

# Teoría Javascript II

¡Hola! 🖐️ Te damos la bienvenida a Javascript II – Parte 2

Seguiremos abordando conceptos de Javascript, que nos ayudarán a entender el lenguaje en profundidad para luego aplicarlo a nuestros proyectos.

¡Que continúe el viaje! 🚀

---

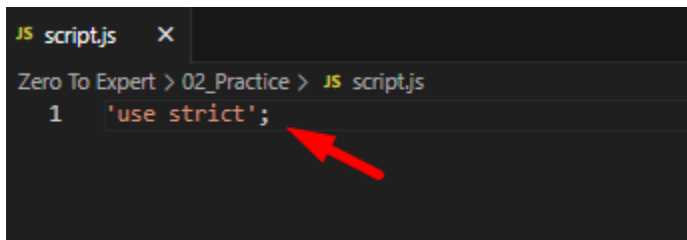
## Desarrollando Habilidades

Vamos a hacer una mini-pausa en el contenido estricto del lenguaje y vamos a ver algunas herramientas que nos van a ayudar a preparar mejor el ambiente de trabajo y a identificar mejor los errores que podemos tener en el código.

### Modo 'use strict'

En JavaScript, **'use strict'** es una directiva que se agrega al principio de un script o una función para indicar que el código debe ser interpretado en "modo estricto" o "strict mode". Este modo introduce una serie de reglas más estrictas y restricciones en la forma en que se escribe el código, lo que puede ayudar a evitar errores comunes y hacer que el código sea más seguro y eficiente.

Se aplica directamente al comienzo del archivo, de esta manera:



```
JS script.js X
Zero To Expert > 02_Practice > JS script.js
1 use strict;
```

💡 A partir de ahora vamos a empezar a utilizar **'use strict'** en todos los archivos

Algunas de las razones para usar 'use strict' son las siguientes:

1. **Prevención de errores silenciosos:** En el modo estricto, ciertas acciones que podrían causar errores silenciosos en el código normal generarán errores en su lugar. Por ejemplo, utilizar variables no declaradas o eliminar variables que no se pueden eliminar causará un error, lo que puede ayudar a detectar y corregir problemas en el código.
2. **Eliminación de comportamientos ambiguos:** El modo estricto desalienta ciertas características de JavaScript que pueden llevar a comportamientos ambiguos o inesperados, lo que puede facilitar la lectura y comprensión del código.
3. **Mejora de la seguridad:** Al evitar ciertos patrones y comportamientos propensos a errores, el modo estricto puede ayudar a reducir la posibilidad de vulnerabilidades de seguridad en el código.
4. **Optimización de rendimiento:** En algunos casos, el modo estricto puede permitir que los motores de JavaScript realicen optimizaciones en el código, lo que puede resultar en un mejor rendimiento.
5. **Preparación para futuras versiones de JavaScript:** El modo estricto también sirve como una forma de preparar el código para futuras versiones de JavaScript, donde ciertas características de JavaScript que se consideran problemáticas pueden eliminarse o cambiar su comportamiento.

## Preparando el ambiente de trabajo

Vamos a crear un snippet para hacer más sencilla la creación del console.log.

Para ello vamos a **File > Preferences > Configure User Snippets > New Global Snippets File** y le colocamos el nombre que querramos.

Luego, tendremos que pegar el siguiente código dentro de las llaves amarillas.

```
"Print to console": {  
  "scope": "javascript,typescript",  
  "prefix": "cl",  
  "body": ["console.log();"],  
  "description": "Log output to console"  
}
```

Esto lo que hará es que cuando escribamos **"cl"** en nuestro archivo javascript, automáticamente va a salir el **console.log("");**

Esto nos permite ahorrar mucho tiempo, ya que no tendremos que tipear todo cada vez.

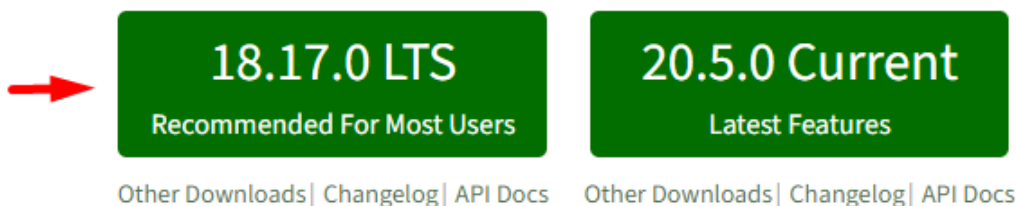
## Instalando Node.Js


Node.js nos permite correr javascript sin necesidad de estar en el navegador.

Para ello deberán ingresar en la página de [Node Js](#) y descargarlo

Node.js® is an open-source, cross-platform JavaScript runtime environment.

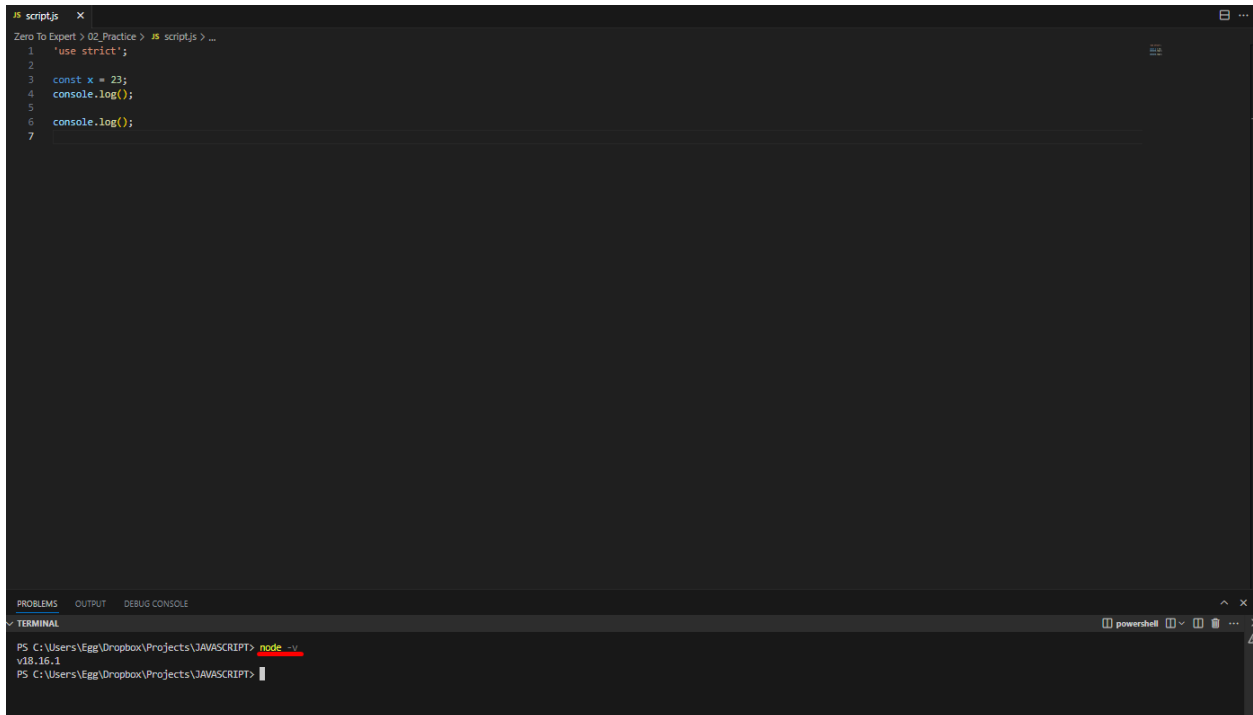
### Download for Windows (x64)



	<b>18.17.0 LTS</b> Recommended For Most Users	<b>20.5.0 Current</b> Latest Features
	<a href="#">Other Downloads</a>   <a href="#">Changelog</a>   <a href="#">API Docs</a>	<a href="#">Other Downloads</a>   <a href="#">Changelog</a>   <a href="#">API Docs</a>

For information about supported releases, see the [release schedule](#).

Para comprobar si se instaló correctamente deberemos abrir **Terminal > New Terminal** y escribiremos **node -v** y si todo está ok, veremos la versión que tenemos instalada.

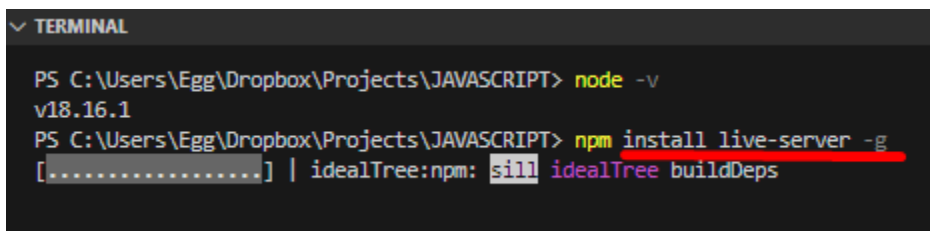


The screenshot shows the Visual Studio Code interface. The editor window displays a file named `script.js` with the following content:

```
1 'use strict';
2
3 const x = 23;
4 console.log(x);
5
6 console.log();
7
```

The terminal window at the bottom shows the command `node -v` being executed, resulting in the output `v18.16.1`.

Ahora, tendremos que instalar el live server. Para ello escribimos en la terminal lo siguiente: **npm install live-server -g**



The screenshot shows the terminal window with the following commands and output:

```
PS C:\Users\Egg\Dropbox\Projects\JAVASCRIPT> node -v
v18.16.1
PS C:\Users\Egg\Dropbox\Projects\JAVASCRIPT> npm install live-server -g
[.....] | idealTree:npm: sill idealTree buildDeps
```

Una vez instalado, podremos tipear `live-server` y se abrirá de manera automática.

💡 **Nota:** Si esto no funciona, es posible seguir utilizando la extensión `live-server` que ya teníamos instalada previamente en visual studio code

# Debugging

El "debugging" (depuración) es el proceso de identificar y corregir errores, fallos o defectos en un programa de software. Es una parte crucial del desarrollo de software, ya que los programas suelen contener errores que pueden afectar su funcionamiento correcto. El objetivo principal es encontrar y eliminar estos errores para que el programa funcione de acuerdo a lo previsto.

Durante el proceso de debugging, los desarrolladores revisan el código, utilizan herramientas y técnicas especiales, y ejecutan el programa en diferentes escenarios para detectar cualquier comportamiento anómalo o resultados inesperados. Una vez que se localiza un error, el desarrollador intenta entender su causa raíz y luego lo corrige mediante la modificación del código.

Las herramientas de debugging son muy útiles en este proceso y pueden proporcionar información valiosa, como el valor de variables en un punto específico del programa, el flujo de ejecución y la pila de llamadas (call stack) en el momento de un fallo.

El proceso de debugging puede ser complejo y requerir habilidades analíticas para identificar y solucionar los problemas de manera eficiente. Además, algunos errores pueden ser más difíciles de detectar que otros, lo que hace que el debugging sea un desafío interesante para los desarrolladores de software.

💡 Se llama **BUG** al comportamiento inesperado o no intencionado dentro de un programa.

---

# El DOM (Document Object Model)

Imagina el DOM (Document Object Model) como un plano detallado de una casa, donde cada elemento de la casa es representado por un objeto que puedes manipular.

En esta analogía, la casa es una página web, y cada parte de la casa, como las habitaciones, muebles, puertas y ventanas, representa diferentes elementos del sitio web, como texto, imágenes, botones y enlaces.

Cuando un navegador carga una página web, crea este plano detallado de la casa (DOM) a partir del código HTML de la página. Es como si el navegador estuviera construyendo una representación interactiva de la casa en su memoria.

Una vez que el DOM está creado, se convierte en la base para interactuar con la página web. Puedes "entrar" en la casa y examinar o cambiar cualquier elemento que desees. Por ejemplo, puedes cambiar el contenido de un texto, mover una imagen o agregar un nuevo botón.

Esta manipulación del DOM se realiza utilizando lenguajes de programación web, como JavaScript. Es como si tuvieras la habilidad de programar la casa para que reaccione a ciertos eventos, como hacer que una puerta se cierre cuando alguien hace clic en un botón.

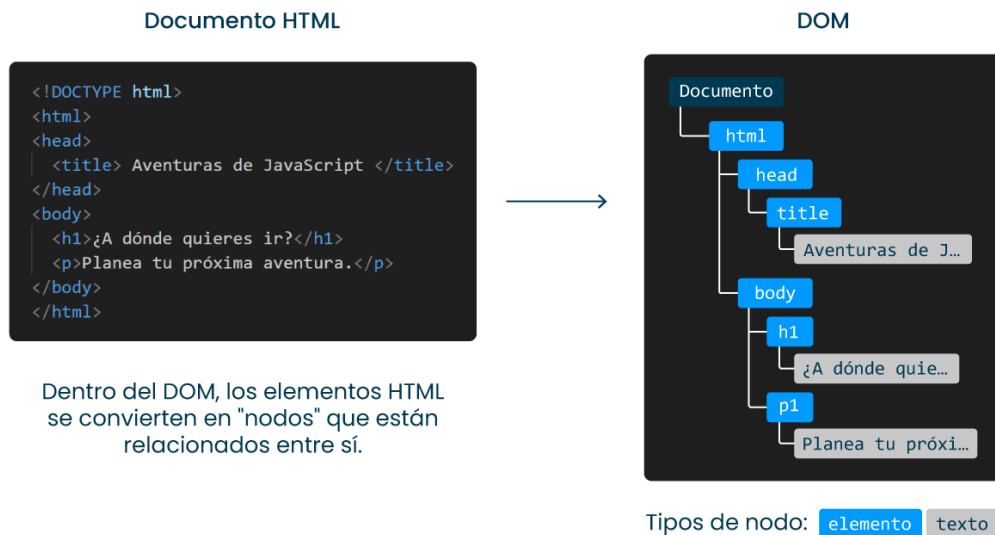
En resumen, el DOM es como un plano interactivo de una casa (página web) que nos permite acceder y modificar sus elementos usando código, proporcionando la base para crear sitios web dinámicos e interactivos.

## ¿Qué es un nodo?

El DOM se construye como un árbol de objetos. Es decir, que **todos los elementos de HTML son definidos como objetos**.

Un nodo es cualquier etiqueta que se encuentra en el **html**.

## ¿Cómo se ve la estructura del DOM?



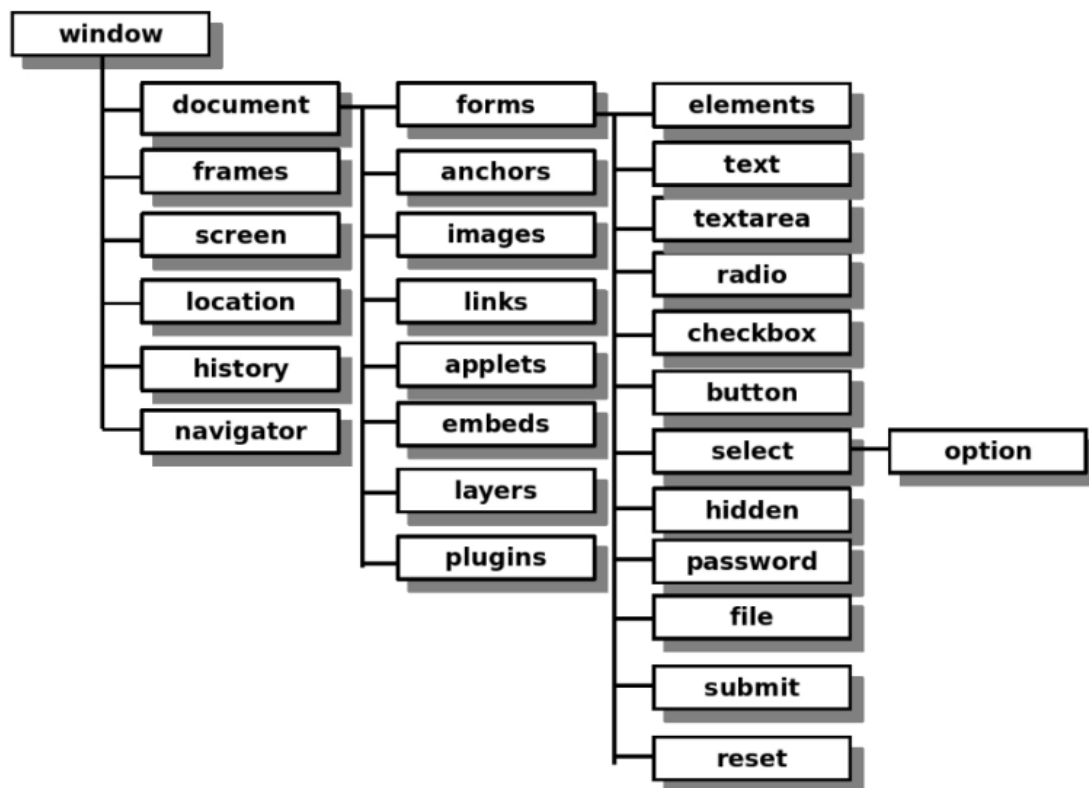
Es una representación similar a un árbol del contenido de una página web, es decir, un árbol de "nodos" con diferentes relaciones dependiendo de cómo estén organizados en el documento HTML.

```
<div id="container">
  <div class="display"></div>
  <div class="controls"></div>
</div>
```

En el ejemplo anterior, el `<div class="display"></div>` es un **hijo** de `<div id="container"></div>` y un **hermano** de `<div class="controls"></div>`.

Imagínalo como un árbol genealógico. `<div id="container"></div>` es un padre, con sus hijos en el siguiente nivel, cada uno en su propia "rama".

En el DOM existe una jerarquía de objetos que se puede representar de la siguiente manera



## Métodos de selección de elementos

Los **métodos** son acciones que podemos realizar sobre los elementos/objetos. Un ejemplo de estas acciones es **“seleccionar los elementos”**.

💡 **Recordemos** que los métodos se aplican a objetos. Y los métodos que vamos a ver a continuación se aplican al objeto **document**

- **document.getElementById("id")** - Seleccionar un elemento a través del ID
- **document.getElementsByTagName("tag")** - Selecciona todos los elementos que coincidan con el nombre de la etiqueta especificada
- **document.querySelector("selector")** - Devuelve el primer elemento que coincida con el grupo especificado de selectores
- **document.querySelectorAll("selector")** - Devuelve todos los elementos que coincidan con el grupo especificado de selectores.



## ¿Cómo seleccionamos un elemento?

Les dejamos el siguiente video para ver cómo seleccionar elementos: [Métodos de selección de elementos | JavaScript II | Egg](#)

Es decir que para seleccionar un elemento del DOM, podemos hacer a través de los selectores de CSS, por ejemplo, dado el siguiente html:

```
<div id="container">
  <div class="display"></div>
  <div class="controls"></div>
</div>
```

Los selectores que podemos tener son:

- `div.display`
- `.display`
- `#container > .display`
- `div#container > div.display`

```
const container = document.querySelector('#container');
// Selecciona el div #container

console.dir(container.firstChild);
// Selecciona el primer hijo del elemento #container => que es .display

const controls = document.querySelector('.controls');
// Selecciona el elemento div que tiene la clase controls

console.dir(controls.previousElementSibling);
// Selecciona el hermano anterior => en este caso, .display
```

Entonces podríamos también seleccionar un nodo, identificando su relación de parentesco con los nodos de alrededor.

## **Métodos para definir, obtener, eliminar valores de atributos**

- **`setAttribute()`** - Modifica el valor de un atributo

- **getAttribute()** - Obtiene el valor de un atributo
- **removeAttribute()** - Remueve el valor de un atributo

## setAttribute()

La sintaxis general del método `setAttribute()` es la siguiente:

```
elemento.setAttribute(nombreAtributo, valorAtributo);
```

Donde:

- **elemento**: Es el elemento HTML al que se le quiere agregar o modificar un atributo.
- **nombreAtributo**: Es una cadena que representa el nombre del atributo que se desea cambiar o agregar.
- **valorAtributo**: Es el valor que se quiere asignar al atributo especificado. Puede ser una cadena de texto o cualquier otro tipo de dato admitido por el atributo en cuestión.

Ejemplo de cómo se utiliza `setAttribute()` para agregar o modificar atributos:

```
<!DOCTYPE html>
<html>
<head>
  <title>Ejemplo de setAttribute()</title>
</head>
<body>
  <div id="miDiv">Este es un div</div>

  <script>
    // Obtenemos el elemento div mediante su ID
    var miDiv = document.getElementById('miDiv');

    // Agregamos el atributo "class" con el valor "destacado"
    miDiv.setAttribute('class', 'destacado');
```

```
// Modificamos el atributo "id" para cambiar su valor
miDiv.setAttribute('id', 'nuevoId');
</script>
</body>
</html>
```

En este ejemplo, hemos utilizado **setAttribute()** para agregar una clase (**class="destacado"**) y modificar el ID (**id="nuevoId"**) del elemento div con el ID "miDiv". Después de la ejecución del script, el elemento div se verá así:

```
<div id="nuevoId" class="destacado">Este es un div</div>
```

En resumen, **setAttribute()** es una función esencial para modificar dinámicamente el contenido y comportamiento de elementos HTML en una página web utilizando JavaScript.

## getAttribute()

La sintaxis general del método **getAttribute()** es la siguiente:

```
var valorAtributo = elemento.getAttribute(nombreAtributo);
```

Donde:

- **elemento**: Es el elemento HTML del cual se quiere obtener el valor del atributo.
- **nombreAtributo**: Es una cadena que representa el nombre del atributo que se desea obtener.

Ejemplo de cómo se utiliza **getAttribute()**:

```
<!DOCTYPE html>
<html>
<head>
```

```
<title>Ejemplo de getAttribute()</title>
</head>
<body>
  <div id="miDiv" class="destacado">Este es un div</div>

  <script>
    // Obtenemos el elemento div mediante su ID
    var miDiv = document.getElementById('miDiv');

    // Obtenemos el valor del atributo "class"
    var claseDelDiv = miDiv.getAttribute('class');
    console.log(claseDelDiv); // Mostrará "destacado"
  </script>
</body>
</html>
```

En este ejemplo, hemos utilizado **getAttribute()** para obtener el valor del atributo **"class"** del elemento div con el ID **"miDiv"**. La variable **claseDelDiv** contendrá el valor **"destacado"**.

Es importante mencionar que **getAttribute()** devuelve siempre el valor del atributo como una cadena de texto. Si el atributo no está presente o no tiene un valor asignado, el método devolverá **null**.

Además, si se quiere acceder a atributos especiales como **href**, **src**, **data-\***, **etc.**, es posible que los navegadores modernos apliquen ciertas transformaciones automáticas al valor devuelto por **getAttribute()**, por lo que puede haber diferencias en la forma en que se obtienen estos atributos en comparación con el código fuente HTML original.

## removeAttribute()

La sintaxis general del método **removeAttribute()** es la siguiente:

```
elemento.removeAttribute(nombreAtributo);
```

Donde:

- **elemento**: Es el elemento HTML del cual se quiere eliminar el atributo.
- **nombreAtributo**: Es una cadena que representa el nombre del atributo que se desea eliminar.

Ejemplo de cómo se utiliza **removeAttribute()**:

```
<!DOCTYPE html>
<html>
<head>
  <title>Ejemplo de removeAttribute()</title>
</head>
<body>
  <div id="miDiv" class="destacado">Este es un div</div>

  <script>
    // Obtenemos el elemento div mediante su ID
    var miDiv = document.getElementById('miDiv');

    // Eliminamos el atributo "class" del elemento div
    miDiv.removeAttribute('class');
  </script>
</body>
</html>
```

En este ejemplo, hemos utilizado **removeAttribute()** para eliminar el atributo "class" del elemento div con el ID "**miDiv**". Después de la ejecución del script, el elemento div ya no tendrá el atributo "**class**".

Es importante tener en cuenta que al eliminar un atributo con **removeAttribute()**, se revertirá cualquier estilo o comportamiento asociado a ese atributo, ya que el elemento volverá a su estado predeterminado. Si el atributo que se intenta eliminar no existe en el elemento, **removeAttribute()** no provocará ningún error o excepción.

El uso de **removeAttribute()** puede ser beneficioso cuando se necesita realizar cambios en la estructura y presentación de una página web de manera dinámica en función de ciertas acciones o eventos.

💡 Para más detalles sobre atributos se puede revisar la documentación de Mozilla [HTML attribute reference](#)

## Métodos para agregar elementos

### createElement()

A través de **createElement**, podemos crear nuevos elementos HTML que luego pueden ser agregados al DOM de una página web.

En JavaScript, la sintaxis para usar **createElement** es la siguiente:

```
document.createElement(tagName)
```

Donde:

- **document** es el objeto que representa el DOM en una página web.
- **tagName** es una cadena de texto que especifica el nombre del elemento HTML que queremos crear, como "div", "p", "span", "a", etc.

Ejemplo de cómo se usa **createElement** para crear un nuevo elemento <p> (párrafo) y luego agregarlo al DOM:

```
// Creamos un nuevo elemento <p>
const newParagraph = document.createElement("p");

// Asignamos contenido al párrafo
newParagraph.textContent = "Este es un nuevo párrafo creado con
JavaScript.";

// Agregamos el párrafo al final del body del documento
document.body.appendChild(newParagraph);
```

En este ejemplo, hemos creado un nuevo **párrafo** (<p>) utilizando **createElement**, luego le hemos asignado texto utilizando la propiedad **textContent**, y finalmente, lo hemos agregado al final del elemento body del DOM utilizando el método **appendChild**.

## Métodos para alterar propiedades de elementos

### Agregar estilos en línea

```
div.style.color = 'blue';  
// le agrega color de letra azul al div  
  
div.style.cssText = 'color: blue; background: white;';  
// agrega más de un estilo  
  
div.setAttribute('style', 'color: blue; background: white;');  
// le agrega varios atributos de estilo, como color y background
```

### Trabajando con clases

```
div.classList.add('new');  
// agrega la clase new al div  
  
div.classList.remove('new');  
// remueve la clase new al div  
  
div.classList.toggle('active');  
// si el div tiene la clase active, la remueve, si no la tiene, la agrega
```

### Agregando o editando texto

Si el contenido del div tiene texto, podemos utilizar:

```
div.textContent = 'Hello World!'  
// crea un nodo de texto que contiene la frase Hello World!
```

```
// Lo inserta en el div

div.textContent = 'Hello World!'
// modifica el texto del div por la frase Hello World!
```

## Agregando o editando valores

Si el elemento es un input y tiene números, podemos utilizar:

```
input.value = 23
// modifica el valor del input por el valor 23
```

## Agregando contenido html

```
div.innerHTML = '<span>Hello World!</span>';
// renderiza el html dentro del div
```

# DOM – Eventos

Ahora que tenemos dominado el manejo del DOM con JavaScript, el siguiente paso es aprender cómo hacerlo de manera dinámica.

Los eventos son la forma en que logramos esa magia en nuestras páginas. Los eventos son acciones que ocurren en nuestra página web, como clics, o pulsaciones de teclas, y mediante el uso de JavaScript, podemos hacer que nuestra página **escuche** y **reaccione** a estos eventos.

Existen tres formas principales de hacerlo:

- puedes especificar atributos de función directamente en tus elementos HTML,



- puedes establecer propiedades de tipo [eventType] (onclick, onmousedown, etc.) en los nodos del DOM en tu JavaScript,
- o puedes adjuntar escuchadores de eventos (**event listeners**) a los nodos del DOM en tu JavaScript.

Los escuchadores de eventos (**event listeners**) son definitivamente el método preferido, pero es probable que veas los otros dos métodos en uso con regularidad, por lo que vamos a cubrir los tres.

## Especificar atributos de función directamente en tus elementos HTML

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

Esta solución no es la mejor, ya que estamos llenando nuestro HTML con JavaScript.

Además, solo podemos establecer una propiedad "onclick" por elemento del DOM, lo que significa que no podemos ejecutar varias funciones separadas en respuesta a un evento de clic utilizando este método.

## Establecer propiedades de tipo [eventType]

Para este ya vamos a necesitar por un lado el html y por el otro un archivo de javascript

### HTML

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

### JAVASCRIPT

```
const btn = document.querySelector('#btn');  
btn.onclick = () => alert("Hello World");
```

Esto es un poco mejor. Hemos sacado el código JavaScript del HTML y lo hemos colocado en un archivo JS, pero aún tenemos el problema de que un elemento del DOM solo puede tener una propiedad "onclick".

## Generar Event listeners

### HTML

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

### JAVASCRIPT

```
const btn = document.querySelector('#btn');
btn.addEventListener('click', () => {
  alert("Hello World");
});
```

Esta es la opción más utilizada dado que podemos aplicar diferentes

**addEventListener** de ser necesario.

El método **addEventListener** en JavaScript permite escuchar diferentes tipos de eventos que ocurren en un elemento HTML. Aquí tienes algunas de las opciones más comunes que puedes utilizar con addEventListener:

- **'click'**: Se activa cuando el usuario hace clic en el elemento.
- **'mouseover'**: Se activa cuando el mouse se mueve sobre el elemento.
- **'mouseout'**: Se activa cuando el mouse sale del área del elemento.
- **'keydown'**: Se activa cuando se presiona una tecla.
- **'keyup'**: Se activa cuando se suelta una tecla que estaba presionada.
- **'submit'**: Se activa cuando se envía un formulario.
- **'input'**: Se activa cuando se ingresa un valor en un campo de entrada (<input>, <textarea>, etc.).
- **'change'**: Se activa cuando el valor de un campo de entrada cambia (útil para elementos <select>).

Esta función se ejecutará cada vez que ocurra un evento particular en ese elemento.

La sintaxis general para addEventListener es la siguiente:

```
element.addEventListener(event, function, useCapture)
```

- **elemento/element**: Es una referencia al elemento HTML al que deseas agregar el evento. Puede ser obtenido a través de diversos métodos, como `getElementById`, `querySelector`, etc.
- **evento/event**: Es una cadena que representa el tipo de evento al que deseas responder, por ejemplo, "click", "mouseover", "keydown", etc. Existen muchos tipos de eventos diferentes que pueden ocurrir en un elemento HTML.
- **funcion/function**: Es una función que se ejecutará cuando ocurra el evento. Puedes proporcionar una función definida previamente o usar una función anónima directamente en el lugar.
- **usarCaptura/useCapture** (opcional): Es un valor booleano que determina si el evento debe ser capturado durante la fase de captura o durante la fase de bubbling (burbujeo). La fase de captura ocurre antes de la fase de bubbling. Si se establece en `true`, el evento será capturado durante la fase de captura, si se omite o se establece en `false`, se capturará durante la fase de bubbling.

💡 Estos son solo algunos ejemplos de eventos que puedes escuchar con `addEventListener`. Puedes encontrar más información sobre eventos y sus detalles en la documentación de JavaScript o en [W3Schools](https://www.w3schools.com/js/default_events.asp). **Es importante tener en cuenta que no todos los eventos son aplicables a todos los elementos HTML, así que es recomendable revisar qué eventos son compatibles con cada tipo de elemento.**

## function

Estos 3 métodos también puede ser utilizados declarando una función. Veamos:

### HTML

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

### JAVASCRIPT

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

// Caso 1
function alertFunction() {
  alert("YAY! YOU DID IT!");
}

// Caso 2
btn.onclick = alertFunction;

// Caso 3
btn.addEventListener('click', alertFunction);
```

Utilizar funciones con nombre puede limpiar considerablemente tu código, y es una muy buena idea si la función es algo que desearás hacer en múltiples lugares.

## function (e)

Con los tres métodos, podemos acceder a más información sobre el evento al pasar un parámetro a la función que estamos llamando.

```
btn.addEventListener('click', function (e) {
  console.log(e);
});
```

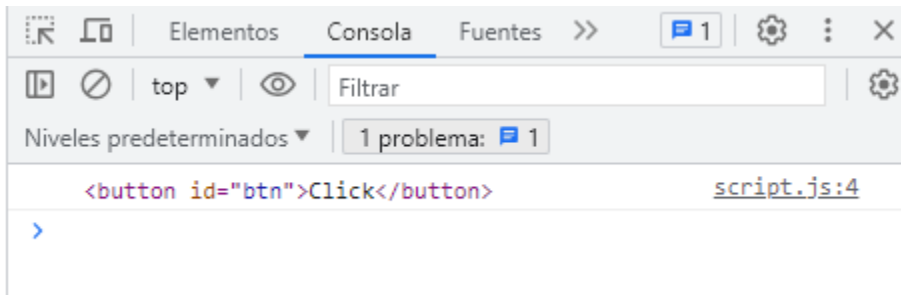
La "e" en esa función es un **objeto** que hace referencia al propio evento. Dentro de ese objeto, tienes acceso a **muchas propiedades y métodos útiles** (funciones que residen dentro de un objeto), como por ejemplo, qué botón o tecla se presionó, o información sobre el objetivo del evento, es decir, el nodo del DOM que fue clicado.

**Prueba esto en tu compu:**

```
btn.addEventListener('click', function (e) {
  console.log(e.target);
});
```

```
});
```

En la consola deberías ver lo siguiente:



Es decir que muestra el elemento HTML al cual se le hizo click.

También puedes probar esto:

```
btn.addEventListener('click', function (e) {  
  e.target.style.background = 'blue';  
});
```

Y verás que el botón cambia a color azul.

**Sorprendente ¿cierto?**

## Generar Event listeners para un grupo de nodos

Más arriba vimos que podemos obtener una lista de nodos utilizando el selector **querySelectorAll(selector)**. Entonces para agregar un Event Listener a cada uno de los elementos de la lista, podemos iterar entre los mismos con ayuda del `forEach`.

Veamos un ejemplo:

### HTML

```
<div id="container">
```

```
<button id="1">Click Me</button>
<button id="2">Click Me</button>
<button id="3">Click Me</button>
</div>
```

## JAVASCRIPT

```
// obtener todos los elementos button
const buttons = document.querySelectorAll('button');

// usamos .forEach para iterar entre cada botón
buttons.forEach((button) => {

  // y por cada uno agregamos un 'click' listener(escuchador)
  button.addEventListener('click', () => {
    alert(button.id);
  });
});
```

---

Este viaje no termina aquí, te esperamos en la siguiente parte teórica 🚀