[11] DOM

The Document Object Model (DOM) is a programming interface for web documents. It represents the structure of an HTML or XML document as a tree-like model where each node corresponds to a part of the document, such as elements, attributes, and text. JavaScript interacts with the DOM to manipulate web pages dynamically, allowing you to create, modify, and delete elements and their content.

DOM Tree Structure

- Document: The root node of the DOM tree, representing the entire HTML document.
- Elements: HTML tags in the document, such as , <p>, and <a>, which can have child elements and attributes.
- Attributes: Properties of HTML elements, such as id, class, and href.
- Text Nodes: Contain text within elements, such as the text within <p> or <a> tags.
- Comments: Represent HTML comments in the document.
- Document Fragments: Lightweight containers for groups of DOM nodes.

Attributes vs Properties

In JavaScript, there is a distinction between attributes and properties when working with DOM elements. Understanding this difference is important because they are used in different contexts and have different behaviors.

Attributes:

1. HTML Source: Attributes are defined in the HTML source code and represent the initial values of elements.
2. String Values: Attributes always return string values, even for non-string attribute types like numbers or booleans.
3. Get and Set: You can use the getAttribute() method to retrieve the value of an attribute, and the setAttribute() method to set or change the value of an attribute.
4. Case Sensitivity: Attribute names are case-insensitive in HTML, but they are case-sensitive in XML.

5. Reflects Changes: Attributes generally reflect the state of the HTML source code and don't always update when an element's property changes.

Properties:

1. Current State: Properties represent the current state or values of DOM elements as they exist in the browser's memory.
2. Type Conversion: Properties return values in their native JavaScript data types. For example, value property returns the value of an input element as a string, number, or other type depending on the input type.
3. Get and Set: You can directly access and modify properties of DOM elements using dot notation.
4. Case Sensitivity: Property names are case-sensitive in JavaScript.
5. Reflects Changes: Properties are typically updated when you change an element's state programmatically. For example, setting the value property of an input element changes its value, and this change is reflected in the property.

Attributes represent the initial HTML source code, while properties reflect the current state of the elements in memory. When interacting with or manipulating elements dynamically, it's often more appropriate to use properties.

```
---HTML
<input id="myInput" type="text" value="Hello">

---JS
const inputElement = document.getElementById("myInput");
// Working with attributes
console.log(inputElement.getAttribute("value")); // "Hello"
inputElement.setAttribute("value", "World");
console.log(inputElement.getAttribute("value")); // "World"
// Working with properties
console.log(inputElement.value); // "World"
inputElement.value = "JavaScript";
console.log(inputElement.value); // "JavaScript"
```

Traversing the DOM

Traversing child nodes:

- childNodes retorna todos los nodos, incluyendo texto y espacios; Child retorna solo lo que tiene un HTML tag.

```
ul.children <-- Toma todos.
ul.children[1] <-- Toma el segundo li, de un ul

document.querySelector('li: last-of-type'); <-- Una forma de tener acceso, pero impacta el performance.
```

Using parentNode & ParentElement

- Solo se tiene UN parent element o node, aunque puedas tener muchos children. Por lo tanto podemos usar o parentElement o parentNode. La excepción es para trabajar con el documento, ahi se usa node, pero eso no tiene sentido, porque todo lo del documento se accede con document.

```
const liFirst = document.querySelector('li');
liFirst.parentNode <--Tenemos acceso al ul que contiene estos li.
liFirst.closest('div') <-- El antecesor div mas cercano, hacia arriba.
```

Selecting Sibling Elements

- Se usa para los que están al mismo nivel.

```
ul.previousSibling <---El node mas cercano, incluyendo textos.
ul.previousElementSibling <---El html anterior mas cercano, ejemplo. el hader.
ul.previousElementSibling <-- El html posterior mas cercano, ejemplo. el hader.
```

Accessing DOM Elements

The document object represents the entire HTML document and serves as the entry point for accessing DOM elements. You can access elements in the DOM using various methods, such as getElementById, querySelector, querySelectorAll, getElementsByTagName, and getElementsByClassName.

Element By ID

```
document.getElementById('main-title') <--- Element ID.
console.dir(document.getElementById('main-title'))

const MyH1 = document.getElementById('main-title')
MyH1.querySelectorAll() <--- Y puedo acceder a metodos en el.
```

Element By Class Name

Still supported. Nos da un HTML collection, que es dom element like-array que podemos iterar

```
document.getElementsByClassName('list-item') <-- Accedemos a todos los que tengan estos.
```

QuerySelector

Si queremos acceder por clase o tag, usamos este. Si queremos un collection. Si usamos clases debemos poner .clase y si es ID #ID

```
document.querySelector('.list-item') <-- Accedemos al primero que tenga esto.
document.querySelectorAll('.list-item') <-- Accedemos a todos los que tengan estos.
```

QuerySelectorAll

Retorna **todos los elementos** que cumplan con el criterio dado. Al igual que querySelector puede ser invocada sobre el document o sobre algún elemento. Siempre retorna un NodeList, el cual no es una array con todas las de la ley (map, reduce, indexOf, etc), pero pude convertirse en uno, con facilidad. Si no encuentra elementos que cumplan el criterio, retorna un NodeList sin elementos, nunca retorna null.

```
Array.prototype.slice.call(document.querySelectorAll("div"))
```

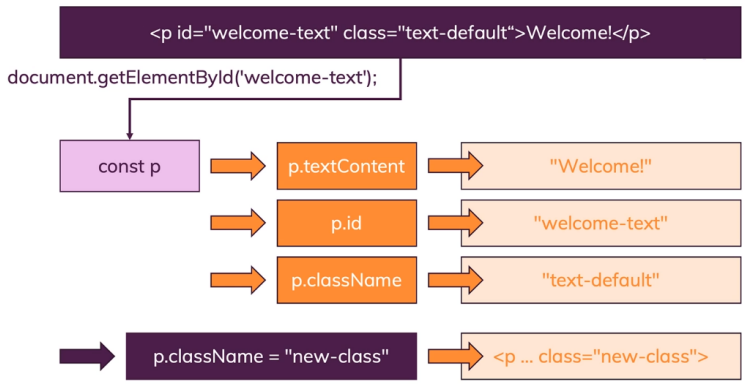
getElementsByTagName

```
const listItemElements = document.querySelectorAll('li') <--- Accedes a todos los li.
const listItemElements = document.getElementsByTagName('li') <--- Accedes a todos los li.

for (const listItemEl of listItemElements) {
console.dir(listItemEl);
}
```

Modifying DOM Elements

Evaluating & Manipulating Elements



You can change element content, attributes, and styles using JavaScript. Common methods include innerHTML, textContent, setAttribute, classList, and style.

```
const element = document.getElementById("myElement");

element.innerHTML = "New content";
element.textContent = "Text content";
element.setAttribute("class", "newClass");
element.classList.add("anotherClass");
element.style.color = "blue";
element.style.backgroundColor = 'black'
```

Styling DOM Elements

Via style Property	Via className	Via classList
Directly target individual CSS styles (on the element)	Directly set the CSS classes assigned to the element	Conveniently add, remove or toggle CSS classes
Controls styles as inline styles on the element	Set/ Control all classes at once	Fine-grained control over classes that are added
Style property names are based on CSS properties but have adjusted names (e.g. backgroundColor)	You can also control the id or other properties	Can be used with className (with care)

1. Style: Nos permite insertar pseudo CSS, se hace de forma manual. Es como el html style.

```
section.style.backgroundColor = 'green'
```

2. Classname: Nos permite cambiar una clase que tenga determinadas propiedades en el css code.

3. Classlist: Nos permite modificar clases, pero tiene ciertas propiedades que podemos usar para evaluar, remover, añadir clases

```
button.addEventistener('click', () => {
section.classList.toggle('invisible')
}

----Hubiesemos tenido que usar esto, de no haber la existencia de classlist.toggle------

button.addEventistener('click', () => {
if (section, className === 'red bg visible') {
section.className = 'red-bg invisible';
} else {
section.className = 'redlbg visible';
}
```

Creating and Deleting Elements

JavaScript allows you to create new DOM elements and append them to the document or remove existing elements. Methods like createElement, appendChild, removeChild, and remove are used for this purpose.

InnerHTML

Es usado cuando tenemos que cambiar todos los elementos de nuestro HTML. Sin embargo, NO cuando queremos añadir o manipular nuevos elementos.

```
const list = document.querySelector('ul');

list.innerHTML = list.innerHTML + '<li>Item 4</li>' <---Esto anade el elemento 4, si, PERO REMDERIZA TODO DE NUEVO.

div.inserAdjacentHTML('beforeend'. '<p> Someting went wrong! </p>'); <--Esto SOLO anade al final. (position,text)
```

```
El down site es que modificar lo que acabamos de crear, es complicado, porque no nos da un acceso directo a ese new element
```

CreateElement

```
const newElement = document.createElement("div");
newElement.textContent = "Newly created element";
document.body.appendChild(newElement);

const oldElement = document.getElementById("oldElement");
oldElement.parentNode.removeChild(oldElement);
```

Cloning DOM nodes

```
const newLi2 = newLi.cloneNode(true/flase) <-- true copia todos los childs y etc, false no.
list.append(newLi, newLi2)
```

RemoveChild

```
const.list = docuemnt.querySelector('ul');
list.remove(); <--- removeria toda la lista
list.parentElement.removeChild(list)
```