① 2023-09-25
② 2023-sep-25
③ 25-sep-2023.

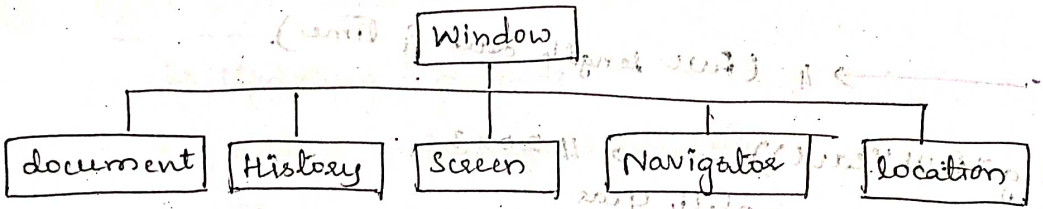
} Laptop [Date method]

27/9/23

BOM

→ stands for Browser object model.

- When we execute HTML file in the browser, browser acts as an object and the object name is BOM
- Bom is also known as Window object
- Bom is a global object



DOM

→ stands for Document Object model.

- When we execute HTML file in the browser, the browser converts HTML file to a tree like structure.
- Tree like structure is known as Dom
- Elements section in the browser development tools represents Dom
- Dom is used to manipulate HTML through Javascript.
- By using DOM we can create new HTML elements, can delete the existing HTML elements, can update existing HTML elements, can add HTML elements etc
- By using DOM we can create new HTML attributes, can delete existing HTML attributes, can update existing HTML attributes, can add HTML attributes etc

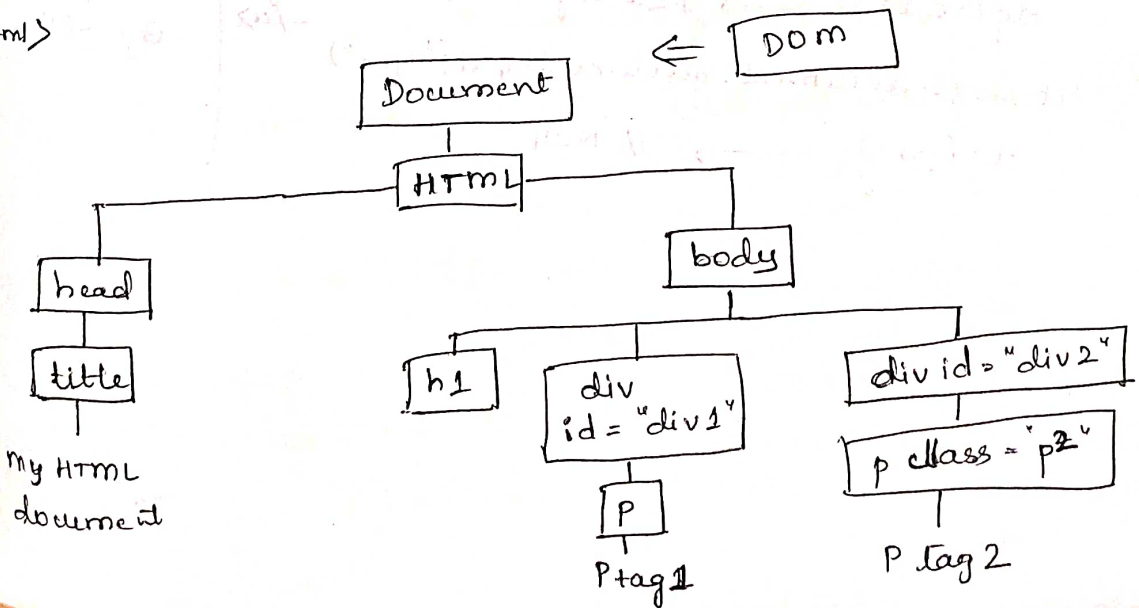
- using DOM

- Every tag in the DOM is called as Node of the tree.
- Every Node (HTML tag) in DOM tree is considered as an object.
- Attributes of the tags are considered as properties of the object.
- whatever is available on DOM tree, that will be displayed on the webpage.
- In DOM tree, Document object is a parent object.

HTML document ⇒

Body tag is the parent Node.
All the inner tags are child node.

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



DOM methods to get HTML elements

1) document.getElementById("elementId")
2) document.~~get~~querySelector("selectors")
3) document.querySelectorAll("selectors")
4) document.getElementsByClassName("classNames")
5) document.getElementsByName("elementName")

---

① document.getElementById("elementId")

→ Used to get the element based on Id.
→ Accepts one arguement of type string, where the arguement is element's Id.
→ If the Id is available for any element, it returns first occured element.
→ If Id is not available for any element, it returns Null.

```
let res = document.getElementById("demo")
dg(res)  ⟶ // <h1 Id="demo"> Hello</h1>

let res1 = document.getElementById("test")
dg(res1)  ⟶ // <a href="" Id="demo"> Bye </a>

let res3 = document.getElementById("Syed")
dg(res3)  ⟶ // Null
```

HTML

```
<h1 Id="demo">
  Hello </h1>
<p Id="demo">
  Hi </p>
<a href="" Id="~~demo~~test">
  Bye </a>
```

② document.querySelector("selectors")

→ ~~Selects~~ Gets the element based on the selectors

→ It accepts one arguement of type string, where the arguement is selector.

→ If any of the selector is available, it gets the first occured element. Else, returns Null.

```
let res = document.querySelector("a")
dg(res)
let res1. document. querySelector("h1")
dg(res 1)
let res 2. document. querySelector(".a")
dg(res 2)
let res 3. document. querySelector("demo")
dg(res 3) ——— // Null
let res 4. document. querySelector("#a")
dg(res4)
let res 5. document. querySelector("div h2")
dg(res5)
```

HTML

```
<h1 id="demo">
  Hello </h1>
<h1 class="a">
  Heading.</h1>
<p id="demo">
  Hi </p>
<span class="a">
  span </span>
<span id="test">
  JS </span>
<a href="" id="demo">
  Bye </a>
```

③ document.querySelectorAll("selectors")

→ Similar to querySelector, but selects all the occurance.

→ If the selector is available, it returns all the element with the given selector, in the form of Node list [Arraylike object]

→ If the given selector is not available, it returns empty Node list

```
let res = document.querySelectorAll("a")
dg(res) ——— Nodelist [a]
let res1 = document. querySelectorAll("h1")
dg(res1) ——— Nodelist (2) [h1  h1]
dg(res1[0]) ——— <h1> ——— </h1>
```

```
let res2 = document.querySelectorAll("h1")

let res3 = document.querySelectorAll("#a")
clg(res3) ——— Nodelist []
let res4 = document.querySelectorAll("div h2")
clg(res4)
let res5 = document.querySelectorAll("h1")
clg(res5)
```

④ document.getElementsByClassName("classNames")

→ Used to get the elements based on class names.

→ Accepts one arguement of type string, where arguement is class name

→ Returns all the elements with the given classname, in the form of HTML collections [Array like object]

→ If the element doesn't have the classname, it returns empty HTML collection.

```
let res = document.getElementsByClassName("a")
clg(res) ——— HTML collections (2) [h1.a, span.a]
let res1 = document.getElementsByClassName("h1")
clg(res1) ——— HTML collection []
```

8 document.getElementsByTagName('TagName')

- Used to get elements based on tagname.
- Accepts one argument of type string, where the argument is tagname
- It returns all the elements with keyword tagname, in the form of HTML collection.
- If any tag is not available, it returns empty HTML collections.

let res1 = document.getElementsByTagName("h1")
dg(res1) —— HTML collection (2) [h1#demo, h1]

let res2 = document.getElementsByTagName("selection")
dg(res2) —— HTML collection []

dg(document) —— complete HTML structure
dg(document.head) — prints Head section of HTML page
dg(document.body) — prints Body section of HTML page
dg(document.title) — prints title section
dg(document.script) — prints script tag

Above are all unique selectors [They are not repeated]

dg(document.body.h1) —— undefined

document.title = "Hello Javascript"
↳ re-initializing title in DOM.

Adding or Getting Attributes

let input = document.querySelector("input")
dg(input)
dg(input.type)
dg(input.minLength)

input.type = "password";
input.size = "80";

using getAttribute()

```
clg(input.getAttribute("type"));
input.setAttribute("max", 200);
```

Adding CSS

→ Adding CSS to HTML through Javascript

⇒ Inline CSS

```
let demo = document.getElementById("demo");
clg(demo).

demo.style.backgroundColor = "aqua"
demo.style.color = "red"
demo.style.fontsize = "50px"
demo.style.fontWeight = "bold".
```

(or)

```
demo.style.cssText = `background-color : aqua; color: red;
    font-size: 50px; font-weight: Bold;`
```

⇒ Internal or external css classes

→ demo.className = "bg-color"
↳ adds a class named bg-color

→ demo.className = "bg-color"
demo.className += " font-design"
} adds two classes
class = "bg-color font design"

→ demo.classList.add("bg-color")
demo.classList.add("font-design")
demo.classList.remove("dummy")

→ demo.classList.toggle("dummy")
demo.classList.toggle("bg-color")
clg(demo.classList)
} adds if not present & removes if present.

Event Handlers

→ Event handlers are the special attributes in HTML which is used to handle the events. That means, used to call the functions based on users actions.

→ Event is nothing but function.

→ Event handlers are used to call the functions only when the user performs some actions.

→ Example for Event handlers. [Types of Event Handlers]

① Mouse Events
② Keyboard Events
③ Input Events
④ Button Events etc

Mouse Events

① onclick : A user clicks on an element.
② oncontextmenu: A user right-clicks on an element.
③ ondblclick : A user double-clicks on an element.
④ onmousedown : A mouse button is pressed over an element.
⑤ onmouseenter : The mouse pointer moves into the element.
⑥ onmouseleave : The mouse pointer moves ~~over~~ out of an element.
⑦ onmousemove : mouse pointer moves over an element.
⑧ onmouseout : mouse pointer moves out of an element.
⑨ onmouseonto : " ——— " onto an element
⑩ onmouseup : " ——— " ——— " released over an element

keyboard Events

① onkeydown : A user presses a key
② onkeypress : A user presses a key (Only character value)
③ onkeyup : User releases a key

Input Events

① oninput : when the value/content of an input element is changed

② onblur : When user leaves an input field.

③ onchange : when user changes the content of an input fiel
- when a user selects a dropdown value

④ onfocus : when input field is focused.

⑤ onselect : when input text is selected.

Button Events

① onBlur

② onclick

③ onfocus

④ onscroll widget Position

⑤ onkeypress

⑥ onkeyup

⑦ onkeydown.

28/9/23

HTML:- <button onclick = "bgDark" > Dark </button>

<button onclick = "bgLight" > Light </button>.

TS :-

```
function bgLight() {
    document.body.style.backgroundColor = "white"
}

function bgDark() {
    document.body.style.backgroundColor = "black"
}
```

Toggling password using checkbox

JS

```
function togglePassword() {
    let pwd = get document.getElementById("pwd");
    if(pwd.type === "password") {
        pwd.type = "text"
    }
    else {
        pwd.type = "password"
    }
}
```

(method-01)

```
<input type="password" id="pwd">
<input type="checkbox" id="i" onclick="togglePassword()">
<label for="i"> show Password </label>
```

(Or)

(method-2)

```
function togglePassword() {
    let pwd = document.getElementById("pwd")
    let i = document.getElementById("i")
    if(i.checked === true) {
        pwd.type = "text"
    }
    else {
        pwd.type = "password"
    }
```

(Or)

```
function togglePassword() {
    let pwd = document.getElementById("pwd");
    let i = document.getElementById("i");
    if(pwd.type === "password") {
        pwd.type = "text";
        i.className = "fa fa-eye-slash"
    }
```

```
else {
    pwd.type = "password";
    i.className = "fa fa-eye";
}
```

```
<input type="password" id="pwd">
```

→ `<i class="fa fa-eye" onmousedown="togglePassword()" aria-hidden="true" onmouseup="togglePassword()" id="i"></i>`

→ `<i class="fa fa-eye" id="i" aria-hidden="true" onclick="togglePassword()"></i>`

→ `<i class="fa fa-eye" id="i" aria-hidden="true" onmouseover="togglePassword()" onmouseout="togglePassword()"></i>`

To get the content of Element

```
let i = document.getElementById("i");
clg(i.innerText)
clg(i.textContent)
clg(i.innerHTML)
```

```
<div id="i">
    hello I am div
    <h1> Welcome </h1>
        <Span> I am Span </Span>
</div>
```

↳ `clg(i.innerText)` ⇒

hello I am div
Welcome
I am Span

} only the text is extracted

↳ `clg(i.textContent)` ⇒

hello I am div
Welcome
I am Span

} Text, breaks and the spaces are extracted.

↳ dg (i.innerHTML) ⇒ hello Iam div
<h1> Welcome </h1>
 Iam Span

extracts Texts, line breaks spaces as well as tags.

1). innerText :- Used to get the inner text only. as well as re-initialize the text to the element

i.innerText = "<h1>Syed Imraam </h1>"

↳ <h1> Syed Imraam </h1>

2). textContent :- Used to get line-breaks, spaces and innertext. Also re-initialize all the above

~~i.inn~~ i.textContent = "<h1> Syed Imraam </h1> "

↳ <h1> Syed Imraam</h1>

3). innerHTML :- Used to get line-breaks, spaces, tags and innertext. Also re-initialize all.

i.innerHTML = "<h1> Syed Imraam </h1> "

↳ Syed Imraam

JS

```
function tiger() {
  let img = document.getElementById("image");
  img.src = "_____"
}
```

30/8/23

Event Bubbling and Event Capturing

Event Bubbling :- [Inner Event to Outer Event]

→ It is a process of executing inner event first then the outer event and so on.

→ By default event handlers perform Event Bubbling.

HTML

```
<div class="g-p" onclick="gp()">
  <div class="p" onclick="p()">
    <div class="c" onclick="c()">
      <div class="g-c" onclick="gc()">
      </div>
    </div>
  </div>
</div>
```

JS

```
function gp() {
    alg("Grand Parent");
}
function p() {
    alg("Parent");
}
function c() {
    alg("child");
}
function gc() {
    alg("Grand child");
}
```

Adding Event listeners through Javascript

. addEventListener is a method which is used to add the event handlers through Javascript.

→ It Accepts 3 arguements, where 1st arguement is Event handler type. 2nd arg is function to be executed (function structure or call-back function). 3rd arg is Boolean value [False by-default].

~~Event Bubbling~~ → Event handlers to be passed without prefix on.

→ 2nd arg → function structure

Event Bubbling through Javascript

HTML

```
<div class="g-p">
  <div class="p">
    <div class="c">
      <div class="g-c">
      </div>
    </div>
  </div>
</div>
```

JS

```
let grand_parent = document.
        querySelector(".g-p");
let parent = document.querySelector(".p");
let child = document.querySelector(".c");
let grand_child = document.querySe-
    -ctor(".g-c");

function gp() {
  alg("Grand Parent")
}
grand_parent.addEventListener("click", gp, false)

function p() {
  alg("Parent");
}
parent.addEventListener("click", p, false)

function c() {
  alg("child");
}
child.addEventListener("click", c, false)

function gc() {
  alg("Grand child")
}
grand_child.addEventListener("click", gc, false)
```

Event Capturing :

→ It is a process of executing outermost event first and then the innermost event and so on.

→ Event capturing can be performed by using .addEventListener()

→ If we pass 3rd arguement as true, to the .addEventListener() it performs Event capturing.

JS

```
let grand_parent = document.querySelector("g-p");
let parent = document.querySelector(".p");
let child = document.querySelector(".c");
let grand_child = document.querySelector("g-c");

function gp() {
    alg("Grand Parent");
}
grand_parent.addEventListener("click", gp, true);
function p() {
    alg("parent");
}
parent.addEventListener("click", p, true);

function c() {
    alg("child")
}
child.addEventListener("click", c, true);
function gc() {
    alg("Grand child")
}
grand_child.addEventListener("click", gc, true);
```

HTML

```
<button id="btn"> count </button>
```

JS

```
let btn = document.getElementById("btn")
let count = 0;
btn.addEventListener("click", function () {
    clg(++count)
})
```

// Used to count when a button is clicked.

Remove Event Listener

.removeEventListener() is used to remove the event handlers from the given Element

→ If you want to remove the event handlers, we should pass the same function reference to the .removeEventListener().
[The function reference of .addEventListener()].

```
<button id="btn"> count </button>

<button onClick="removeCounter()"> Remove </button>
```

JS

```
let btn = document.getElementById("btn")
let count = 0;

function counter() {
    clg(++count)
}

btn.addEventListener("click", counter)   // same reference

function removeCounter() {
    btn.removeEventListener("click", counter)   // same reference
}
```

Creating HTML Elements Using Javascript [Using DOM]

document.createElement(), is used to create the new HTML element
It accepts one arguement of type string, where arg is tagname

```
let body = document.body;
let h1 = document.createElement("h1");
clg(h1);
h1.innerText = "Heading";
h1.align = "center";
```

body.append(h1) → adds HTML element at the end of body tag.
body.prepend(h1) → adds HTML element at the start of body tag

---

Creating login Form using DOM [Append & Prepend]

```
let body = document.body;
let form = document.createElement("form");
form.action = "./home.html";

let user = document.createElement("input");
user.type = "text";
user.placeholder = "Username";

let pass = document.createElement("Input");
pass.type = "password";
pass.placeholder = "password";

let btn = document.createElement("button");
btn.type = "submit";
btn.innerText = "login";
```

```
let hr1 = document.createElement("hr");
let hr2 = document.createElement("hr");

body.prepend(form);
form.append(user, hr1, pass, hr2, btn);
```

clg(form.firstChild) // <input type="text" placeholder="username">
↳ first child element will be fetched

clg(form.lastChild) // <button type="submit">Login</button>
↳ used to fetch last child element.

clg(form.children) // HTML collections (5) [input, hr, input, hr, button]
↳ displays all the tags in HTML collection

use [array.from] to convert to pure array

clg(form.children[1]) // <hr>
↳ used to fetch using index position

#Append() :-

→ Used to add HTML elements or strings at the end of the parent tag.

→ Append() accepts multiple argument, where the arguments can be HTML tag, string or any kind of data.

→ Append() doesnot return anything / any value

#Prepend()

→ Used to add HTML elements or strings at the start of parent tag.

→ Prepend() accepts multiple argument, where the arguments can be HTML tag, string or any kind of data.

→ Prepend() doesnot return any value

Append Child ()

→ Used to add single HTML element at the end of parent element.

→ It accepts one arguement, where the arguement is HTML Tag

→ It returns the added element.

```
let div = document.getElementById("div")
let h1 = document.createElement("h1")
h1.innerText = "Heading one".
let h2 = document.createElement("h2")
h2.innerText = "Heading TWO".

clg(div.append(h1,h2))
clg(div.appendChild(h1,h2))

div.append("Hello Bangalore")
div.appendChild("Hello Bangalore")
```

Tables → Row data → for of method
→ for in method
→ map method

```
if (productsList === 0) {
    clg(productsList)
}
```
→ [Bad Practise]

```
if (!productsList) {
    clg(productsList)
}
```
✓ [good practise]

[Truthly & Falsy]

01/10/2... Web storage in browser

1) Local storage:

→ It is a builtin storage in a browser which allows us to store the data inside the browser

→ It is an object of window object.

→ When the browser is closed, local storage data will remain in the browser [Example: facebook, gmail].

2) Session storage

→ It is a built-in storage in a browser which allows us to store the data inside the browsers.

→ It is an object of window object.

→ Session storage's data will be deleted from the browser once the browser is closed.

Note: The size of local and session storage ranges from 5mb to 10mb. [Depends on browser as well]

Methods

① .setItem() :— Used to add (key and value) an item

```
localStorage.setItem('sname', "syed")
localStorage.setItem('age', 31)
localStorage.setItem('dept', "CSE")
```

② .removeItem() :- Used to remove the item

```
localStorage.removeItem("age")
```

③ .clear() :— Used to clear the entire section

```
localStorage.clear()
```

* cg(localStorage.length)
* cg(localStorage.key(2))
* cg(localStorage.key(1))

same methods from both
→ local storage
→ session storage

How to store an object in local storage | Session storage | HTML file

```
let student = { name : 'syed',
                age : 31 }
```

} cannot store an object directly !!

```
document.body.innerHTML = `<h1> ${JSON.stringify(student)} </h1>
```
[HTML]

convert it to JSON object.

```
localStorage.setItem("student", JSON.stringify(student))
```
[In local / session storage]

to get it back ⟶ Need to convert it to pure object again

```
clg (JSON.parse(localstorage.getItem("student")))
```

Timing Functions in Javascript

setTimeout() :- used to call another function after some span of time.

clearTimeout()

→ It is calls the function only once.

```
setTimeOut (() ⇒
    {
      clg (" will be called only once after some span of time")
    }, 2000);
```

→ It accepts 2 arguements, where the first arguement is a call-back function to be executed after some span of time. And the second arguement is time in milliseconds.

setInterval() :- used to call another function for every span of time.
↓
clearInterval()

→ It accepts 2 arguement, where 1st arg is a call-back function to be executed after for every interval.

2nd arg is time in milliseconds.

```
setInterval (() =>
    {
      clg ("Will be executed at certain interval of time")
    }, 2000);
```

Example for setTimeOut()

HTML

```
<div> Registred Succesfully </div>
<button onclick="submitMessage()">
    Register </button>
```

JS

```
function submitMessage() {
  let div = document.querySelector
                ("div");
  div.style.display = "block";
  setTimeOut() => {
     div.style.display = "None";
  }, 5000)
}
```

Counter [Start, Stop, Reset]

JS

```
let count = 0;
let span = document.getElementById("counter")
let counterInterval;

function StartCounter() {
    counterInterval = setInter() => {
          span.innerText = ++count
    }, 1000)
}

function stopCounter() {
    clearInterval (counterInterval);
}

function resetCounter() {
        count = 0;
        span.innerText = count
    }   clearInterval (counterInterval)
```

Start counting

Stop counting

Reset counting.

HTML

```
<h1> counter : <span id="counter"> 0 </span> </h1>
<button onclick="startCounter()"> Start </button>
<button onclick="stopCounter()"> Stop </button>
<button onclick="resetCounter()"> Reset </button>
```

02/10/23

```
clg("start")
setTimeout(()=>
  {
    clg("timeout 1")
  }, 2000)
setTimeout(()=>
  {
    clg("timeout 2")
  }, 3000)
setTimeout(()=>
  {
    clg("timeout 3")
  }, 2000)
setTimeout(()=>
  {
    clg("timeout 4")
  }, 1000)
clg("End")
```

Start
End
timout4
timout3
timout2
timout1

```
clg("start")
setTimeout(()=>
  {
    clg("timeout 1")
  }, 0)
clg("End")
```

Start
timout1
End

Start
End
timeout1

setTimeout will also display after clg statement.

this keyword

→ when passed as an argument in the function and assigned to an event handler in an element, it refers to element itself

→ when passed in object, it refers to object.

```
function m1(a,b)
{ clg(arguements)                    // Arguements(2) ['syed',... ]
  clg(arguements.length)             // 3
  clg(this)                          // window { object }
}
m1("syed","hello","hi")
clg(this)                            // window { object }
function slopTimeout(i)
{ clg(i)
}
```

Hoisting

→ It is the process of moving variable declarations and function declaration to the top of their scopes.

→ There are two types of hoisting.

① Function hoisting.

② Variable hoisting.

Function Hoisting

→ It is the process of moving function declaration to the top of its scope, before execution of the program.

→ It is very useful and it is used in many ~~attractions~~ scenario.

```
m1()

function m1(){                  } Is hoisted
  clg("m1")
}

m1().
```

→ It's advantage is that, we can call the function before its declaration

Variable Hoisting

> Note:
> only Named functions will be hoisted

```
# clg(m1)  ------------------------------> // undefined
var m1 = function () {
    clg("m1")
}
m1()  ------------------------------> // m1
```

```
# clg(m1);  ------------------------------> // error.
let m1 = function () {
    clg("Anonymous")
}
```

```
# clg(m1);  ------------------------------> // error.
const m1 = function () {
    clg("Anonymous")
}
```

```
# clg(m1);  ------------------------------> undefined.
var m1 = function () {
    clg("Anonymous")
}
```

Variable Hoisting

→ It is the process of moving variable declaration to the top of its scope, before execution of the program.

→ Only var type of variables can be hoisted.

→ let and const type of variables can't be hoisted.

→ If we try to access the variable of type const and let before its declaration, we get an error.

→ If we try to access the variable of type var before its declaration we get undefined.

```
# dg(a) ⟶ // undefined
var a = 20;
dg(a) ⟶ // 20.
```

```
# dg(b) ⟶ // undefined
var b = "Hello"
dg(b) ⟶ // Hello
```

```
# dg(c) ⟶ // error
let c = "Bye"
dg(c) ⟶ // Bye
```

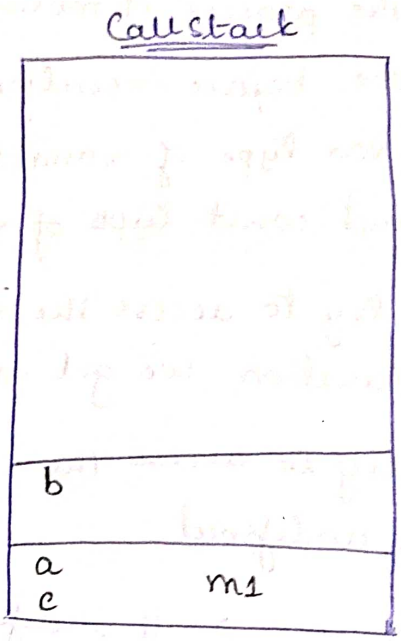
```
# function m1()
{
  dg(a); ⟶ // error : cannot access 'a'
  let a = 20;
  dg(a); ⟶ // 20

  dg(b); ⟶ // undefined
  var b = "Hello";
  dg(b); ⟶ // Hello.
}
m1()
```

#) Closure

```
let a = 20;

function m1() {
  let b = 40;
  clg(b);
}

let c = 70;

m1();
```

Phases

* Creation Phase
* Execution Phase.



Global Execution content

→ Creation Phase :— All the global scope identifiers / function are assigned memory location.

a = undefined
c = undefined
m1 = function str.

→ Execution Phase :— All the identifiers will be initialized.

a = 20;
c = 70;

m1 function will be called m1()

Function Execution Content

→ Creation Phase :— identifiers will be declared.

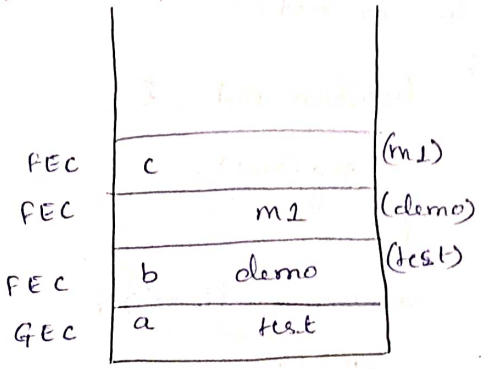
b = undefined.

→ Execution Phase :— b = 40;

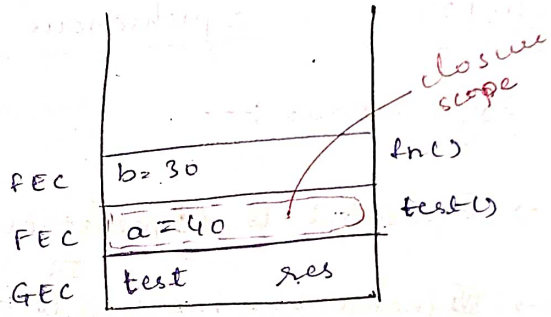
function pops out,
control is back to m1()

```
let a = 40;
function test() {
  let b = 60;
  function demo(s) {
    clg(s);
    function m1() {
      let c = "Hi";
      clg(c);
    }
    m1();
  }
  demo();
}
test();
```

Call stack



```
function test()
{
  let a = 40;
  return function ()
    { let b = 30;
      clg(a+b);
    }
}
let res = test()
res()
```



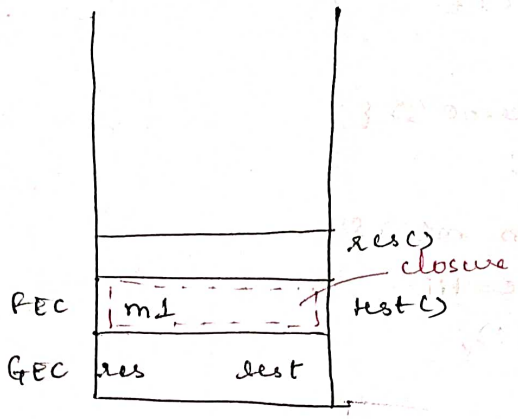
// 70

closure :

→ The inner function is having access to outer function variables and functions even after the outer function is popped out from the stack.

→ When the outer function is popped out from the stack, a copy a outer function will be created in the stack, only when the inner function is having access to outer function's variables and function, and the copy is known as closure.

```
function test()
{
    function m1(){
        clg(m1);
    };
    return function(){
        m1();
    };
};
var res = test();
res();
```



03/10/23 **Synchronous & Asynchronous**

Synchronous :—

→ Javascipt is Synchronous and single treaded.

→ Synchronous means, JS makes other statements to wait till it executes the previous statements. That means, it blocks the statements till the previous statement executes.

→ Javascipt is single Treaded because it peeforms one task at a time.

Asynchronous :—

→ Asynchronous means, doesnot make other statements to wait. That means, JS will allow next statement to execute only if the previous statements are in waiting list.

→ To make Javascipt Asynchronous, we use call back function, promises and Async await

* Call back() → function passed as an arguement
* Promises → object
* Async & Await → keywords.

Call back function :—

→ It is the function that is passed as an argument to another function and is used to handle Asynchronous type of data.

→ We don't prefer call back() because it creates Call back hell, which is very difficult to maintain & understand.

→ Call back hell is nothing but the nested call backs.

Promises :—

→ Promise is a built-in object in Javascript which is used to handle asynchronous type of data.

→ Promise constructor accepts one argument, where the argument is a call back function.

→ Call back function accepts 2 arguement.

1st arg : Resolved function

2nd arg : Reject function.

→ Resolved function will be executed when the promises fulfilled

→ Reject function will be executed when the promises rejected

→ The promises have 3 state, &

1st — Fulfilled or resolved. [response from services]

2nd — Pending.

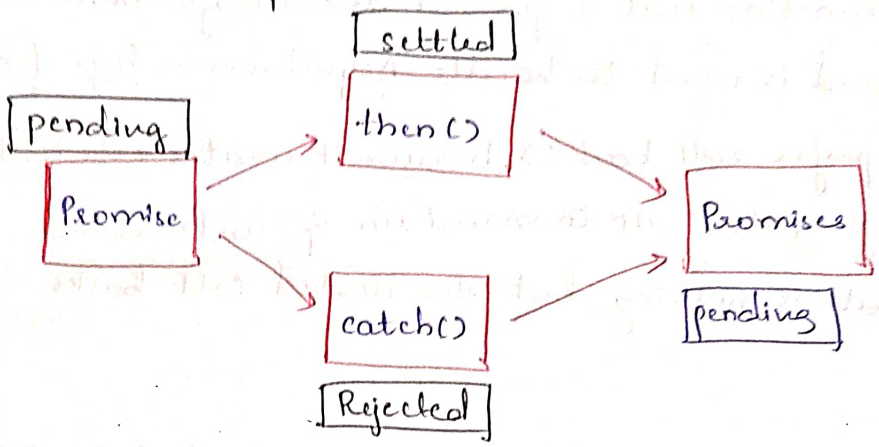
3rd — Rejected. [Errors from services]

→ If you want to handle resolved or fulfilled data, then we use then block or then method [then()]

→ If you want to handle rejected data, we use catch block or catch()

→ When we fetch the APIs using fetch() of Javascript or any other library, then we get promise object as response.

→ Responses can be in pending state (or) rejected state (or) in fulfilled state.




```
let a=60;
let p= new Promise((resolve, reject) =>
   {
      if (a===40) {
        resolve ("students Data")
      }
      else{
        reject ("Error not found")
      }
})
```

→ ({sname: "syed", age: 31})

```
clg (p.sname)
```

→ error

```
p.then((data) => {
    clg (data)
    clg (data.sname.toUpperCase
}).
catch ((err) => {
    clg (err)
})
```

} fetching data

} fetching error

```
p.then((data) => {
    clg(data)
    clg(data.sname.toUpperCase)
    return JSON.stringify(data)
}).
then((d) => {
    clg(d)
}).
catch((eee) => {
    clg(eee)
}).
finally(() => {
    clg("finally always Runs")
})
```

multiple .then() is available.
returned statement of the first then block will be the argument for the second then block

multiple catch() is not available.

No argument

- finally will always run.
- irrespective of promises being fulfilled or rejected.

API Fetching

```
let users = fetch("URL");
users.then((data) => {
    clg(data)                    // Responses.
    // clg(data.JSON())          // Promises.
    return data.json()
}).then((userdata) => {
    clg(userdata)                // Is object [Array]
}).
catch((error) => {
    clg(error)
})
```

Async & Await

→ Async and Await are the keywords in Javascript which is used to handle asynchronous type of data.

→ Async keyword makes the function Asynchronous.

→ Asynchronous function return promises.

→ Await keyword ~~be~~ waits only for the promise but it allows other task or statements to execute.

→ Await keyword must be inside async function ()

→ If you want to handle the erross in asynchronous type of functions we use Try() catch().

→ where try block handles resolved data. And Catch block handle rejected data.

→ Async and await are ~~synch~~ syntactic sugar for promises

```
let a = 20;
let p = new Promise ((resolve, reject) =>
  {
    if (a === 40) {
      resolve ("ok")
    }
    else {
      reject ("error")
    }
})
```

```
async function getData()
{
  try {
    let data = await p;
    clg (data)
  }
  catch (err) {
    clg (err)
  }
}
getdata()
```

Fetching API using Async & Await

```
async function getUsersData() {
  try {
    let users = await fetch("URL");
    users = await users.json()
    users.map(({avatar_url}) => {
    document.body.innerHTML +=
    `<img src=${avatar_url} width=150px>`
  })
  }
  catch(err) {
     alg(err.message)
  }
}
getUsersData()
```