this code is parsed by parser gets transfer to executable byte code and finally gets executed.

Note :- Create a global object that stores all the variables & functions in the global scope as methods (or) properties (of function (or) properties

* We cannot access the code of the function context defined in it.
* Set up memory space for variables & function defined globally.

Looping Statements :-

1. for loop :-

The praess of executing statement (or) blak of statements repeatedly multiple times is known as loop.

Syntax :-

```
for ( Decleration (or) Initialization ; condition ; updation ) {

    // statements

}
```

Example :-

```
for (let i=0 ; i<=10 ; i++ ) {

    console.log (i);

}
```

Note :- When we design a loop it is responsibility of the programmer to break the loop after achieving the desired task. The programmer gets into infinate loop.

2 -: While loop :-

A while loop evaluates the condition inside the paranthesis. i.e ( ) .

If the condition evaluates to True the code inside the while loop is executed. the condition is evaluated again and again until the condition becomes false.

Syntax!-

```
while (condition) {
    // statements
    - updation -
}
```

Example!-

```
let i=0;
while (i<10){
    console.log (i)
    i++;
}
```

III :- do while loop :-

* The body of the loop is executed first then the condition will be executed.

* If the condition is evaluated and becomes true the body of the loop inside the do statement is executed again.

* The condition is evaluated again and again until the condition becomes false; Once the condition becomes false the execution of loop stops.

Syntax :-

```
do {
    // statements
}
while (condition);
```

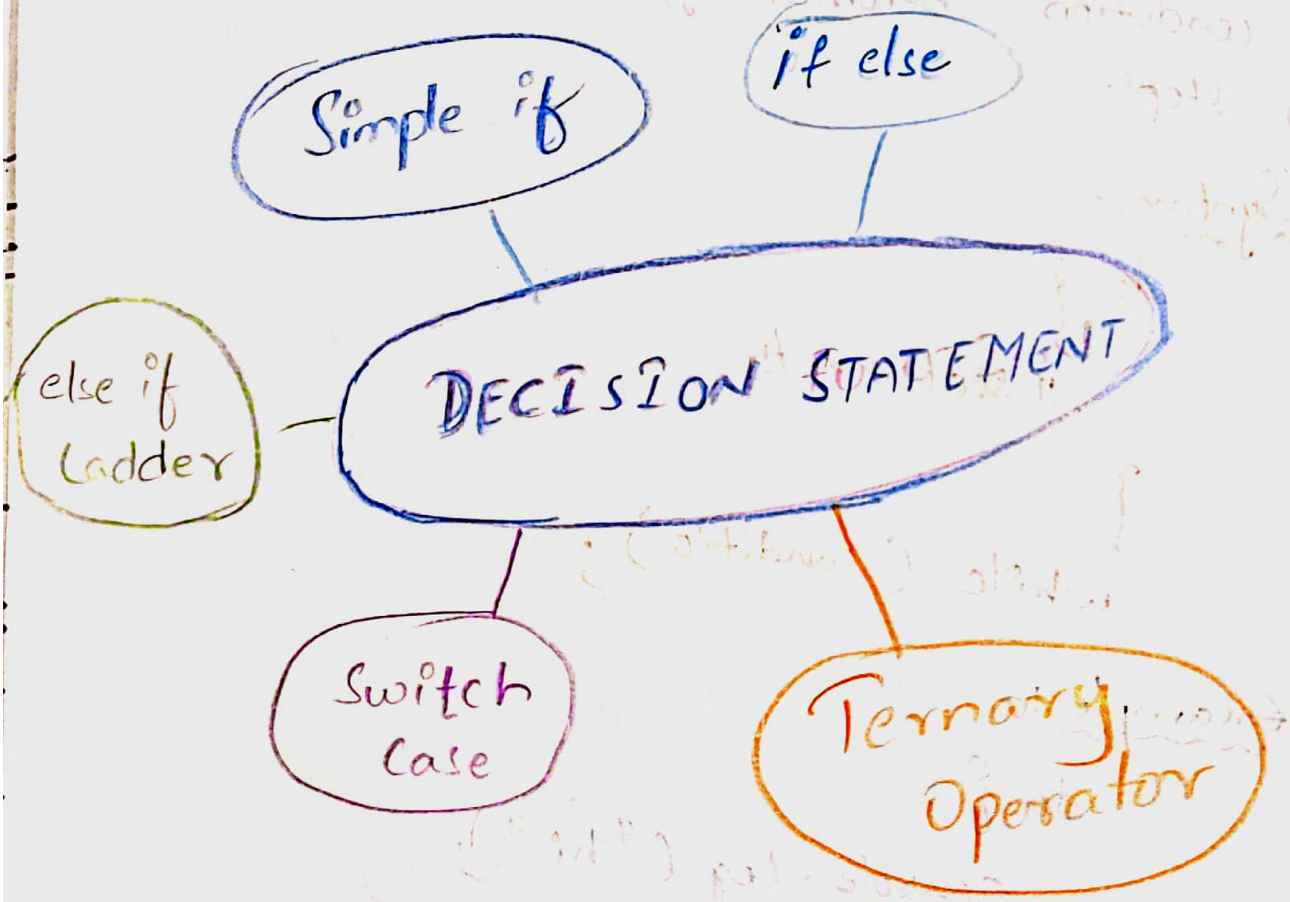
Example :-

```
do {
    console.log("hi");
}
while (5 < 6)
```

Decision Statements :-

Decision statements helps to skip the block of instruction when we don't have a favouring situation.

Note!-
The instruction of loading home page should be skipped if entered password is incorrect.



I st. ~: simple if :~

Syntax:-

```
if (condition) {
    // statements
}
```

Example:-

```
if (7>6) {
    console.log ("hi");
}
```

II ~: if else :~

Syntax:-

```
if (condition) {
    // statements
}
else {
    // statements
}
```

Example:-

```
if (5>6) {
    console.log ("hi");
}
else {
    console.log ("bye");
}
```

III ~: else if ladder :~

Syntax:-

```
if (condition) {
    //statements
}
else if (condition) {
    //statements
}
else if (condition) {
    // statements
}
else {
    // statements
}
```

Example:-

```
if (n%.5==0 & n%.3==0)
{ console.log ("fizz buzz");
}
else if (n%.5==0) {
    console.log ("fizz");
}
else if (n%.3==0) {
    console.log ("buzz");
}
else {
    console.log ("Nah!");
}
```

(iv) -: Switch case :-

Syntax :-

```
switch (expression) {
    case value 1 : statement
    case value 2 : statement
    case value 3 : statement
    :
    case value n : statement
    default : statement
}
```

(v) ~: Ternary Operator :~

Syntax :-

```
condition ? expr_if true : expr_if false
```

~: FUNCTION :~

* A set of operations written inside a block is called function.

(or)

* Named block of instructions which is used to perform a specific task

* Function gets executed only when it is called.

* The main advantage of function is we can achieve code reusablity.

Main Purpose :- Code Reusablity

* which means write once call n number of times (or) call anywhere.

* Function is an object, generally we have n number of functions.

* We are going to discuss 7 types functions.

FUNCTIONS

i, Function Declaration (or) Named function (or) General function

ii, General function (or) Pure function

iii, function Expression (or) Anonymous function

iv, Arrow function

v, Nested function

vi, functional Programming

- Callback function
- Higher order function

vii, Immediate Invoking function Expression (IIFE)

1) Function Declaration :-

Parameters

Syntax :-

```
function function_Name (---) {
    // statements
}
```

function_Name (---)
Arguments.

* Function is an object
* Name of a function is a variable which holds the reference of a function object.
* Creating a function using function statements support function hoisting.
* We can also have a function call before the function declaration (hoisting)

i) function without arguments :-

Syntax :-

```
function_Name ( )
function function_Name.( ){
    // statements
}
```

Example :-

```
demo ( )
function demo ( ) {
    console.log ("hi");
}
```

ii, function with arguments & Parameters

Syntax:-

```
function_Name(arg1, arg2)

function Function_Name (Parameter1 , Parameter2)
{
    // statements;
}
```

Example:-

```
demo (10,20)
function demo (a,b) {
    console.log (a+b);
}
```

iii, function with arguments & without Parameters

We can access the arguments with arguments variable present inside the functions.

Syntax:-

```
function Function_Name ( ) {
    // statements
    console.log (arguments);
    console.log (arguments[0] + arguments[1]);
}
function_Name (arg1, arg2);
```

Example :-

```
function demo () {
    console.log (arguments);
    console.log (argument [0] + argument [1]);
}
demo (10,20)

console.log (demo); // Prints function.
```

```
var a = 20;
function demo () {
    var a = 10;
    clg (a); // 10
    clg (this.a) // 20
}
demo ()
```

this keyword :-

this keyword is used to access only current object variable which is present inside the window object (only global space variable)

-: Using 'return' keyword :-

Syntax:-

```
function function_Name ( ) {
    // statements ;
    return "Hello return keyword";
}

function_Name ( );
clg (function_Name( )); // Hello return keyword
```

Example:-

```
function demo ( ){
    clg ("hi");
    return "Hello return stmt";
}

demo ( ); // hi
clg ( demo ( )); //Hello return stmt
```

* Without return statement if I console the function the output will be undefined. To remove the undefined we will go with return keyword.

-: use strict :-

Javascript strict mode is a way to opt into a restricted varient of javascript.

use strict directive was new ECMA script version 5.

It is not a statement but a literal expression ignored by earlier version of js.

You can use strict mode in all your programs, it helps you to write cleaner code like preventing you from using undeclared variables.

Declare inside the function. It has local scope only the code inside the function is strict mode.

Syntax!-

```
'use strict'

// Statements;

function function_Name(){
    // Statements
}

function_Name();
```

Examples!-

```
'use strict'
x = 10
clg (x) // identifier error

function demo (a,a,b){ // Syntax Error
    clg (a+a+b);
}

demo (10,20,30);
```

~: Prototype :~

It is a copy of an object (mirror image of an object). All functions will have a prototype but except some functions. If you want to check the prototype of the function we need to follow two steps. they are:

I clg (object.getprototypeof(a) == function_Name.Prototype);

II clg ("Prototype" in function_Name);

2. GENERATOR FUNCTION :-

Scenario :- We know that we cannot print (or) access the statements that are written after the "return" statement. But for a change — if we want to print (or) access the statements after the return statement we have to go for generator function.

Ex:-
```
function demo() {
    clg ("hi");
    return "hello";
    clg ("bye");
    return "tata";
}
```
Unreachable statements

If you want to print unreachable statements (or) if you want to give multiple return statements you want to give to the function we can do it by using Generator function.

In Normal function it is not possible

Syntax :-
```
function * function_Name ( ) {
    // statements ;
    yield " message " ;
}
```
Yield is a keyword - which works as return statement only but this yield keyword will help the developer to reach (or) print unreachable statements

The method that helps to print (or) reach yield keyword and unreachable statements is next().value.

Example :-

```
function * demo (.) {
        clg ("hi");
        clg ("hello");
        yield "Good Morning";
        clg ("Madam");
        yield "Generator function starts";
        clg ("Bye");
}

let d = demo ()
clg (d.next().value); // hi hello Good Morning
clg (d.next().value); // Madam
```

Generator function starts

Bye

FUNCTION EXPRESSION
(or)
Anonymous function

* Function Expression or Anonymous function is a function with out a name.

* An anonymous function or function expression is not accessible after its initial creation.

* Therefore, you often need to assign it to a variable.

Syntax :-

```
let a = function () {
        // statements
}
a();
```

To understand let a & a(), read 2nd & 3rd points.

By using a(), we can call the anonymous function.

* Function Expression is same as function Declaration but function name won't be there in function expression.

* Hoisting is not possible in function Expression.
  why? - Because we will get reference error if we try to call the function before function declaration.

* If we check datatype of the function expression - it will return function only.

Example!-

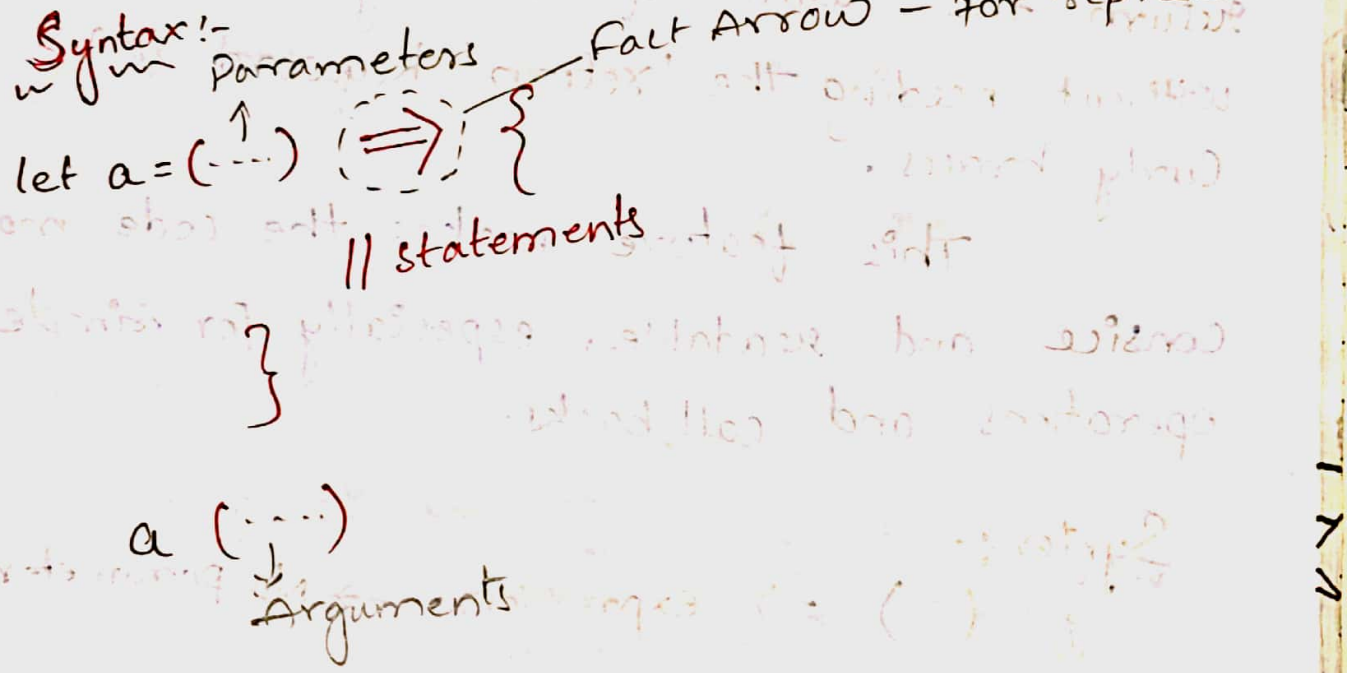
```
a();  // reference error (or) type error.

var a = function () {
        clg ("hi");
}
a();
clg (typeof a);
```

~: ARROW FUNCTION :~

Arrow function was introduced in 2015 from ES6 version of javascript.

Purpose :- To reduce the syntax.

Syntax :-



There are two types of Arrow functions, they are:

1. Implicit Arrow function.
2. Explicit Arrow function.

Implicit Arrow function :-

An implicit arrow function is a shorthand writing for arrow functions, where this implicit arrow function automatically return the result of a single expression without needing the 'return' keyword (or) curly braces.

This feature makes the code more concise and readable, especially for simple operations and call backs.

Syntax :-

i, ( ) => expression → No parameters.

ii, Parameter => expression → Single Parameter

iii, (P1, P2, P3, ...) => expression → Multiple Parameters

Examples

i, No Parameters :-

```
const greet = ( ) => 'Hello';
clg (greet ( )); // Hello.
```

ii) Single Parameter:-

```
const square = x => x * x;
clg (square(5)); // 25
```

iii) Multiple Parameters:-

```
const add = (a,b) => a+b;
clg (add (2,3)); // 5.
```

Return keyword not required: it will automatically return the values

Explicit Arrow function :-

An explicit arrow function in javascript is an arrow function, where the function body is enclosed in curly braces '{}' and an explicit 'return' statement is mandatorily used to return a value.

This explicit arrow function is used for the functions that require multiple statements or complex operations.

The explicit arrow function makes the code more readable and clear when the logic involves more than a single expression.

Syntax:-

```
() => {
    return "Hello"
}
```

* Arrow function accepts this keyword.

Disadvantages of Arrow function

i, Hoisting is not possible.
ii, Constructor is not possible
iii, new keyword is not possible
iv, call, apply, bind is not possible
v, argument - variable is not possible
vi, Generator function is not possible

These all are not possible because it does not have its own prototype.

~: FUNCTIONAL PROGRAMMING :~

Functional Programming is a programming Paradigm or style of programming that relies heavily on the use of pure and isolated functions

In functional Programming we have two types, they are:

1. Higher order function
2. Call back function.

1. Higher order function :-

A function which accepts another function as parameter is called as higher order function.

2. Call back function :-

A function which accepts another function as an argument is called as call back function

Example :-

```
function demo (a, b, c) {
    return c (a, b);
}

demo (10, 20, function (e, f) {
    console.log (e+f) // 30
});

let add = demo (70, 80, function (e, f) {
    return e+f
})

console.log (add);
```

Higher Order function

Call back function

~: NESTED FUNCTION :~

In Javascript we can define a function inside of another function.

Syntax:-

```
function Outer_function(){
    // statements
    function inner_function(){
        // statements
    }
}
```

Example :-

```
function demo(){
    clg("Outer function");
    function d1(){
        clg("Inside function");
        // statements;
    }
}
```

* The demo() is known as Parent function.
* The d1() is known as child function.
* The d1() is a local function to demo() function - It cannot be accessed from outside.
* To use d1() outside, the demo() must return the reference of d1().

Nested function can be written in 3 ways.

1st way :-

```
function demo() {
    clg("outer function");
    function d1() {
        clg("1st inside function");
        // statements;
    }
    d1()
    function d2() {
        clg("2nd inside function");
        // statements;
    }
    d2()
}
demo()
```

II way :-

```
function demo () {
    clg ("Outer function");
    function d1 () {
        clg ("1st inside function");
        // statements;
    }
    function d2 () {
        clg ("2nd inside function");
        // statements;
    }
    return d1;
}
demo () ()
```

↑ child call

↓ Parent call

III - way :- [Multiple functions calling at the same time]

```
function demo () {
    clg ("outer function");
    function d1 () {
        clg ("1st inside function");
        //statements.
    }
    function d2 () {
        clg ("2nd inside function");
        //statements;
    }
    return [d1, d2];
}
```

Lexical Scope:-

The ability of javascript engine to search for a variable in the block scope - if it isn't present, then it checks in the local scope - if it isn't present, then it check in the global scope. This process is known as scope chain.

The scope chain is generally established between nested functions i.e., (parent function and child function).

Example:-

```
var a = 10
function demo() {
    clg ("outer function");
    function d1 () {
      var a = 50;
      clg ("hi d1 - inside function");
       clg (a);
    }
    var a = 90;
    function d2 () {
       clg ("hi d2 - inside function-2");
       clg (a);
    }
}
```

Closure :-

The closure is the bind between parent and child function is known as closure.

The disadvantage of closure is Memory wastage - Memory wastage why ? - Because same address is created

Lexical scope is the advantage of Nested function.

- and -

Closure is the disadvantage of Nested function.

Lexical scope and closure only occurs in Nested functions.

~: Immediate Invoking Function Expression :~

(IIFE)

When a function is called immideatly as soon as the function object is created is known as immideate invoking or invokation function Expression.

Steps to achieve IIFE!-

1. Treat a function like expression by declaring a pair of parantheis or brackets.

2. Add another pair of brakets or parantheis next to it which behaves like a function call statement

Syntax!- callBackfunction/Immediately calling function

```
( ) ( )
↓
Expression
```

Ex!-

```
(function fc() {
    clg C"Hi");
}) ( );
```

Use Cases:-

Avoid Polluting the global name scope:-

Our application could include many functions and global variables from different source files, it is important to limit the number of global variables. If we have the same initiation code that we don't need to use again, we could use the IIFE pattern as we will not reuse the code again, Using IIFE in these cases is better than using a function declaration (or) a function expression

In our Javascript 5, we can have only one Immediate Invoking function Expression.

If you want multiple IIFE's then you need to ~~ter~~ terminate each function by using Semicolon (;) (or) store it in to a new variable.

~: DOM :~

~: Document Object Model :~

* Document is an objected created by the browser.
* The document object is a root node of DOM tree.
* In DOM, every HTML element is considered as a node i.e., js object.
* DOM allows to modify the content - rendered by the browser without reloading the page.
* Therefor DOM helps to make a webpage dynamic.

Note :-

* Any modification done using DOM is not updated to original page. Therefore once we reload the page all the - modifications done using DOM will be lost
* We can write the content on the browser dynamically with the help of write() & writeln() of document object

document.write():-

This method helps us to print the content or output on the webpage.

Syntax:-

```
document.write(" ");
```

Examples:-

```
document.write("//strings");
document.write("//number");
document.write("//stored variables");
```

like:

```
let a = 55;
document.write(a);
```

document.writeln():-

This method also prints the content (or) output on the webpage as well as it will print the space.

Syntax:-

```
document.writeln(" ");
```

Example:-

```
document.writeln("hi"); // hi_
document.writeln("js"); // hi_js_
```

Window.alert():-

This method will give you popup message or alert message on the webpage with ok button.

Syntax:-

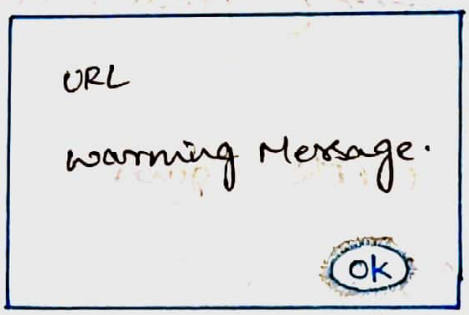
```
window.alert("  ");
```

Example:-

ip:-

```
window.alert("warning Message");
```

op:-



Window.prompt():-

This method allows the end user to give inputs.

We can mention the datatypes before this method.

Syntax:-

```
[Datatype] window.prompt("  ");
```

Examples:-

```
window.prompt("Enter your message");
Prompt("Message");
```

window.alert():-

This method will give you popup message or alert message on the webpage with Ok button.

Syntax:-

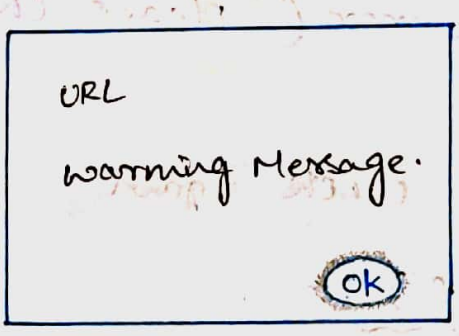
```
window.alert("  ");
```

Example:-

ip:-

```
window.alert("warning Message");
```

op:-



window.prompt():-

This method allows the end user to give inputs.

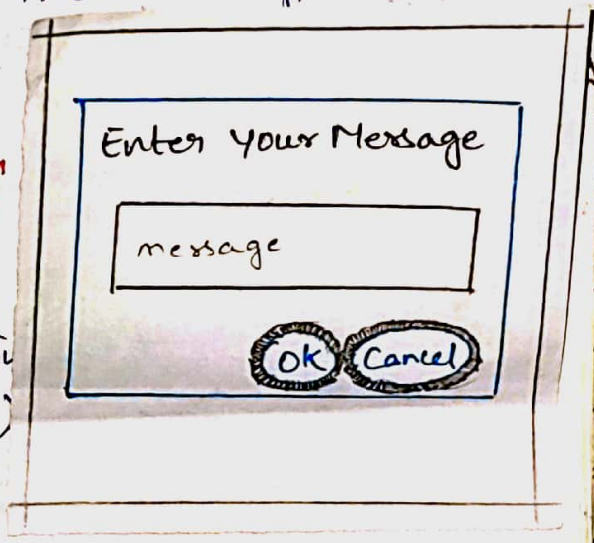
We can mention the datatypes before this method.

Syntax:-

```
[Datatype] window.prompt(
```

Examples:-

```
window.prompt("E
Prompt("Message")
```



Console.error( );-

This method allows the developer to provide the error messages in console.

The error in console is appeared as "the error message is highlighted in red color with a cross mark"

Syntax:-

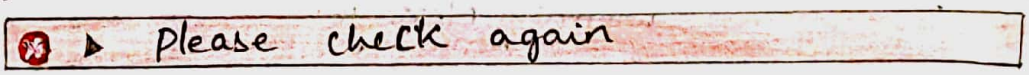
```
console.error("    ");
```

Example:-

IP:-

```
console.error("please cheik again");
```

OP:-



console.warn( ):-

This method allows the developer to provide the warning message in console.

The warning in console is appeared as: "the warning message is highlighted in yellow color with warning emoji"

Syntax:-

```
console.warn("    ")
```

Example:-

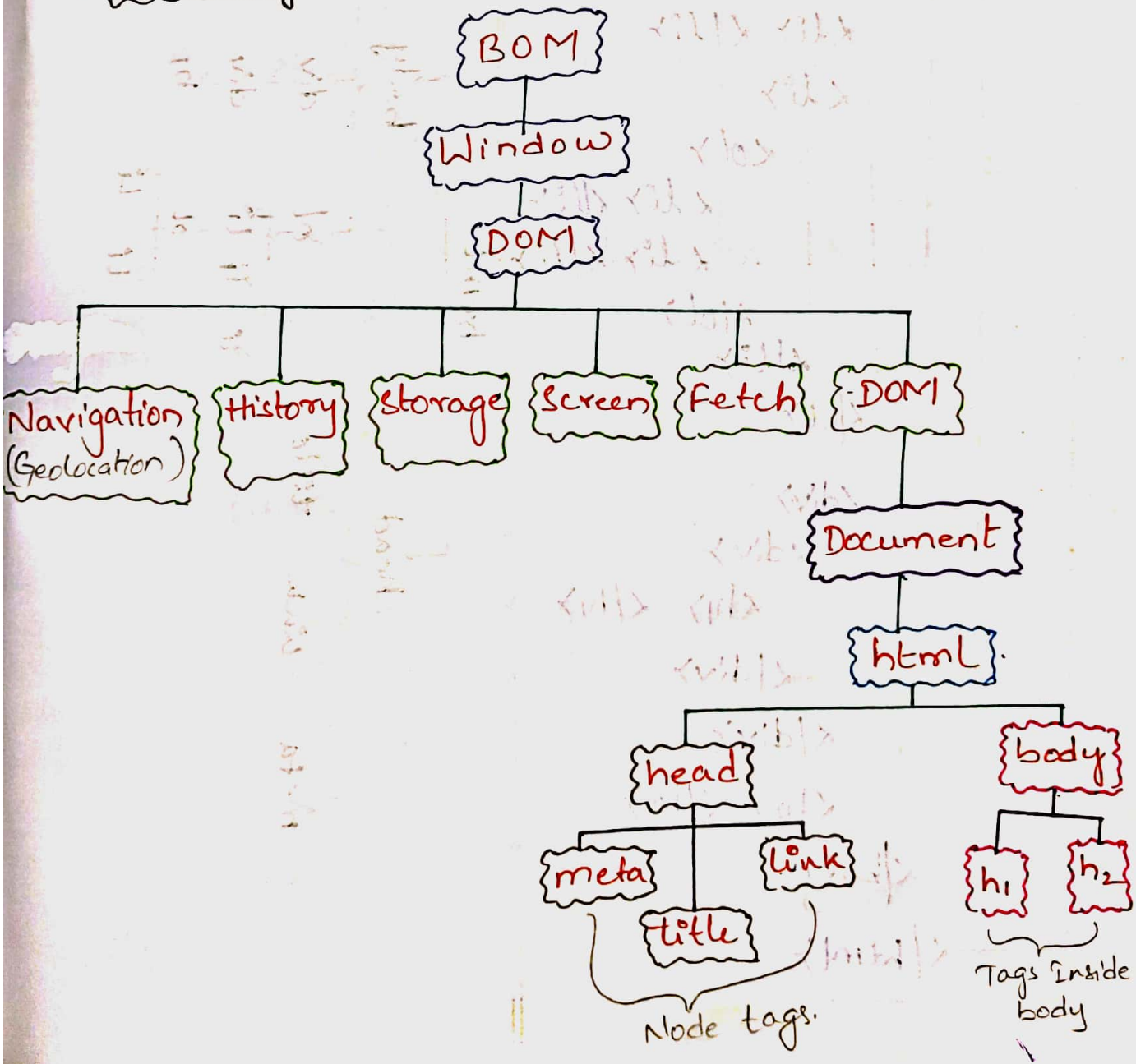
IP:-

```
console.warn("warning");
```

OP:-

* DOM is built by a window.
* DOM is an Object and it is an API provided by window.
* It is having a root element for DOM and it is document.
* The life span for DOM is until the session ends or page reloads.
* DOM is not built by using javascript.
* DOM is using javascript.

-!Hierarchy!-

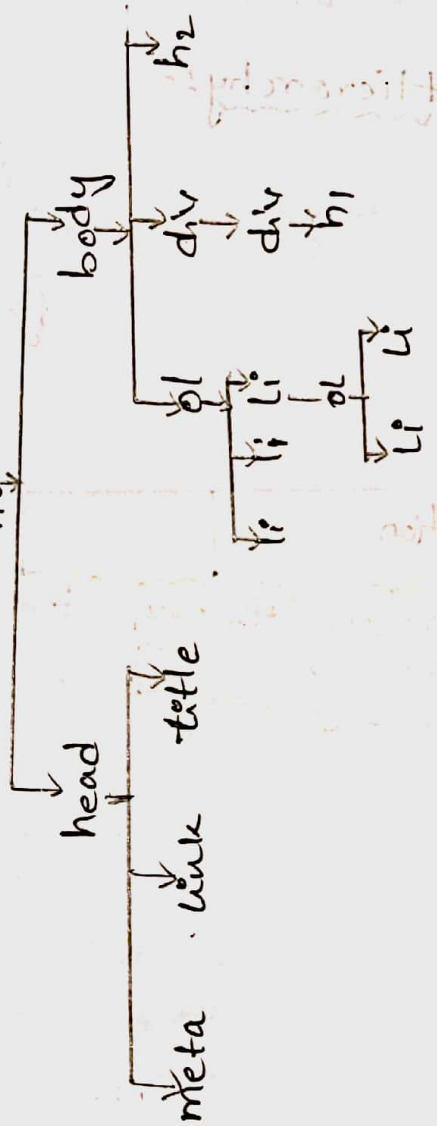


// Creating DOM structure from HTML code :-

// HTML code :-

```
<html>
  <head>
    <meta>
    <title> </title>
    <link>
  </head>
  <body>
    <ol>
      <li> </li>
      <li> </li>
      <li>
        <ol>
          <li> </li>
          <li> </li>
        </ol>
      </li>
    </ol>
    <div>
      <div>
        <h1> </h1>
      </div>
    </div>
    <h2> </h2>
  </body>
</html>
```

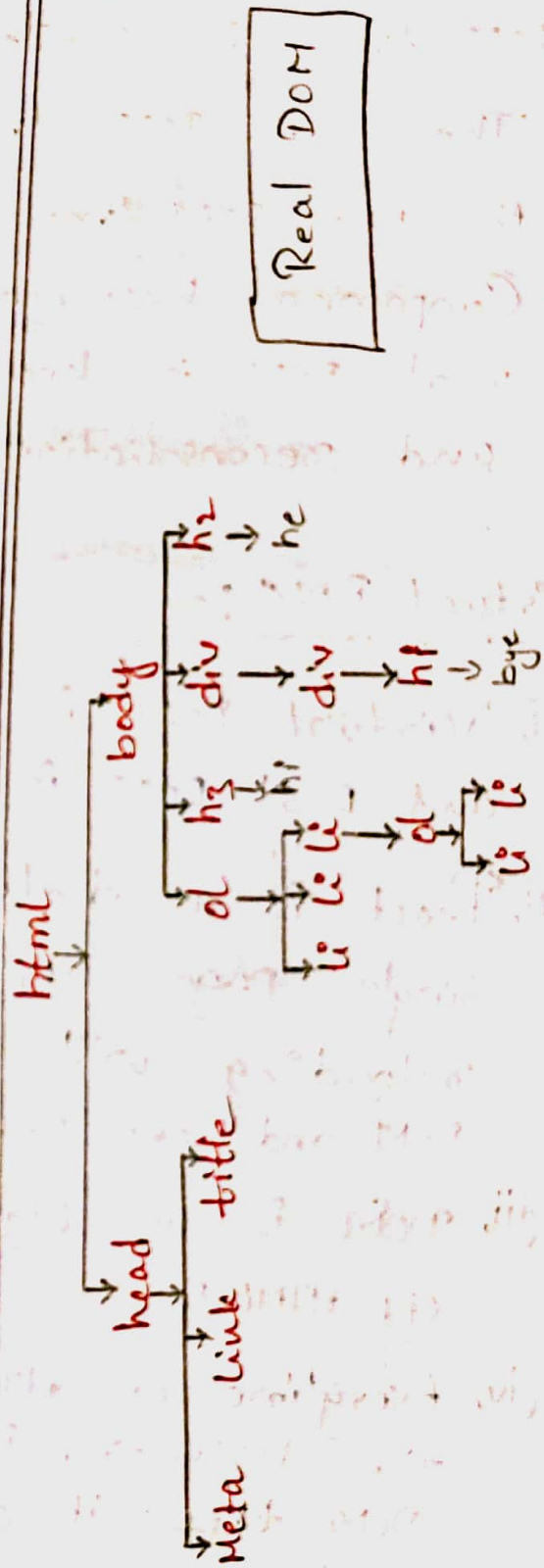
Virtual DOM



// Making one change to before HTML code!-

// HTML code!-

```
<html>
  <head>
    <Meta>
    <title> </title>
    <Link>
  </head>
  <body>
    <ol>
      <li> </li>
      <li> </li>
      <li>
        <ol>
          <li> </li>
          <li> </li>
        </ol>
      </li>
    </ol>
    <h3>hi</h3>
    <div>
      <div>
        <h1>bye</h1>
      </div>
    </div>
    <h2>he</h2>
  </body>
</html>
```



Note!-

i) Whenever a new element is added in HTML5 then old DOM tree is distroyed and new DOM tree is created.

ii) The old DOM tree is virtual DOM and the new DOM tree is called real DOM.

iii) Comparing between the virtual DOM and real DOM is known as Patching, Deffing and reconsoliation.

~: Virtual DOM :~

i) Virtual DOM is like a human shadow and Real DOM is like a human.

ii) React is a declarative and it is a single page application. In React page reloading will not happen only one virtual DOM and one real DOM will be there

iii) DOM is an Object oriented representation of HTML 5.

iv) Everytime an HTML document is given to the browser, it automatically generates DOM tree. it can be accessed and modified using the root node in the Java script.

DOM METHODS

- Direct Access Methods
- Indirect Access Methods

To print these use console.log( ) statement.

- document.body ( )
- document.URL ( )
- document.baseURI ( )
- document.images ( ) [img, img, img]
- document.forms ( ) [form form form]
- document.links ( ) [a, a, a]

- document.getElementById ( )
- document.getElementsByClassName ( )
- document.getElementsByTagName ( )
- document.getElementsByName ( )
- document.querySelector ( )
- document.querySelectorAll ( )

Direct Access Methods :-

I. document.body ( ) Targets only body.

II. document.URL ( ) Targets complete web page

III. document.baseURI ( ) Returns the HTML document's base Uniform Resource Identifier (URI)
The working of this property is quite similar to that of document.URL

IV. document.images ( ) Targets all images in the webpage and it store all images in an Array known as HTML collection - Not Pure Array

V. document.forms ( ) Targets all the forms in the webpage and it stores all forms in an Array, known as HTML collection
- Not pure Array.

VI. document.links(): Targets all the links in the webpage in the form of array. This is known as HTML collection.

DOM Direct Methods:-

We can target the tags directly like forms, anchor tags and image tags, etc. These will give you HTML collection.

HTML collection:-

* An HTML collection is an array-like collection (list) of HTML elements.
* The elements in a collection can be accessed by index (starts at 0).
* The length property returns the number of elements in the collection
* HTML collection is not a pure Array
* We cannot modify that array.

Indirect Access Methods :

I. document.getElementById() :

It will target the tag based upon Id name in the method.

Ex!-

ip!- let a = document.getElementById("id name");

= clg(a);

op!- // It will provide you the whole specific tag for which the id name is matched.

* It will not give you any collection.

II. document.getElementsByClassName() :

It will target the tag based upon the class name - name in the method.

Ex!-

ip!- let b = document.getElementsByClassName("//class name");

= clg(b);

op!- // It will provide you HTML collection of tags

III. document.getElementByTagName() :

It will target the tag based upon the tag name name in the method.

Ex!-

ip!- let c = document.getElementByTagName("// tag");

op!- // It will provide you HTML collection of tags.

IV. document.getElementsByName():

It will target the tag based upon the name attribute's name in the method

It is same as class name but it is present in node list

Ex!-

ip!- let d = document.getElementByName("//Name attribute value");
clg(d);

op!- // It will give you node list with selector

V. document.querySelector():

This method will target the tag based upon the Id, class name and tag name by using simple selector symbols.

Ex!-

ip!- let e = document.querySelector("//#id, .class, element");
clg(e);

op!- // It will give you only tag but the tag will also be printed based upon occurance. mean!- when a specific selector matched with the value given inside the querySelector(), then the first occurred of that selector only gets printed

(VI). document.querySelectorAll( ):

This method will target the tag based upon the Id, class name and tag name by using selector symbol.

Ex!-

ip:- let f = document.querySelectorAll("#id4 .class1 Element");
clg(f);

op!- // It will give you node list.

Note!-

* HTML collection and node list both are not a pure array, but it is an array
* HTML collection - it will target only the tags. Ex!- [h1 tag, h2 tag, div tag, etc..]
* Node list - it will target plain text inside the div as well as the remaining tags.
Ex!- [text, h1, text, h2, text, h3, etc, ~~h4~~ ...]

DOM Properties / Traversal :-

I. firstChild :-

It will target only the plain text of a Parent which is present in first position.

Ex :-

```
<div id = "div-1">
    hello
    <h1> hi </h1>
</div>
clg (document.getElementById ("div-1").firstChild);
```

II. firstElementChild :-

It will target the first element child which is present at first position in a div tag

Ex :-

```
clg (d.firstElementChild);
```

III. Last child :-

It will target the plain text of a Parent tag which is present at last position

Ex :-

```
clg (d.lastChild);
```

(iv) lastElementChild :-

It will target the last element child which present at last position in a div tag

Ex:- clg(d.lastElementChild);

(v) nextSibling:-

It will target the plain text which is present after a particular element in the parent tag.

Ex:- clg(d.nextSibling);

(vi) nextElementSibling:-

It will target the element or tag which is present after a particular element in the parent tag.

Ex:- clg(d.nextElementSibling);

(vii) previousSibling :-

It will target the plain text which is present before the particular tag/element in the parent tag.

* It will target from bottom to top.

Ex:- clg(d.previousSibling);

(VIII) previous Element Sibling :-

It will target the tag or the element which is present before the particular tag in the parent tag.

Ex :-

```
clg (d.previousElementSibling);
```

(IX) children :-

It will target all the children of a parent tag and gives you a HTML collection

Ex :-

```
clg (d.children);
```

(X) childNodes :-

It will target all the children like tags as well as plain text of the parent tag and gives you node list.

Ex :-

```
clg (d.childNodes);
```

op :-

[text, h1, text, h2, text]

(XI) Parent Element (or) Parent Node :-

It will target the parent element of the children.

Ex :-

```
clg (d.parentNode);
```

-: DOM MODIFICATION / MANIPULATION :-

1. innerText :-

It will target the text of a particular tag and also it can update the content of a particular tag.

Ex:-

```
<h1 id="d1"> hi </h1>
let d = document.getElementById("d1"); // hi
clg(d.innerText);
d.innerText = "hello";
d.innerText += "hi Js";
clg(d.innerText) // hello hi Js.
```

2. textContent :-

It is same as innerText, it will update the content and we can print the content.

Ex:-

```
d.textContent = "Ethel";
d.textContent = "Dousan";
clg(d.textContent); // Ethel Dousan.
```

(ii). innerHTML :-

It is used to update the content and it can print the content of a particular tag and also update the tag i.e; it can create a new tag.

Whenever we use innerHTML, we will face three disadvantages. They are:

i) Security Issue
ii) Override the content
iii) Reduce the efficiency of browser (DOM Tree)

Whenever we are not using +=, it may print empty space and it will not take previous value

Ex:-

```
<h1 id = "d1"> Hello </h1>
let h = document.getElementById("d1");
clg (h.innerHTML) // Hello
h. innerHTML = "Hero"
clg (h.innerHTML); // Hero
h. innerHTML = " "
clg(h.innerHTML); // space
h. innerHTML = " Good Morning JS ";
h.innerHTML += "_Good Morning Reshma Mam.";
clg (h.innerHTML); // GoodMorningJS_GoodMorning Reshma Mam.
```

```
let b = document.body
let o = b.innerHTML = `<ol>
                        <li> home </li>
                        <li> resource </li>
                        <li> login </li>
                       <ol>`;

clg(b);
clg(o.firstElementChild); // undefined.
```

To overcome the disadvantage of innerHTML, we can use createElement().

(IN). createElement():-

* We will create the element (or) tag with the help of createElement() which is present in document.

* Whenever we are using "createElement()" we can need to follow one more step i.e; append() (or) appendChild().

Ex:-

```
let b = document.body
let ol = document.createElement("ol");
   b.append(ol); (or) b.appendChild(ol);
let li = document.createElement("li");
   ol.append(li);
```

10 setAttribute() :-

* Whenever we want to give any attribute to the particular tag by using javascript, we can use this setAttributes().

* This setAttribute() accepts the key and value Pair.

Syntax :-

```
element.setAttribute("key", "value");
```

Ex :-

```
ol.setAttribute("id", "list");
```

11 removeAttribute() :-

* This removeAttribute() is used to remove the attribute from the targetted tag.

Syntax :-

```
element.removeAttribute("key");
```

Ex :-

```
ol.removeAttribute("id");
```

Note :-

* If we want to provide css / style properties to a particular tag we can use style variable.

Ex :-

```
ol.style.color = "red";
ol.style.backgroundColor = "yellow";
ol.style.border = "2px solid";
```

String interpolation

String interpolation in javascript is a technique for embedding variables (or) expression directly within strings.

This lets you create dynamic strings that can be changed based upon data

It improves code readblity and maintainablity compared to string concatination

Syntax :- `` `stmts ${ } stmts` ``

Example :-

Ip :-
```
let age = 23
let str = `hi I am Ethel
My age is ${age}.`
clg (str);
```

Output :-
hi I am Ethel
My age is 23

-! EVENT :-

* Event is an action performed by End user on a web page.

* An End user can perform different events such as:

1. Mouse Events
2. Keyboard Events
3. Pointer Events
4. Onclick Events
5. form Events.

* There are two ways of adding events on DOM:
1. By using JavaScript
2. By using JQuery.

* There are 3 possible ways to perform Events. They are:

I. HTML Event Handler. [Inline Event]
II. DOM level Event Handler
III. Event Listeners

I. HTML Event Handler:-

→ In HTML Event Handler, we assign an event handler with the function as an attribute in the HTML tag.

→ But using HTML event handlers are considered as a bad practice because of 2 reasons They are!

I. Readablity.

II. Timing Issue.

I. Readablity:-

→ Difficult to read when we have multiple event handlers on same element because of Even Handler code mixed with HTML code.

II. Timing Issue:-

→ If the element is loaded fully before javascript code, user starts interacting with it, which in turn gives error.

Ex:-

HTML:

```
<button onclick = "clickHere()" > HTML Event Handler </button>
```

JS:-

```
function clickHere() {
    document.write("Hello I'm HTML Event Handler");
}
```

output:-

HTML Event Handler	⇒Hello I'm HTML Event Handler

II. DOM level Event Handlers:-

→ In DOM level Event Handlers, we have to give our element (or) tag an identity which could be an id (or) a class.

→ As compared to HTML Event listener, In this DOM level Event Handler the Scope of function can easily be controlled.

→ One common point is We can have Only 1 event handler per event in both.

Ex:-

HTML code:-

```
<button id = "but1"> DOM level Event Handler </button>
```

Js code:-

```
let a= document.getElementById("but1");
a.onclick = () => {
        document.write("Hello I'm DOM level EventHandler");
}
```

output:-

DOM level Event Handler.

⇓

Hello I'm DOM level Event Handler.

III. Event Listeners :-

→ In Event listeners, we can use events by adding an event listner to an object. By this way we can handle wide range of events triggered by user.

→ In this we can have multiple event handlers as per event type.

→ Here we have two types of Event Listeners They are :

1. addEventListeners ( ) - to register an event handler
2. removeEventListener(,) - to remove an event handler

I. MOUSE EVENTS :-

→ The mouse event is an interface that represents the events that occur due to the user interacting with the help of mouse.

→ In mouse events we have different types of events such as :

1. mouse over
2. mouse leave
3. mouse enter...

II. KEYBOARD EVENTS :-

→ The keyboard event object describe a user interaction with the keyboard.

→ Each event describes a single interaction between the user and a key on the keyboard.

→ In keyboard events we have different types of events such as:

1. Key press
2. Key Down
3. Key up ...

III POINTER EVENTS :-

→ The Pointer events are DOM events that are fired for a pointing device. They are designed to create a single DOM event model to handle pointing input devices such as a Mouse (or) Pen (or) stylus (or) touch.

→ It is similar to mouse Events...

→ In pointer events we have different types of events such as

1. Pointer Down
2. Pointer Leave
3. Pointer Enter...

IV Onclick EVENTS :-

→ The Onclick event occurs when the user clicks on an HTML element.

V FORM EVENTS :-

→ In form events we have :

1. Submit button
2. Reset button
3. Geographical submit button.

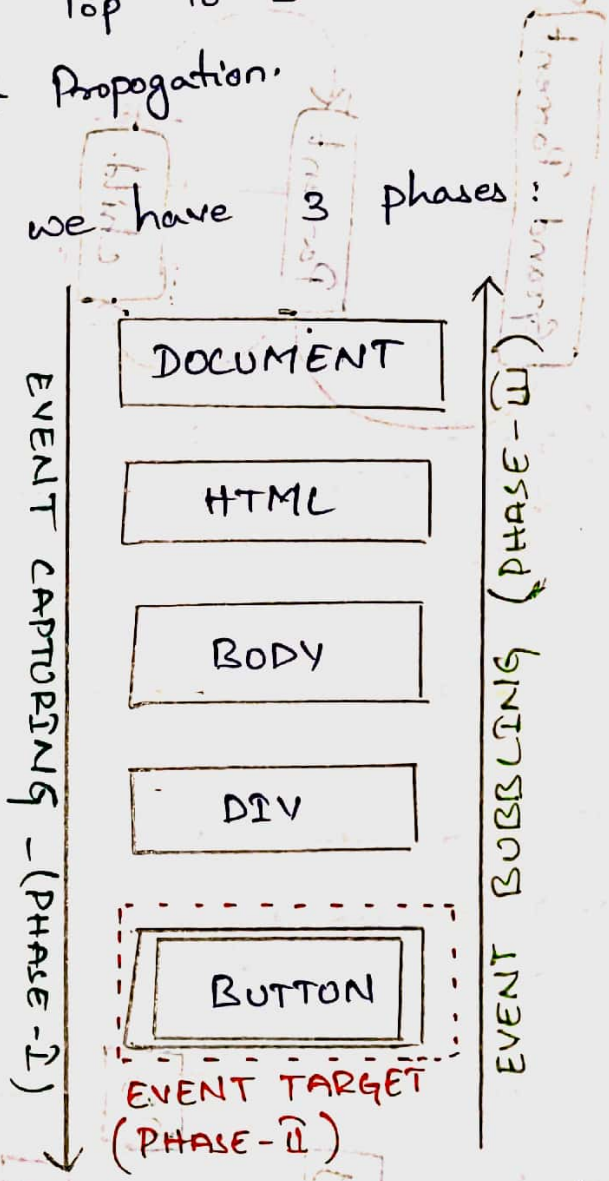
-: EVENT PROPOGATION :-

Event Propogation in Javascript is a Mechanism that dictates how events travel through the Document Object Model (DOM) when an event occurs

→ flow of events from Top to Bottom and Bottom to Top is known as Event Propogation.

→ In Event Propogation we have 3 phases :

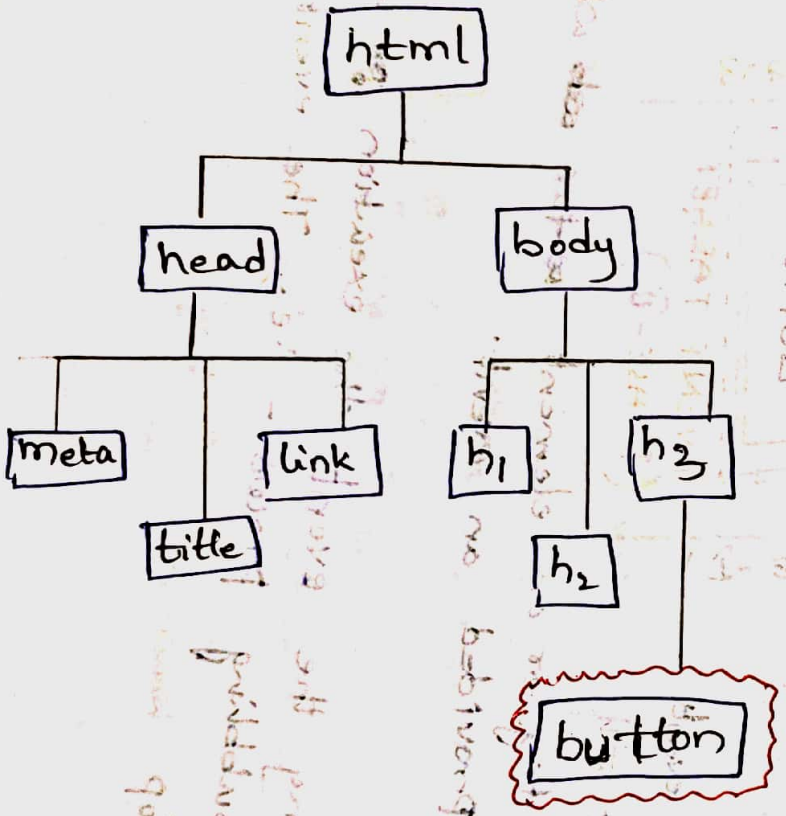
I. Event Capturing
II. Event Target
III. Event Bubbling

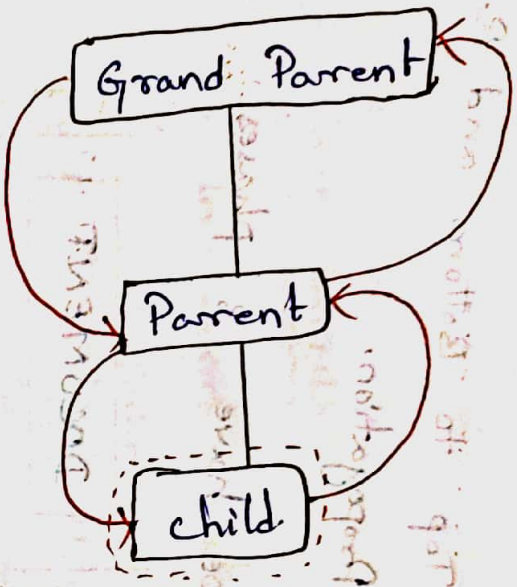


I. Event Capturing :-
→ Event flow from top to bottom is known as Event Capturing.

II. Event Target :-
→ After flow of events from Top to Bottom, It will target a particular tag or an element for which we have already provided an event.

III. Event Bubbling :-
→ After targetting the event the execution will start the Event bubbling proces - i.e., The event flow from Bottom to Top known as Bubbling.





-! Event Handler !-

To handle the event or flow of event from top to bottom and bottom to top. We can use addEventListner() method.

Syntax!-

element.addEventListner(event, function, boolean (or) useCapture)

Example!-

H1.addEventListner("click", () => { }, T/F)

T → Capturing
F → Bubbling

Bubbling:-

Whenever after targetting the tag it will start bubbling (it will target all the parents & grand parents elements as well as root elements)

When we target the child it should not target the parent element, We need to avoid parent element by using stopPropogation() method (or) stopImmediatePropogation() method.

Note:-

In adding a listner method the third argument is not MANDATORY; because it takes BUBBLING: (False) by default.

Example:-

EventPropogation.html:-

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta>
    <title>
    <link>
  </head>
  <body>
    <style>
      #gp{
        border : 2px solid;
        height : 200px;
        width : 200px;
        margin : auto;
      }
      #parent{
        border : 2px solid;
        height : 100px;
        width : 100 px;
        margin : auto;
      }
```

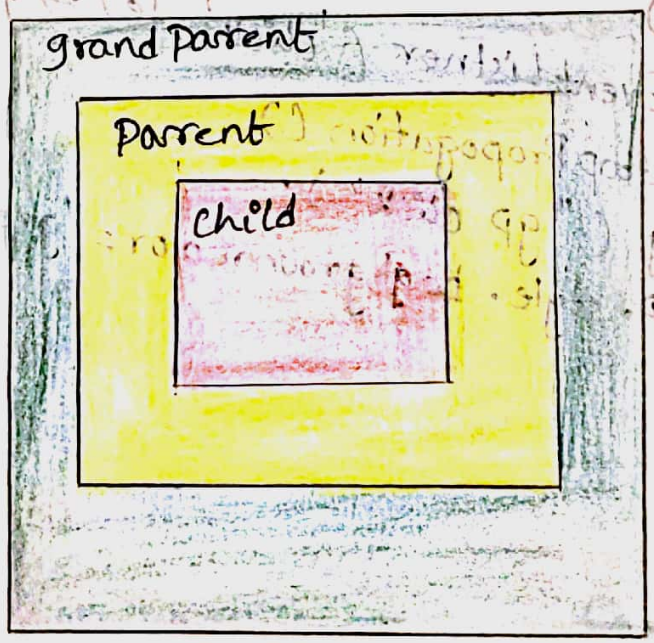
```
#child {
    border : 2px solid;
    height : 50px;
    width : 50px;
    margin : auto;
</style>
<div id="gp"> Grand Parrent
    <div id="parrent"> Parrent
        <div id="child"> child
        </div>
    </div>
</div>
<script src="./event-propogation.js"></script>
</body>
</html>
```

event-Propogation.js :-

```
let gP= document.getElementById("gP")
clg (gP);
gP.addEventListner ('click', (e) => {
    e.stopPropogation()
    clg ("gP click");
    gP.style.backgroundColor="green"
})
```

```
let Parrent = document.getElementById("parent")
clg (parrent);
Parrent.addEventListner ('click', (e) => {
        e.stopPropogation();
        clg ("Parent click");
        Parrent.style.backgroundColor = "yellow"
})

let child = document.getElementById("child")
clg (child);
child.addEventListner ('click', (e) => {
        e.stopPropogation()
        clg ("child click");
        child.style.backgroundColor = "red"
})
```

Output:-



:- EVENT DELIGATION :-

Event Deligation is a pattern based upon the concept of event propogation (bubbling).

→ It is an event handling pattern that allows you to handle event at a higher level in the DOM Tree other than the level where the event was first received.

Example :-

. EventDeligation.html :~

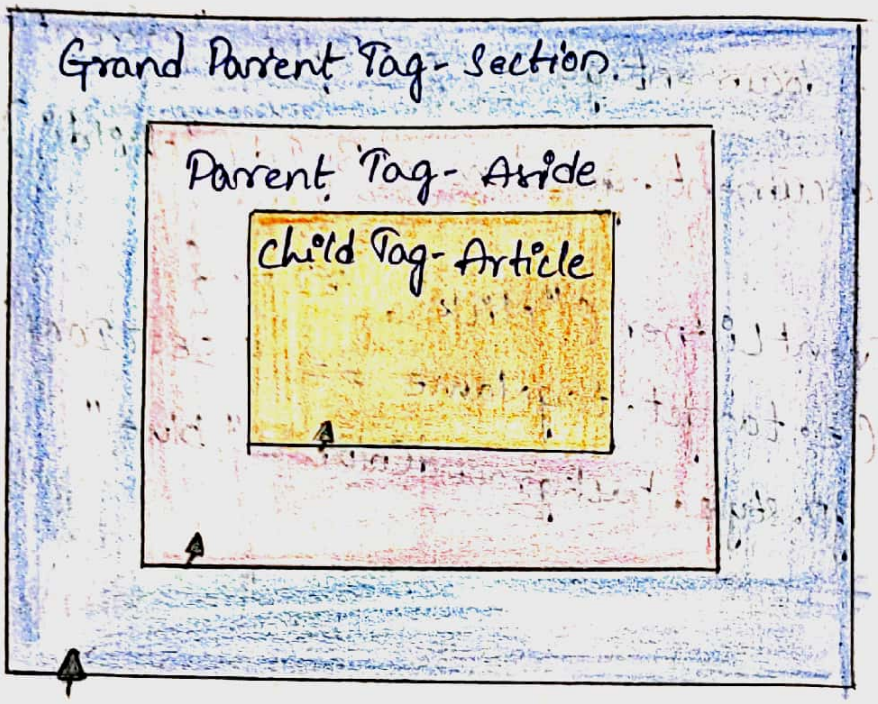
```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta>
    <title>
    <link>
  </head>
  <body>
    <style>
      #gp {
          border : 2px solid;
          height : 300px;
          width : 300px;
          margin : auto;
      }

      # parent {
          border : 2px solid;
          height : 200px;
          width : 200px;
          margin : auto;
      }
```

```
#child {
    border : 2px solid;
    height : 100px;
    width : 100px;
    margin : auto;
}
</style>

<section id="gp">
    Grand Parent Tag - Section
    <aside id="parent">
        Parent Tag - Aside
        <article id="child">
            child Tag - Article
        </article>
    </aside>
</section>

<script src="./Event Deligation.js"></script>

</body>
</html>
```

Event Deligation :JS

```
let gp = document.getElementById("gp")
let parrent = document.getElementById("parrent")
let child = document.getElementById("child")

gp.addEventListner("click", (e) => {
    if (e.target.tagName == "SECTION") {
        gp.style.backgroundColor = "blue"
    }
})

Parent.addEventListner("click", (e) => {
    if (e.target.tagName == "ASIDE") {
        Parent.style.backgroundColor = "pink"
    }
})

child.addEventListner("click", (e) => {
    if (e.target.tagName == "ARTICLE") {
        child.style.backgroundColor = "orange"
    }
})
```

Output :-



~: ASYNCHRONOUS :~

A function which gives a way to another function to execute is known as Asynchronous.

→ To make asynchronous we need to use Two methods:

1. setTimeout()
2. setInterval()

→ Both the methods will accept two arguments.

I. Call back function

II. Delay time

1. setTimeout() - Example :-

```
function f1 (a, b) {
    setTimeout (() => {
        for (let i=a ; i<=b ; i++) {
            clg (i);
        }
    }, 5000)
}
```

5 Seconds written in Milli seconds.

arg1 arg2

```
f1 (5, "10a")

function f2 () {
    clg ("Hello Web");
}
f2()
```

Outputs

Case i:- f1() having Numbers as two arguments.

f1(1,5)

Hello Web
1
2
3
4
5

Case ii:- f1() having one as number and second as string

f1(5, "10a")

Hello Web
>
// continuously keep on waiting ....
// Here It Doesn't give any error -> "Dev gave wrong ip"
// It is a Disadvantage
// TO overcome we go to
- PROMISE -

2. setInterval() - Example :-

```
function d1(a,b){
    setInterval(()=>{
        for(let i=a; i<=b; i++){
            clg(i);
        }
    },5000)
}
d1(5,"10a")
d2
function d2(){
    clg("hi setInterval()");
}
d2()
```

Outputs :-

Case i) d1() having both the arguments as Numbers :-

```
d1(1,5)
hi setInterval()

1
2
3
4
5
1
2
3
4
5
1
2
3
4
5
```

// After Every 5 seconds d1() executes. But d2() executes only one time i.e., first time.

Case ii) :- d1() having one argument as number and other as string.

```
d1(5,"10a")
hi setInterval()
>
```

// continuously keep on waiting....
// Here It doesn't give any error even when dev gives wrong input.
// It is the disadvantage
// TO overcome this we go to
— PROMISE —

Note:-

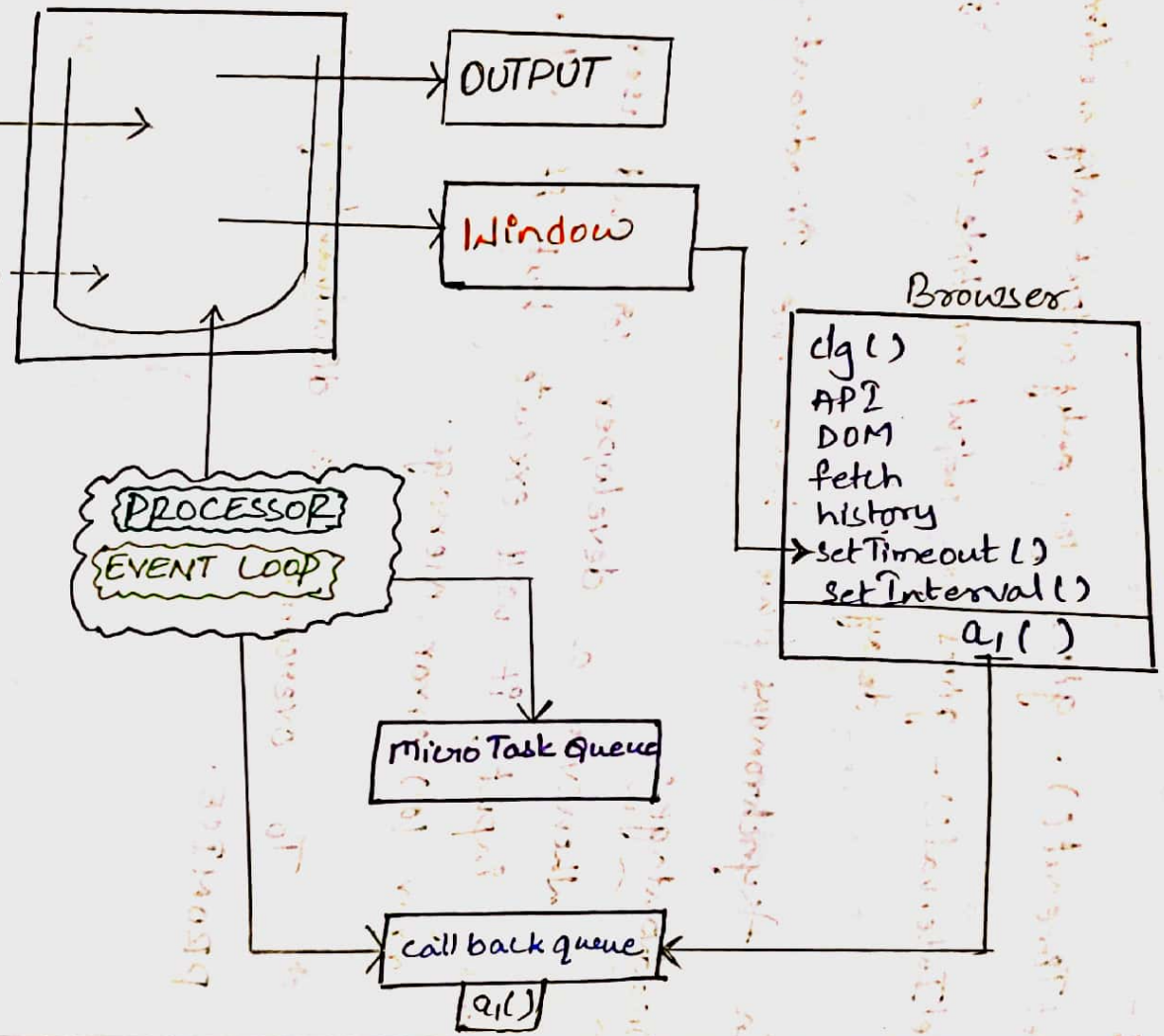
* setTimeout() - gives you output only one time.
* setInterval() - gives you output multiple times after the delay time.
* In Asynchronous we have one Disadvantage.

Disadvantage:

* Whenever a developer (or) end user gives a wrong input, it will execute but it will not give any error (or) error message.
* To overcome this disadvantage we go to PROMISE.

~: HOW ASYNCHRONOUS WORK INTERNALLY :~

```
function a1(){
    setTimeout(()=>{
        // 20 min
    }, 5000)
}

a1()

function a2(){
    clg("Asynchronous working Proves");
}

a2()
```



Event loop :-

First checks the JS engine. If the JS engine is empty. It'll go to mtq - if mtq is also empty then, it will go into call back queue.

~: PROMISES :~

* It is an object.
* Promise object keeps an eye on the asynchronous task given.
* If the asynchronous task is not yet completed successfully then promise is considered as Pending.
* If the asynchronous task is completed successfully then promise is considered as resolved state. (or) resolve state.
* If the asynchronous task is completed but not successful then it is considered as reject state.

Syntax:-

```
new promise ((resolve, reject) => {
    setTimeout (() => {
        // statements
    }, 5000)
})
.then (() => { })
.catch (() => { })
.finally (() => { })
```

Example:-

```
function demo1 (a,b) {
    return new promise ((resolve, reject) => {
        setTimeout (() => {
            if (isNaN(a) || isNaN(b)) {
                reject ()
            }
            else {
                resolve ()
            }, 5000)
        })
    .then (() => {
        for (let i=a ; i<=b ; i++) {
            clg (i);
        }
    })
    .catch (() => {
        clg ("error input");
    })
```

```
.finally ( () => {
        clg ("finally method Executes for sure");
    })
}
demo1 (1,5);

function demo2 (){
    clg ("Hello Web");
}
demo2 ();
```

Promise → Advance Topics
* Promise all
* Promise any
* Promise race
* Promise all setteled

. then ( ) :
* It can accept a call back function.
* The call back ( ) function passed to .then ( ) method - gets executed only when the promise return resolve ( ) method.

. catch ( ) :
* It can accept a call back function.
* The call back ( ) function passed to .catch ( ) method - gets executed only when the promise return reject ( ) method.

. finally ( ) :
* It does not care about reject ( ) (or) resolve ( ), what ever the things you write in .finally ( ) method block gets executed irrespective of .then ( ) (or) .catch ( ).

~ : PROMISE CHAINING :~

A function with promise having Multiple .then()'s and multiple .catch()'s is known as promise chaining (or) the multiple .then()'s will be executed is known as promise chaining.

Note:-

Multiple .then()'s gets executed; but only one .catch() gets executed single time.

-: ARRAYS :-

Def:-

Array is a block of memory which is used to store multiple values of any type.

Arrays in javascript is Hetrogenious.

You can store any kind of data in an Array.

Ex:-

```
Var arr = ["hi", 123, true, Naal, Null, 12.5, undefined]
```

Array Creation :-

i, In Javascript Array is an object

ii, We can create array object in 3 ways

(a) using Array literal

(b) using new operator

(c) using Array constructor

(a) using Array Literals :-

Syntax :- Let arr = [ ]

Here arr is the variable which store the address of array object.

(b) Using new Operator :-

Syntax :- Let arr = new Array ( );

* By creating an instance of Array using 'new' operator.
* Here parenthesis "( )" means function call
* "Array( )" (constructor). Array function Constructor.
* "Array" - Anything starts with uppercase is known as class.

* () is constructor function which is having same name as class.

* 'new' is a keyword which helps in creating new object.

Note!-

* When an Array object is created but no data is present in it.

Ex!-

```
let arr2 = new Array(10, 20, 30);
clg(arr2);
```

op!- [10, 20, 30]

(c) using Array() Constructor :-

Syntax!-

```
let arr = Array();
```

We can create array object by using only Array() without using 'new' keyword

Ex!-

```
let arr3 = Array(10, 20, "Hi");
clg(arr3);
```

op!- 10, 20, Hi

Note:-

How to Access an Array Element ?

We can access Array Element with the help of array object reference, array operator, array variable with index

Ex:-

```
let array = ["hi", 123, true, Nan, null, 12.5, undefined]
clg(arr[2]);
```

op:- true

Array Methods :-

i) push()

ii) pop()

iii) unshift()

iv) shift()

i, Push():-

* The push() is used to add the element at the end of the array (or) at the tail of the array.

* It will accept 'n' number of arguments.

Ex:-

```
let arr1 = [10,20];
arr1.push(40,30);
clg(arr1);
```

op:-

[10,20,40,30]

ii, Pop():-

* The pop() is used to remove element from the end of the array (or) tail of the array.

Ex:-

```
let arr2 = [10,20,30,40];
arr2.pop();
clg(arr2);
```

op:-

[10,20,30]

(iii) unshift ( ) :-

* This method is used to add elements from start of the array (or) Head of Array.

* It accepts n number of arguments.

Ex:-
```
let arr3 = [10, 20]
arr3.unshift (100, 200);
clg (arr3);
```

op:-
```
[100, 200, 10, 20];
```

(iv) shift ( ) :-

* This method is used to remove an element from the start of an array (or) head of array.

* It accepts no arguments.

Ex:-
```
let arr4 = [40, 30, 10, 20]
arr4.shift ( );
clg (arr4);
```

op:-
```
[30, 10, 20]
```