DOM + MODERN JS

Window

A window represents an open window in a browser.

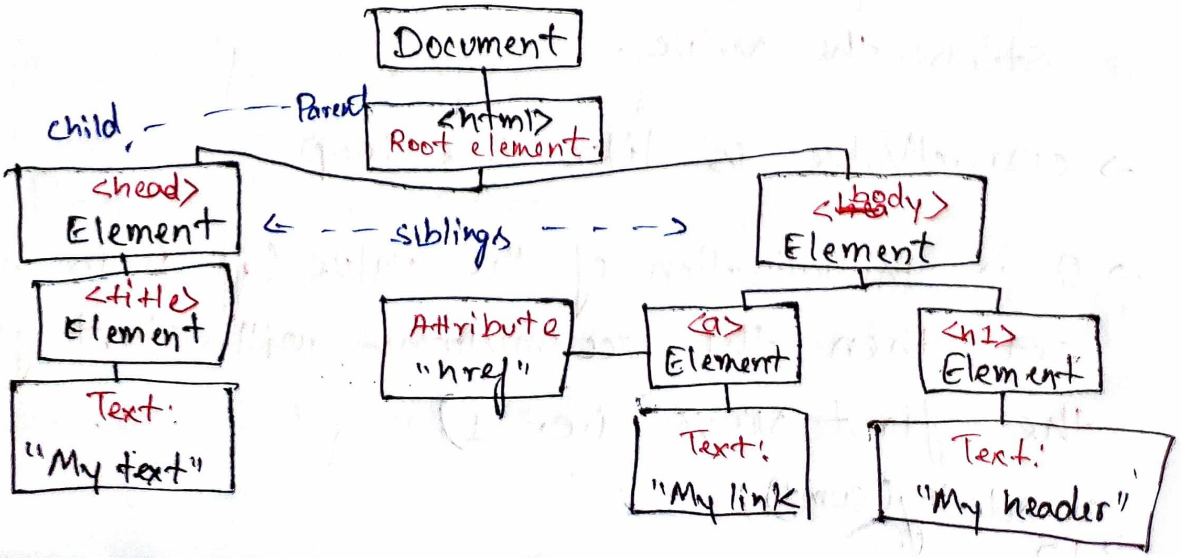
It is a global object.

It is created by the browser

DOM (Document Object Model)

When a webpage is loaded, the browser creates a DOM of the page.

HTML DOM is a tree like structure



The HTML DOM Documents

* The Document object

→ The document object is the root node of the HTML document

→ The document object is a property of the window object

→ The document object is accessed with:

window.document or just document

Converting the whole HTML code into the JS object is called DOCUMENT and this Model is known as DOM.

Rendering of DOM

characters → tags → tokens → nodes → DOM

(tokenizer)

* DOM Methods (to access elements)

(Making changes in HTML using JS)

HTML DOM methods are actions you can perform (on HTML elements)

* The getElementById Method

The most common way to access an HTML element is to use the id of the element

Ex- getElementById('heading')

```
document.getElementById('content');
```

-> It is called on document object.

-> It returns a single object because it works on ID and its unique.

* The getElementsByClassName Method

This method of document interface returns an array-like object of all elements which have all the given class names.

NOTE - Array-like object is not an array.

Ex- document.getElementsByClassName("test");

* The getElementByTagName Method

→ It is similar to the previous method.

→ It also uses document object

→ It also returns multiple lines.

→ The list returned is not an array,

→ we can iterate on that list by using for loop.

Trick

$0 returns the most recently selected element or JavaScript object, $1 returns the second most recently selected one and so on.

* The querySelector Method

NOTE: Use querySelectorAll to get the multiple objects

This method returns the first element within the document that matches the specified selector, or group of selector.

If no match is found then null is returned.

Ex – querySelector('#header') → It is a ID

querySelector('.header') → It is a class

querySelector('header') → It is a tag

* Update Existing Content

- * .innerHTML
- * .outerHTML
- * .textContent
- * .innerText

* .innerHTML

→ It can get or set the ~~HM~~ HTML content within the element

→ It can get element or all of its descendents

→ It can set an element's HTML content

* .outerHTML

The outerHTML attribute of the element DOM interface gets the serialized HTML fragment describing the element including its descendants.

* .textContent

The textContent property sets or returns the text content of the specified node, and all its descendents.

Inner~~Text~~ HTML vs Textcontent

In inner~~Text~~ HTML if there is a tag inside a tag then it will be rendered but in case of textContent it will be treated as the normal text.

Ex

 SWE

⇒ In case of inner~~Text~~ HTML the 'SWE' will be bold. **SWE**

⇒ In case of textContent will be printed as it is without bolding the SWE.

 SWE

* .innerText

It is similar to .textContent with some difference.

Ex -

<> —— <>

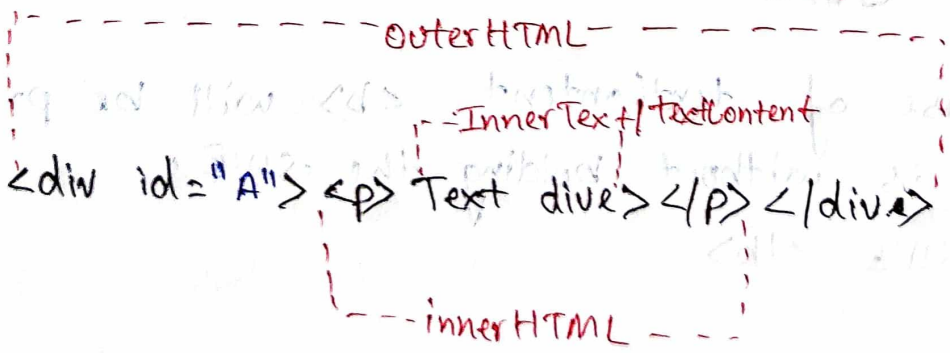
<display:hidden> —— <>

⇒ In case of text content the selected part will be printed.

⇒ But if we use innerText the selected part will be printed except the display: hidden line.

⇒ If the display: hidden property is used then it will not appear when we use innerText.

Visual Representation



* Adding new content / element

To add a content we need to first create the content.

* .createElement()

This method creates the HTML element specified by tagName.

Syntax

```
createElement(tagName)
```

=> .appendChild()

-> To add the content we use .appendChild().
-> It puts the content at the last position in the element.

* Creating a Text Node

Its a long way

```
let newPara = document.createElement('p');
let textPara = document.createTextNode('Hello jee');
newPara.appendChild(textPara);
content.appendChild(newPara);
```

 ⟶ Hello jee

Short way

```
let myPara = document.createElement('p');
myPara.textcontent = 'Hello jee';
content.appendChild(myPara);
```

From the above tag we can only add siblings at the last position only.

To add the sibling at the specified position we use below method.

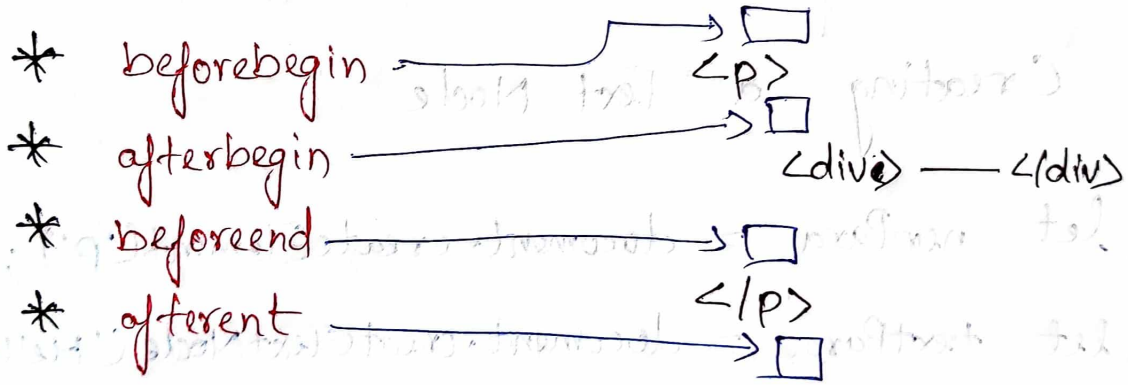
* .insertAdjacentHTML()

It has to be called with 2 arguments

i) location / position → where

ii) HTML text / content → what

There are 4 positions in it:



* Deleting an Element / Content

⇒ .removeChild()

→ It is opposite of .appendChild()

→ We must know its parent element.

→ We must know the child element which we are removing.

Syntax

```
parent.removeChild(cElement);
```

Deleting a child element without knowing its parent element.

Here, Parent = childElement.parent

```
child.parent.remove(child);
```

* Making changes in CSS using JS
  * .style → you can make single change only
  * .cssText
  * .setAttribute
  * .className
  * .classList

  → you can make multiple changes at a time.

Ex

```
let content = $0;
```

→ `content.style.color = 'red';`

→ `content.cssText = 'color:red; font-size:4em;';`

→ `content.setAttribute("style", "color:red; background-color: white;");`

* .className

→ It sets or returns an element's class attribute.

→ It returns all the class names as a string and to make change you'll have to convert it into an array.

* .classList

→ It returns the CSS classnames of an element.

→ It returns in the array format of 'object' type.

Features

add() – to add a class

remove() – to remove a class

toggle() – if class is present then it will delete it if not present then it will add it.

contains() – to check whether a class class is present or not.

It returns True or false.

Browser Events

* events
* respond to events
* data stored in events
* stop an event
* lifecycle of events

* Events

An event is a signal that something has happened. All DOM nodes generate such signals (but events are not limited to DOM).

Ex - click, submit, mouseover, etc.

→ moniterEvents() is a property by which we can check all the events that are happening at a particular time.

By using the unmonitorEvents() we can turn off the events.

* Event Target

It is a interface (blueprint) implemented by objects that can recieve & may have

listerers for them.

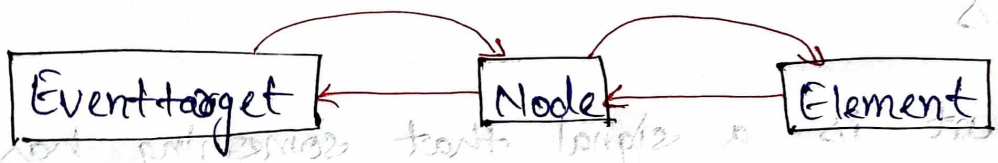
It consist of 3 methods

→ addEventListener()

→ removeEventListener()

→ dispatchEvent()

Event target is a top level interface.



→ Node is inheriting the properties of Event target

→ ~~Event~~ Element is inheriting the properties of Event target as well as Node.

* addEventListener()

→ Listen to a event

→ Respond to event

→ Hook into event

Pseudocode

<event.target>.addEventListener(<event-to-listen-for>,
<function-to-run-when-event-happened>);

⇒ To apply a event Listener we need to have a:

i) event-target → component (document, div, p, article)

ii) event-type → click, scroll

iii) function → defines what to do when event happened

Ex

```
document.addEventListener('click', function() {
    console.log('I clicked!');
});
```

* .removeEventListner()

This method of EventTarget interface removes an event listener previously registered with EventTarget.addEventListener() from the target.

Ex

```
document.removeEventListner('click', eventFunction)
```

It is a function which needs to be defined before execution.

In order to successfully run a removeEventListner() we need 3 conditions to be correct.

i) same ~~target~~
ii) same type
iii) same function

What does same function refer to?

```
document.addEventListner('click', function() {
                console.log('Hi');
});
```
①

```
document.removeEventListner('click', function() {
                console.log('Hi');
});
```
②

① & ② are different function and it will not able to remove eventlistener.

```
function print() {
        console.log('hi');
}
document.addEventListner('click', print);
document.removeEventListner('click', print);
```

It will work perfectly fine because here, the function is same that is being refered.

* .dispatchEvent()

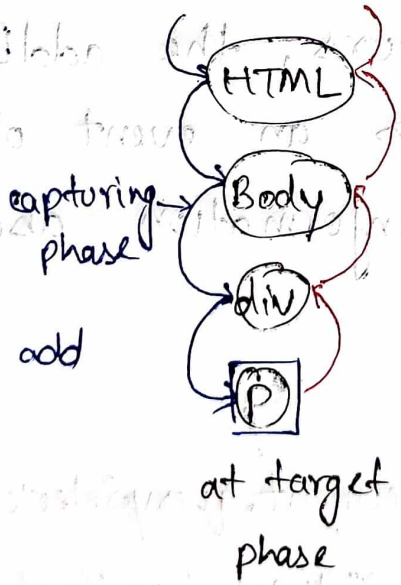
This method of eventtarget sends an Event to the object, invoking the affected EventListners in the appropriate order.

Calling dispatchEvent() is the last step in to firing an event. It should have already been created. and initialized.

=> .dispatchEvent(event)

* Phases of an Event
  * Capturing phase
  * At target phase
  * Bubbling phase

Ex



Here we are finding <p> to add a listner

Let say we want to add listener to <p>

```
<html>
<body>
  <div id="wrap">
    <p> ---- </p>   (Listner)
  </div>
</body>
</html>
```

By default addEventListener() will execute in the bubbling phase.

If I want to execute my event listner to the capturing phase.

Then I will use 3rd parameter.

.addEventListner('click', print, true)

here true value will turn on the capturing phase

NOTE – No we can't apply an event listner on Target phase.

* The Event Object

→ Browser sends the events

When an event occurs, the addEventListner( ) function gets an event object with a lot of information about that ~~object~~ event.

Ex

```
const content = document.querySelector('#wrappe');
content.addEventListner('click', function(event) {
    console.log(event);
});
```

Note: Here 'event' is just a name, you can write anything.

* The default action

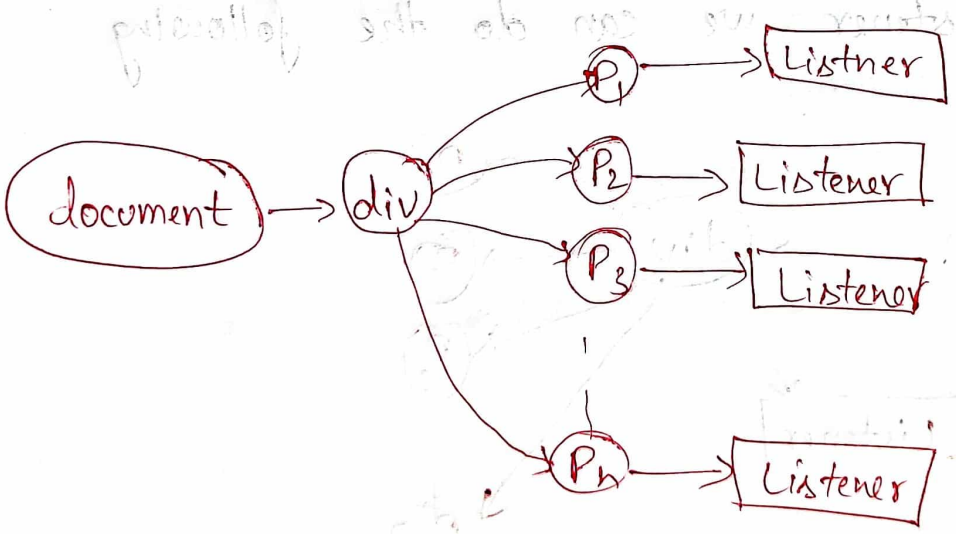
Let say if there is an <a> and its default action is to open a link.

Here if we want we can prevent the default of the <a>, not to open the link.

⟹ .preventDefault()

With the use of this method of the Event interface we can prevent any default action of the tag.

* How to avoid too many Events?

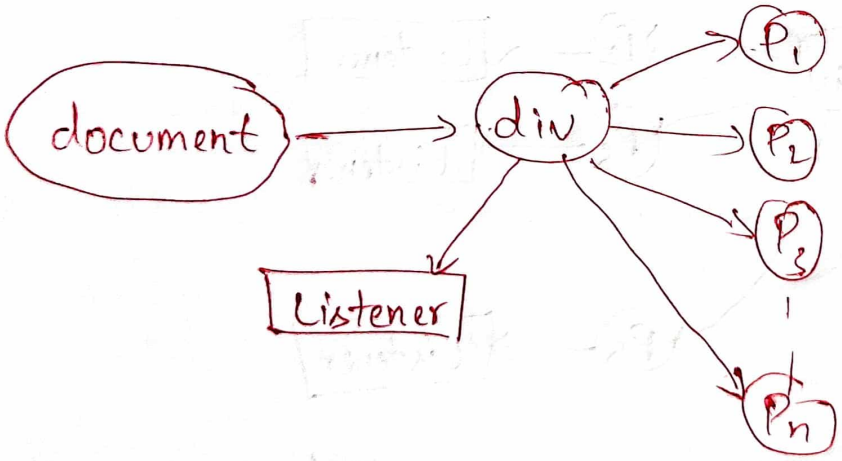


In the above diagram same listner is attached to many paragraph with different objects, so the memory usage is high.

Code

```
let myDiv = document.createElement('div');
for (let i=1 ; i<=100 ; i++) {
    let newElement = document.createElement('p');
    newElement.textContent = 'This is para '+i;
    newElement.addEventListner('click', function(event) {
        console.log('I clicked para');
    });
    myDiv.appendChild(newElement);
}
document.body.appendChild(myDiv);
```

* To avoid creating multiple objects for a same listener we can do the following



Instead of creating multiple mappings we created only 1 mapping for div.

But here is a issue, we cannot access each paragraph individually.

Now if we click anywhere in the , the listener will execute.

We lost the individuality of the paragraph.

⟹ Now we will make use of phase to avoid the issue of lost of individuality.

Code

```
let myDiv = document.createElement('div');

function paraStatus(event) {
        console.log('I clicked para');
}
myDiv.addEventListener('click', paraStatus);
for(let i=0; i<=100; i++) {
        let newElement = document.createElement('p');
        newElement.textContent = 'This is para ' + i;

        myDiv.appendChild(newElement);
}
document.body.appendChild(myDiv);
```

* The target Property

To gain the individuality of the paragraph we will use the target property

Definition

The target property returns the element where the event occured.

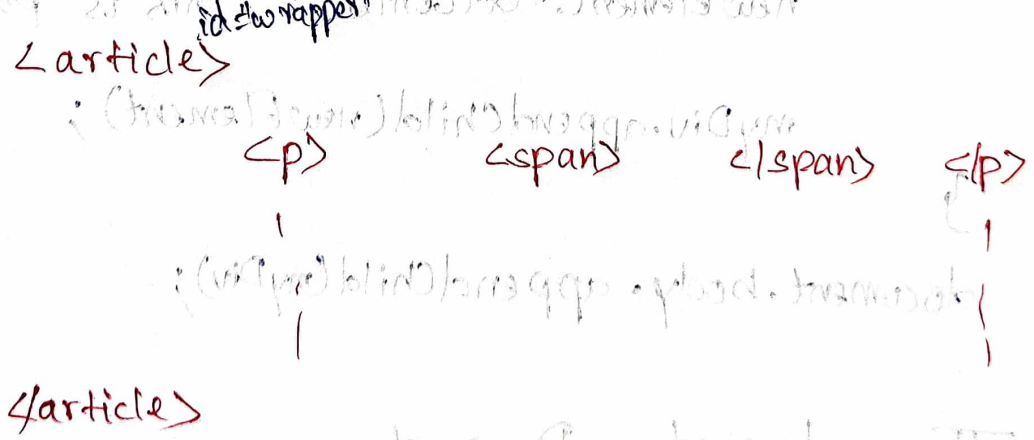
Code

Here the entire code will be same, the only change occur in function.

```
function paraStatus(event) {
    console.log("Para ", event.target.textContent);
}
```

Now we can access individual para by using eventListener on a div.

But there is a catch in it if we modify it little bit.



The issue in above structure is that we want code to run when we click only on span but when we click on para the code is still running.

We don't want listener to execute when we click on <p>.

We can avoid the above problem by using the property .nodeName

Code (code with problem)

```
let element = document.querySelector("#wrapper");

element.addEventListner('click', function(event) {
        console.log('I clicked span' + event.target.textContent);
});
// Here the code will run in both cases i.e. <p> & <span>
```

Code (without problem)

```
let element = document.querySelector('#wrapper');

element.addEventListener('click', function(event) {
    if (event.target.nodeName === 'SPAN') {
        console.log('I clicked span' + event.target.textContent);
    }
}); // Here the code will run only when we click on <span> tag.
```

NOTE: Always use <script> tags in the last of <body> tag as it is a best practice.

* JavaScript Performance

⇒ performance.now()

This method returns a high resolution timestamp in millisecond. It represents the time elapsed

* How to speed up your JS code?

→ Reduce activity in loops
→ Reduce DOM access
→ Reduce DOM size
→ Avoid unnecessary variables
→ Delay JS loading

* Reflow

It means re-calculating the position and geometries of elements in the document, for the purpose of re-rendering part or all of the document.

* Repaint

It is nothing but the repainting elements on the screen as the skin of element

change which affects the visibility of an element but do not affects layout.

Example.

i) changing visibility of an element

ii) changing background.

Repaint is a relatively faster process than the reflow.

The less the number of reflow and repaint the faster JS will be.

Let say if we are working with 100 paragraphs the there will be reflow and repaint 100 time which makes JS slower.

To avoid this we use ~~ble~~ below concept.

* DocumentFragment

It is a lightweight version of Document that stores a segment of a document structure comprised of nodes just like a standard document.

If we add 100 paras here then there will be no reflow and repaint.

The only reflow and repaint will happen when this fragment will be added to the document.

Fragment creation .createDocumentFragment();

→ JavaScript is a single threaded language.

→ It follows syncronous approach.

* Call Stack

A call stack is a mechanism for an interpreter to keep track of its place in a script that calls multiple functions — what a function currently being run and what functions are called from within that function, etc.

* Event Loop

An event loop is a mechanism in JS that allows it to handle multiple events and execute code asynchronously. It's a core part of the JS runtime environment and is responsible for managing the exception of code, handling events and maintaining the event queue.

For more details refer to "Philip Robert" video.

* ~~Set~~ setTimeout()

It allows you to schedule a function to be executed after a specified amount of time

Syntax ⇒ setTimeout(function, delay);

* Features of Async Code

→ Clean & consize

→ Better error handeling

→ Easier debugging

→ Improved performance

* Promise

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

A Promise is in one of these states:

→ pending: initial state, neither fulfilled nor rejected.

→ fulfilled: when operation is completed successfully.

→ rejected: when operation is failed.

Promises provide two main methods to handle their states: `then()` and `catch()`. The `then()` method is used to handle a fulfilled promise and receives the resulting value as its arguement. The `catch()` method is used to handle a rejected promise and receives the reason for the rejection as its arguement.

The constructor syntax for a promise object is:

```
let promise1 = new Promise(function(resolve, reject)
{
    // executor
});
```

The function passed to new Promise is called executor. When it is created, the executor runs automatically.

* Async

The keyword 'async' before a function makes the function return a promise

Ex

```
async function Hello(){
    return "hyz";
}
```

Some scenarios when 'async' is used:

→ Network request
→ Timers
→ File I/O

* Await

'await' is a keyword in JS that is used to wait for a Promise to resolve or reject before executing the next line of code.
It can only be inside 'async' function, which are functions that are marked as asynchronous using the 'async' keyword.

Ex

```
let value = await promise1;
```

used inside async function

* API

Application Programming Interface

An API is a way for different software applications to talk to each other. It defines the rules for how one application can interact with another application, allowing them to share data and functionality.

* Fetch API

The Fetch API interface allows web browser to make HTTP requests to web servers.

GET - To receive data,

POST - To send data, etc

[explore more]

* JSON

JavaScript Object Notation

→ It is a text format for storing and transporting data.

→ Here data is in key value pair.

→ Data is separated by commas

→ Curly braces holds objects

→ Square brackets hold arrays

Ex- {"name": "Rishabh", "car": null}

* Closure

A closure is a feature of the JS that allows a function to access variable (through reference not by copy) and function from its out lexical scope, even after the outer function has returned.

In other words, a closure is a function that has access to variables in its own scope, as well as variables in the scope of the function that created it.