Indice

[CSS 2](#_Toc142164113)

[Selectors 2](#_Toc142164114)

[DOM Manipulation 2](#_Toc142164115)

[Select elements 2](#_Toc142164116)

[Create elements 2](#_Toc142164117)

[Append elements: 2](#_Toc142164118)

[Remove elements 2](#_Toc142164119)

[Style elements 2](#_Toc142164120)

[Working with classes 3](#_Toc142164121)

[Modify text 3](#_Toc142164122)

[Modify attributes and classes 3](#_Toc142164123)

[Events 4](#_Toc142164124)

[Get event information 5](#_Toc142164125)

[Conectar Javascript con selector de CSS 5](#_Toc142164126)

# CSS

## Selectors

1. Class Selector (.) Syntax: .classname
2. ID Selector (#) Syntax: #idname
3. Descendant Selector Syntax: ancestor descendant
4. Child Selector Syntax: parent > child
5. Attribute Selector Syntax: element[attribute=value]
6. Pseudo-Classes and Pseudo-Elements Syntax: :pseudo-class and ::pseudo-element

# DOM Manipulation

Lo primero es siempre guardar todo en una variable

let variable = document.querySelector()

Y luego usamos esa variable para lo que necesitemos

## Select elements

1) GetElementByID()

2) GetElementByClassName()

3) getElementByTagName()

4) querySelector() = Selecciona solo el primero. Se puede mandar Id, tag, class, lo que sea

5) **querySelectorAll() = Returns a nodelist**

## Create elements

Solo se lo crea en memoria. Una vez creado hay que hacer el append

const div = document.createElement('div');

## Append elements:

Para que sea haga visible:

parentNode.appendChild(childNode) appends childNode as the last child of parentNode

parentNode.insertBefore(newNode, referenceNode) inserts newNode into parentNode before referenceNode

container.append(element)

## Remove elements

parentNode.removeChild(child) removes child from parentNode on the DOM and returns a reference to child

## Style elements

div.style.color = 'blue';

// adds the indicated style rule

div.style.cssText = 'color: blue; background: white;';

// adds several style rules

div.setAttribute('style', 'color: blue; background: white;');

// adds several style rules

Si queremos cambiarle el style a todos los elementos de una lista, hay que usar un for loop para ir loopeando por ellos. No se puede usar xxx.styles.fontSize = 15rem

## Working with classes

div.classList.add('new');

// adds class "new" to your new div

div.classList.remove('new');

// removes "new" class from div

div.classList.toggle('active');

// if div doesn't have class "active" then add it, or if

// it does, then remove it

## Modify text

Hay 3 metodos. La diferencia es en lo que se va a mostrar como resultado final

1. **innerText**: El recomendado. Se toma todo el texto que se encuentra en el nodo y se lo muestra en una sola linea sin importar si hay saltos o no
2. textContent: Se muestra todo el texto y se respetan los saltos de linea si los hay. Se lo muestra tal cual está en el HTML
3. innerHTML: No se recomienda usar por seguridad. igual que el textContent pero también muestra los tags

## Where to include script in HTML

Your JavaScript, for the most part, is run whenever the JS file is run, or when the script tag is encountered in the HTML. If you are including your JavaScript at the top of your file, many of these DOM manipulation methods will not work because the JS code is being run before the nodes are created in the DOM. The simplest way to fix this is to include your JavaScript at the bottom of your HTML file so that it gets run after the DOM nodes are parsed and created.

Alternatively, you can link the JavaScript file in the <head> of your HTML document. Use the <script> tag with the src attribute containing the path to the JS file, and include the defer keyword to load the file after the HTML is parsed, as such:

<head>

  <script src="js-file.js" defer></script>

</head>

## Modify attributes and classes

div.setAttribute('id', 'theDiv');

// if id exists, update it to 'theDiv', else create an id

// with value "theDiv"

div.getAttribute('id');

// returns value of specified attribute, in this case

// "theDiv"

div.removeAttribute('id');

// removes specified attribute

1) x.setAttribute(attribute, valor)

2) x.removeAttribute(attribute)

3) x.classList.add(valor) > Le asigna a x la clase valor

4) x.classList.remove(valor) > Le quita a x la clase valor

## Events

There are three primary ways to go about this: You can specify function attributes directly on your HTML elements, you can set properties of form on[eventType] (onclick, onmousedown, etc.) on the DOM nodes in your JavaScript, or you can attach event listeners to the DOM nodes in your JavaScript. Event listeners are definitely the preferred method, but you will regularly see the others in use, so we’re going to cover all three.

#### Method 1

<button onclick="alert('Hello World')">Click Me</button>

This solution is less than ideal because we’re cluttering our HTML with JavaScript. Also, we can only set one “onclick” property per DOM element, so we’re unable to run multiple separate functions in response to a click event using this method.

#### Method 2

<!-- the HTML file -->

<button id="btn">Click Me</button>

// the JavaScript file

const btn = document.querySelector('#btn');

btn.onclick = () => alert("Hello World");

This is a little better. We’ve moved the JS out of the HTML and into a JS file, but we still have the problem that a DOM element can only have 1 “onclick” property.

#### Method 3

<!-- the HTML file -->

<button id="btn">Click Me Too</button>

// the JavaScript file

const btn = document.querySelector('#btn');

btn.addEventListener('click', () => {

  alert("Hello World");

});

Now, we maintain separation of concerns, and we also allow multiple event listeners if the need arises. Method 3 is much more flexible and powerful, though it is a bit more complex to set up.

Note that all 3 of these methods can be used with named functions like so:

<!-- the HTML file -->

<!-- METHOD 1 -->

<button onclick="alertFunction()">CLICK ME BABY</button>

// the JavaScript file

function alertFunction() {

    alert("YAY! YOU DID IT!");

  }

  // METHOD 2

  btn.onclick = alertFunction;

  // METHOD 3

  btn.addEventListener('click', alertFunction);

Using named functions can clean up your code considerably, and is a really good idea if the function is something that you are going to want to do in multiple places.

## Get event information

The e in that function is an object that references the **event** itself. Within that object you have access to many useful properties and methods (functions that live inside an object) such as which mouse button or key was pressed, or information about the event’s **target** - the DOM node that was clicked.

btn.addEventListener('click', function (e) {

    console.log(e);

  });

## Conectar Javascript con selector de CSS

Supongamos que tenemos la siguiente lista:

  <audio data-key="keyA" src="sounds/clap.wav"></audio>

  <audio data-key="keyS" src="sounds/hihat.wav"></audio>

  <audio data-key="keyD" src="sounds/kick.wav"></audio>

  <audio data-key="keyF" src="sounds/openhat.wav"></audio>

  <audio data-key="keyG" src="sounds/boom.wav"></audio>

  <audio data-key="keyH" src="sounds/ride.wav"></audio>

  <audio data-key="keyJ" src="sounds/snare.wav"></audio>

  <audio data-key="keyK" src="sounds/tom.wav"></audio>

  <audio data-key="keyL" src="sounds/tink.wav"></audio>

Podríamos agregarle una clase a los audios para poder manipularlos fácilmente desde el script. Sin embargo lo que más conviene es utilizar los selectores de CSS dentro del script:

const audio = document.querySelector(`audio[data-key="keyA"]`);

Si queremos realizarlo más en genérico deberíamos reemplazar el valor hardcodeado por el de la variable

const audio = document.querySelector(`audio[data-key="${e.code}"]`);

## Bubbling

In the context of JavaScript, "bubbling" refers to the order in which events are triggered and propagated through the DOM (Document Object Model) hierarchy. When an event occurs on a DOM element (like a click event on a button), it can trigger handlers on that specific element and then continue propagating up through its parent elements in the DOM tree. This upward propagation is known as event bubbling.

Supongamos que tenemos el siguiente div:

A rectangle with orange rectangles

Description automatically generated

  <div class="one">

    <div class="two">

      <div class="three">

      </div>

    </div>

  </div>

Si agregamos un event listener para ver las propiedades de los *div*, nos vamos a dar cuenta que al darle click al del medio (Class three) también sale las propiedades de toos los padres que le anteceden. De la misma forma, también del body.

<script>

  const divs = document.querySelectorAll('div');

  const button = document.querySelector('button');

  function logText(e) {

    console.log(this.classList.value);

    // e.stopPropagation(); // stop bubbling!

    // console.log(this);

  }

  divs.forEach(div => div.addEventListener('click', logText

  }));

Si solo queremos las propiedades del div del medio y no de los padres, tenemos 2 formas de resolverlo:

1. Utilizando el e.stopPropagation()

  function logText(e) {

    console.log(this.classList.value);

    // e.stopPropagation(); // stop bubbling!

    // console.log(this);

  }

1. Agregando parámetros al *addEventListener*
2. divs.forEach(div => div.addEventListener('click', logText, {
3. capture: true,
4. once: true
5. }));

El *once* funciona como un equivalente al *removeEventListener*. Al estar en true hace que una vez que se haga click, el event listener desaparezca.