

JAVA SCRIPT

LEVEL

 **LEVEL 2** 

DISCLAIMER

Todo el código existente en este material es pensando con fines didácticos y no necesariamente propone la forma idónea de implementarse en proyectos reales.

ESTRUCTURA DEL RESUMEN:

EL RESUMEN PRETENDE BRINDAR UN PANTALLAZO GENERAL DE
JAVASCRIPT

¡NO SE ABARCAN TODOS LOS CONCEPTOS EXHAUSTIVAMENTE!

A LO LARGO DEL RESUMEN SE USAN 4 TIPOS DE TÍTULOS:

TITULO 1

TITULO 2

TITULO 3

TITULO 4

LOS CUALES SE SIGUEN DE SUS RESPECTIVAS TIPOGRAFÍAS:

“Siendo esta la letra por defecto para los títulos 1 y 2”

“Y siendo esta la letra por defecto para los títulos 3 y 4”

Cuando se trata de referencias externas o documentación particular

Se usa esta tipografía

CUANDO SE QUIERE HACER UN COMENTARIO INFORMAL CON EL FONDO BLANCO SE
USA ESTA TIPOGRAFÍA

Cuando se quiere hacer un comentario informal con el fondo negro se usa esta tipografía

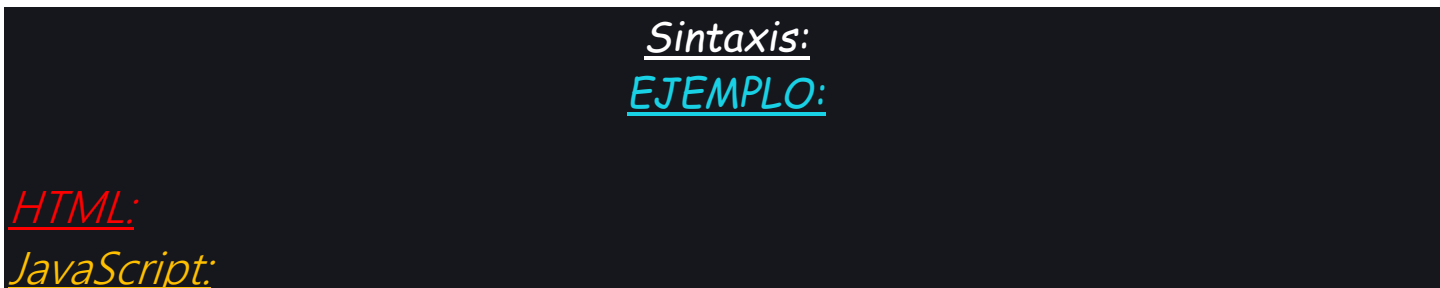
Cuando se quiere hacer un comentario importante o más técnico se usa este color y esta tipografía.

DESPUÉS DE CADA TEMA SUELE HABER UNA MUESTRA TENTATIVA DE LA SINTAXIS QUE DESCRIBE *GROSSO MODO* LA IDEA DE CÓMO SE ORGANIZA/ESTRUCTURA DETERMINADA PALABRA RESERVADA Y SU FUNCIONAMIENTO.

Y DESPUÉS DE LA SINTAXIS SUELE HABER UN EJEMPLO

Y DOS ETIQUETAS: UNA HTML Y UNA JAVASCRIPT CON SUS RESPECTIVOS COLORES:

DE ESTE MODO:



LA ETIQUETA HTML:

MUESTRA LO QUE CONTENDRÍA EL ARCHIVO HTML PERO IGNORA LAS ETIQUETAS FUNDAMENTALES DE TODO DOCUMENTO HTML

A ESCLARECER:

```
<!DOCTYPE html>  
<html></html>  
<head></head>  
<body></body>
```

TENIENDO EN CUENTA QUE EL LECTOR TIENE UNA COMPRENSIÓN DE
HTML Y CSS

LA ETIQUETA JAVASCRIPT:

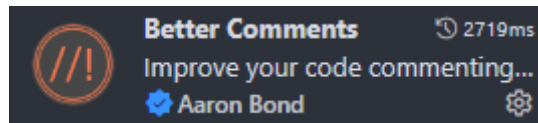
SE REMITE A MOSTRAR CADA CASO DE USO PARTICULAR, PARA QUE EL LECTOR PUEDA VER Y RAZONAR LAS DIFERENTES FORMAS DE TRABAJAR EN LOS DIFERENTES TEMAS CON JAVASCRIPT.

RESPECTO A LOS COMENTARIOS:

LOS CÓDIGOS ESTÁN COMENTADOS CON COLORES PARA QUE SEA MAS PRACTICO Y AMENO PODER LEERLOS, ADEMÁS ESTOS COMENTARIOS TIENEN UNA JERARQUÍA PARTICULAR QUE SIGUEN TODOS.

```
//# TITULO:  
//. Subtitulo 1:  
//_ Subtitulo 2:  
//, Subtitulo 3:  
//$ Subtitulo 4:  
//* Mensaje importante  
//+ Explicacion detallada  
//- Explicacion sintetica  
/// Comentario personal
```

PARA QUE ESTO SEA POSIBLE, SE UTILIZA LA SIGUIENTE EXTENSIÓN...



GRACIAS A ESTO LA INTERACCIÓN ENTRE EL RESUMEN Y EL CÓDIGO ES MÁS INTUITIVA Y MENOS TEDIOSA DE RELACIONAR.

POR DEFECTO LA EXTENSIÓN VIENE CON UNA CONFIGURACIÓN QUE BRINDA OTROS COLORES EN LOS COMENTARIOS.

PARA PODER TENER EXACTAMENTE LOS MISMOs COLORES QUE LOS VISTOS EN LA IMAGEN ANTERIOR ES NECESARIO CAMBIAR LA CONFIGURACIÓN DE AJUSTES PERSONALES DEL EDITOR DE TEXTO.

EL JSON QUE PERMITE LA CONFIGURACIÓN DE COLOR DE ESTOS COMENTARIOS ES EL SIGUIENTE:

```
"better-comments.tags": [  
  {  
    // #comentario  
    "tag": "#",  
    "color": "#E8DFCA",  
    "strikethrough": false,  
    "underline": true,  
    "backgroundColor": "#7895B2",  
    "bold": true,  
    "italic": false  
  },  
  {  
    // .comentario  
    "tag": ".",  
    "color": "#0A1714",  
    "strikethrough": false,  
    "underline": true,  
    "backgroundColor": "#B6F57F",  
    "bold": true,  
    "italic": true  
  },  
  {  
    // _comentario  
    "tag": "_",  
    "color": "#d66127",  
    "strikethrough": false,  
    "underline": true,  
    "backgroundColor": "#dade8c",  
    "bold": true,  
    "italic": true  
  },  
  {  
    // ,comentario  
    "tag": ",",  
    "color": "#0fdb8d",  
    "strikethrough": false,  
    "underline": false,  
    "backgroundColor": "#580cc2",  
    "bold": true,  
    "italic": false  
  },  
  {  
    // $comentario  
    "tag": "$",  
    "color": "#262e2b",  
    "strikethrough": false,  
    "underline": false,  
    "backgroundColor": "#FFACC7",  
    "bold": true,  
    "italic": true  
  },  
  {  
    // *comentario  
    "tag": "*",  
    "color": "#FF8C00",  
    "strikethrough": false,
```

```
"underline": false,
"backgroundColor": "transparent",
"bold": true,
"italic": false
},
{
  //+comentario
  "tag": "+",
  "color": "#E0BB20",
  "strikethrough": false,
  "underline": false,
  "backgroundColor": "transparent",
  "bold": false,
  "italic": false
},
{
  //-comentario
  "tag": "-",
  "color": "#A7FFE4",
  "strikethrough": false,
  "underline": false,
  "backgroundColor": "transparent",
  "bold": false,
  "italic": false
},
{
  ///comentario
  "tag": "/",
  "color": "#c4c4a5",
  "strikethrough": false,
  "underline": false,
  "backgroundColor": "transparent",
  "bold": false,
  "italic": true
}
```

ES POSIBLE QUE A LO LARGO DEL RESUMEN CAMBIE EL COLOR DE LAS PALABRAS RESERVADAS, YA QUE AL LLEVAR TANTO TIEMPO HACERLO SE HAN USADO DIFERENTES PERSONALIZACIONES, NO OBSTANTE, EL CODIGO FUNCIONA IGUAL.

GLOSARIO ESCENCIAL:

SCRIPT:

Se le llama script a cualquier porción de código, literalmente significa “guion”

ELEMENTO:

Cada etiqueta **HTML** es un elemento customizable en **CSS** y dinamizable en **JavaScript**.

ATRIBUTO O PROPIEDAD:

Una **propiedad** o un **atributo** es una característica que un elemento de cierto tipo puede poseer.

VALOR:

Los valores sirven para darle un comportamiento diferente a cada atributo de cada elemento.

FUNCION:

Una función es una porción de código que puede recibir parámetros y devolver un resultado

METODO:

Un método es una función específica de un objeto.

JSON:

Java script object notation, es un tipo de escritura, útil para transportar datos.

API:

Application programming interface, es un grupo de funcionalidades o datos disponibles para importarlas en otro software.

SINCRONIA Y ASINCRONIA:

La **sincronía** hace referencia a código que se ejecuta secuencialmente en un orden.

La **asincronía** hace referencia a código que se ejecuta independientemente y sin un orden fijo.

PROMESA:

Una promesa es una función asíncrona que devuelve el resultado de una evaluación

OBJETO:

En JS todo excepto los datos primitivos representan un objeto, estos tienen una estructura formada por: propiedades con valores y métodos.

CLASE:

Una clase representa una estructura básica que pueden compartir varios objetos

HERENCIA:

Gracias a la herencia los objetos y las clases pueden heredar partes deseadas de la estructura que los conforman

SIMBOLOS:

Estos mismos pueden aparecer después de los títulos para indicar lo que simbolizan.



Simboliza que la propiedad, función o método es un shorthand.

Es decir una propiedad que engloba varias propiedades.



Simboliza que la propiedad, función o método o se usa para obtener un valor u otorgar un valor.

"GET AND SET"

()

Simboliza que el título a continuación se trata de un método por lo tanto en la sintaxis de escritura se debe incluir paréntesis al final

Características de Java Script:

Es interpretado:

JavaScript es un lenguaje interpretado porque utiliza un intérprete que permite convertir las líneas de código en el lenguaje de la máquina. Esto tiene un gran número de ventajas como la reducción del procesamiento en servidores web al ejecutarse directamente en el navegador del usuario, o que es apto para múltiples plataformas permitiendo usar el mismo código.

Orientado a objetos:

JavaScript es un lenguaje orientado a objetos. Que un lenguaje esté orientado a objetos quiere decir que utiliza clases y objetos como estructuras que permiten organizarse de forma simple y son reutilizables durante todo el desarrollo.

Imperativo:

Todas las instrucciones se ejecutan de línea en línea en comparación de otros lenguajes donde se ejecuta todo en conjunto de forma exclusiva.

Case Sensitive:

Es sensible a MAYUSCULAS y minúsculas

Tipado débil:

El valor de las variables puede cambiar durante la ejecución y no es necesario especificar el tipo de dato al crear variables

Dinamico:

La variable no se ajusta al dato el dato se ajusta a la variable.

De alto nivel:

Que *JavaScript* sea un lenguaje de alto nivel significa que su sintaxis es fácilmente comprensible por su similitud al lenguaje de las personas. Se le llama de “alto nivel” porque su sintaxis se encuentra alejada del nivel máquina, es decir, del código que procesa una computadora para ejecutar lo que nosotros programamos.

Usos de JavaScript:

- Dinamismo en Sitios web: Para poder lograr una interacción dinámica del lado del cliente.

Otras posibilidades:

- Inteligencia artificial: Como la librería tensorflow desarrollada por Google.
- Placas electrónicas: Johnny five con Arduino.
- Mobile apps: Es posible crear aplicaciones móviles
- Desktop apps: Es posible crear aplicaciones para escritorio

Aunque no necesariamente JavaScript sea el lenguaje más óptimo para los tipos de desarrollos mencionados previamente, es interesante conocer sus potenciales posibilidades.

Eventos:

Los eventos en **JS** hacen referencia a casos particulares de la interacción del usuario con una aplicación.

Pasar el mouse por arriba, clickear algo, sacar el mouse de un lugar, escribir, estar esperando, abrir/cerrar una pestaña, reproducir/pausar un video, etc... son todos eventos, y nosotros podemos ocupar JS para identificarlos y hacer algo con/en ese preciso evento.

Los eventos constan de dos partes:

Listeners y handlers:

Los listeners “escuchan” el evento, es decir identifican que un determinado evento este o no pasando.

Los handlers llevan a cabo una acción a partir de una escucha; “manejan el enveto”

En resumen: cuando un evento pasa, una función se ejecuta.

Si bien existen formas de usar eventos en el navegador, *no todas son buenas prácticas:*

En los propios elementos **HTML:**

(no recomendado)

Sintaxis:

```
<element EVENT_NAME="function_name()">CONTENIDO</ element >
```

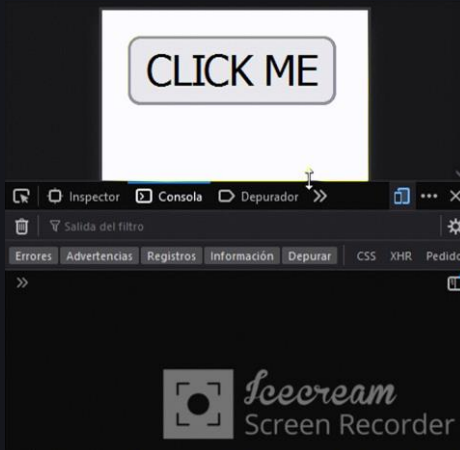
EJEMPLO:

HTML:

```
<button onclick="clickeaste()">CLICK ME</button>
```

JavaScript:

```
function clickeaste(){console.log("clickearon el boton")}
```



Con un listener:

(recomendado)

El cual permite escuchar cualquier evento en el DOM y seria la forma recomendada de trabajar con eventos.

A continuación, una lista de los eventos mayormente utilizados en el navegador, sus explicaciones y sus nombres para la forma recomendada y no recomendada.

SINTAXIS GENERAL:

```
variable_name.addEventListener("event", function{});
```

DOMContentLoaded:

Escucha si el DOM cargo completo.

Este evento es particularmente importante, ya que nos indica cuando están todos los elementos disponibles en nuestro DOM, obstante podemos trabajar con ellos, de lo contrario si operamos con elementos que aun no han sido cargados podríamos tener errores.

JavaScript:

```
document.addEventListener("DOMContentLoaded", ()=>{console.log("Cargaste el DOM!!")})
```

Cargaste el DOM!!

Eventos con el mouse:

Eventos que se pueden disparar al interactuar con el mouse y los elementos **HTML**:

click (onclick):

Escucha cuando se hace click en un elemento.

Sintaxis:

```
variable_name.addEventListener("click", function(){})
```

EJEMPLO:

HTML:

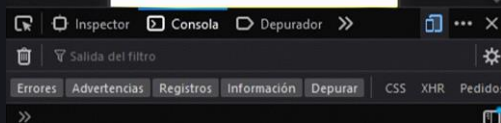
```
<button id="boton">CLICK ME</button>
```

CLICK ME

JavaScript:

```
btn = document.getElementById("boton");  
btn.addEventListener("click", ()=>{console.log("me clickeaste")})
```

CLICK ME



Icecream
Screen Recorder

focus(onfocus):

Escucha cuando un elemento está en foco

solo funciona para los elementos que pueden "enfocarse"

Sintaxis:

```
variable_name.addEventListener("focus", function(){})
```

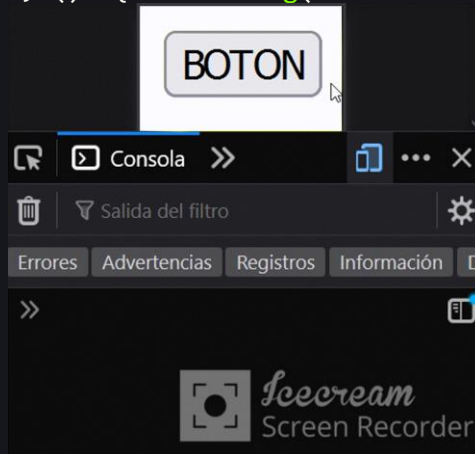
EJEMPLO:

HTML:

```
<button id="boton">BOTON</button>
```

JavaScript:

```
btn = document.getElementById("boton");  
btn.addEventListener("mousemove", ()=>{console.log("Me focuseaste!!")})
```



blur(onblur):

Escucha cuando un elemento esta fuera de foco.

solo funciona para los elementos que pueden "enfocarse"

Sintaxis:

```
variable_name.addEventListener("blur", function(){})
```

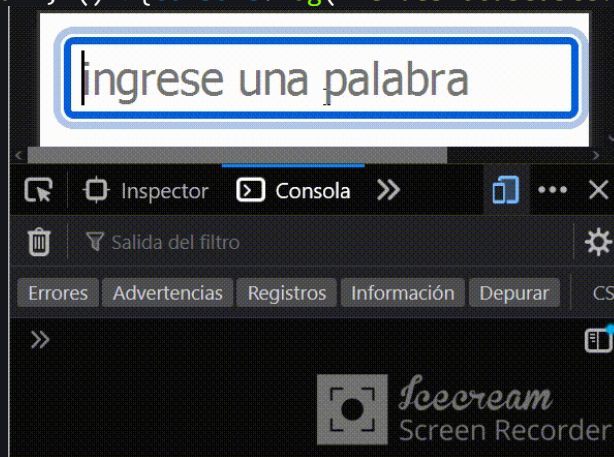
EJEMPLO:

HTML:

```
<input id="campo" type="text" placeholder="ingrese una palabra">
```

JavaScript:

```
campo = document.getElementById("campo");  
campo.addEventListener("blur", ()=>{console.log("Me desfocuseaste!!")})
```



change(onchange):

Escucha dos cosas: cuando un elemento es cambiado y pierde su foco.

Sintaxis:

```
variable_name.addEventListener("change", function(){})
```

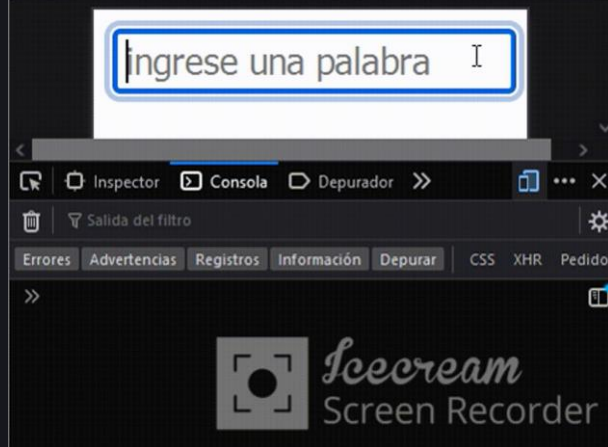
EJEMPLO:

HTML:

```
<input id="campo" type="text" placeholder="ingrese una palabra">
```

JavaScript:

```
const campo = document.getElementById("campo");  
campo.addEventListener("change", ()=>{console.log("Hiciste un cambio! y te fuiste!")})
```



mouseover:

Escucha cuando el mouse entra en el contenedor de un elemento.

Sintaxis:

```
variable_name.addEventListener("mouseover", function(){})
```

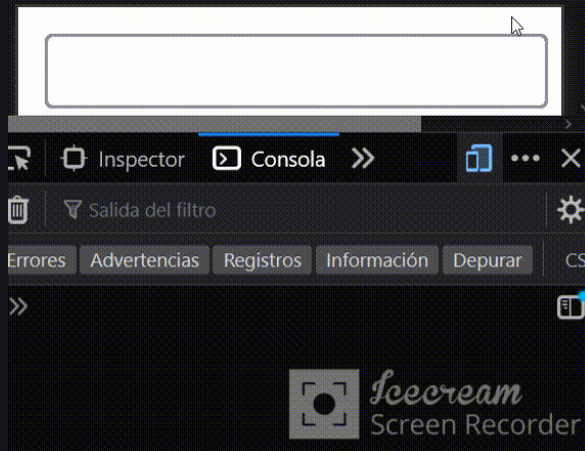
EJEMPLO:

HTML:

```
<input id="campo">
```

JavaScript:

```
btn = document.getElementById("boton");  
btn.addEventListener("mouseover", ()=>{console.log("Entraste!!")})
```



mouseout:

Escucha cuando el mouse sale del contenedor de un elemento

Sintaxis:

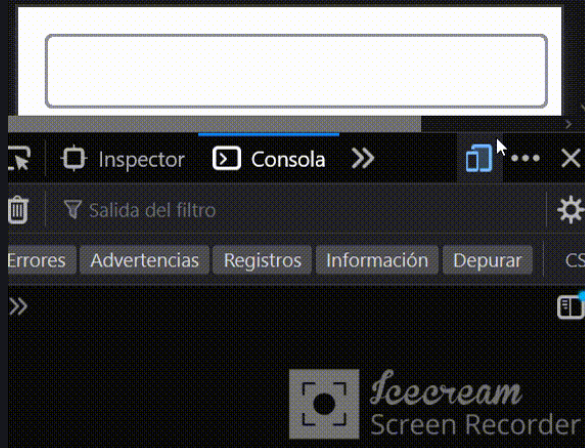
```
variable_name.addEventListener("mouseout", function(){})
```

EJEMPLO:

HTML:

```
<input id="campo">
```

```
btn = document.getElementById("boton");  
btn.addEventListener("mouseout", ()=>{console.log("Te fuiste!!")})
```



mousemove:

Escucha mientras el mouse se mueve por un elemento.

Sintaxis:

```
variable_name.addEventListener("mousemove", function{})
```

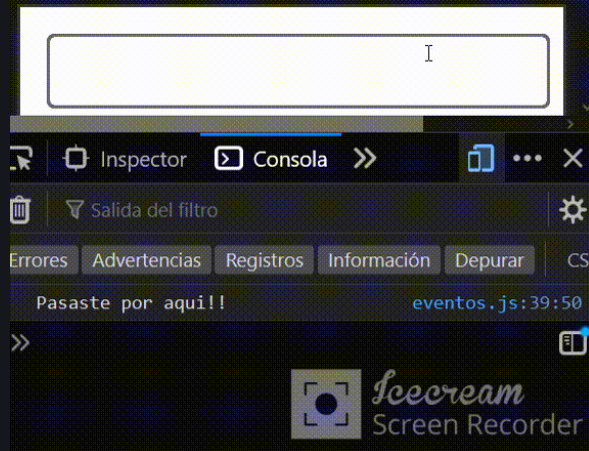
EJEMPLO:

HTML:

```
<input id="campo">
```

JavaScript:

```
btn = document.getElementById("boton");  
btn.addEventListener("mousemove", ()=>{console.log("Pasaste por aqui!!")})
```



mousedown:

Escucha cuando el mouse selecciona un elemento.

Sintaxis:

```
variable_name.addEventListener("mousedown", function{})
```

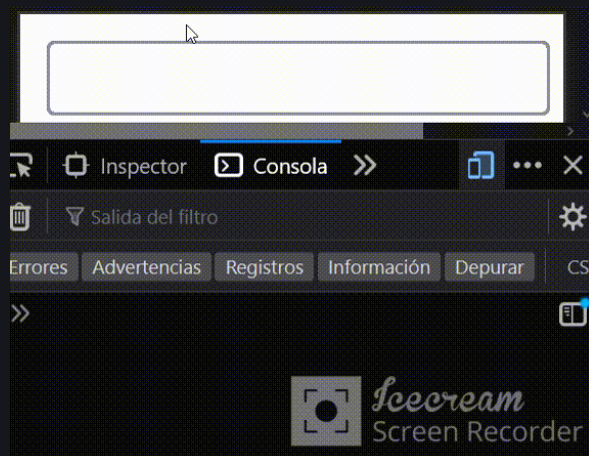
EJEMPLO:

HTML:

```
<input id="campo">
```

JavaScript:

```
btn = document.getElementById("boton");  
btn.addEventListener("mousedown", ()=>{console.log("Me apretaste!!")})
```



mouseup:

Escucha cuando el mouse deselecciona un elemento

Sintaxis:

```
variable_name.addEventListener("mouseup", function{})
```

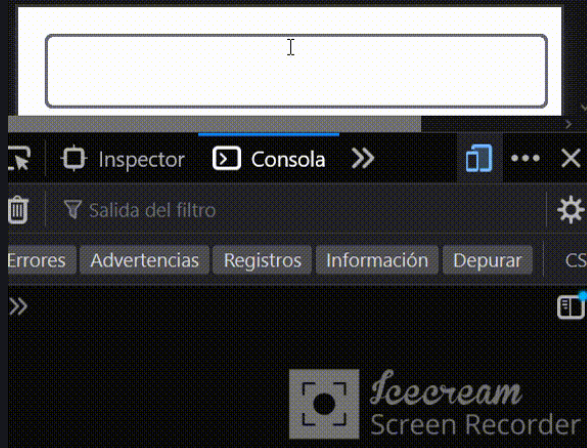
EJEMPLO:

HTML:

```
<input id="campo">
```

JavaScript:

```
btn = document.getElementById("boton");  
btn.addEventListener("mouseup", ()=>{console.log("Me soltaste!!")})
```



reset:

Escucha si un elemento resetea algún formulario.

Sintaxis:

```
variable_name.addEventListener("reset", function(){})
```

EJEMPLO:

HTML:

```
<form id="formulario">  
  <input name="name" type="text" placeholder="Nombre">  
  <input name="email" type="text" placeholder="Email">  
  <input name="pass" type="password" placeholder="Password">  
  <button type="reset">ENVIAR</button>  
</form>
```

JavaScript:

```
form = document.getElementById("formulario");  
form.addEventListener("reset", ()=>{console.log("Has reseteado!!")})
```

A screenshot of a web form. It contains three input fields: the first two are text inputs with the placeholder text "ASD", and the third is a password input with a placeholder of three dots "...". To the right of these inputs is a button labeled "RESETEAR". Below the form, a dark banner displays the message "Has reseteado!!" in a light-colored font.

Select:

Escucha si se ha seleccionado un texto en un campo input.

Sintaxis:

```
variable_name.addEventListener("select", function{})
```

EJEMPLO:

HTML:

```
<input id="campo" type="text" placeholder="ingrese texto">
```

JavaScript:

```
campo = document.getElementById("campo");  
campo.addEventListener("select", ()=>{console.log("Has seleccionado!!")})
```

Has seleccionado!!

submit:

Escucha la acción de submit.

Sintaxis:

```
variable_name.addEventListener("submit", function{})
```

EJEMPLO:

HTML:

```
<form id="formulario">  
<input name="name" type="text" placeholder="Nombre">  
<input name="email" type="text" placeholder="Email">  
<input name="pass" type="password" placeholder="Password">  
<button id="btn" type="submit">ENVIAR</button>  
</form>
```

JavaScript:

```
const form = document.getElementById("formulario");  
form.addEventListener("submit", ()=>{console.log("has enviado el formulario")})
```

has enviado el formulario

Tecclas:

Eventos que se disparan cuando se usa el teclado:

Keydown:

Escucha cuando se pulsa cualquier tecla y mientras sigue pulsada.

Sintaxis:

```
variable_name.addEventListener("keydown", function{})
```

EJEMPLO:

HTML:

```
<input id="campo" type="text" placeholder="ingrese texto">
```

JavaScript:

```
campo = document.getElementById("campo");  
campo.addEventListener("keydown", ()=>{console.log("Pulsaste una tecla")})
```



Keypress:

Escucha cuando una se mantiene pulsada

Sintaxis:

```
variable_name.addEventListener("keypress", function{})
```

EJEMPLO:

HTML:

```
<input id="campo" type="text" placeholder="ingrese texto">
```

JavaScript:

```
campo = document.getElementById("campo");  
campo.addEventListener("keypress", ()=>{console.log("Mantuviste una tecla")})
```



keyup:

Escucha cuando se suelta una tecla presionada previamente.

Sintaxis:

```
variable_name.addEventListener("keyup", function{})
```

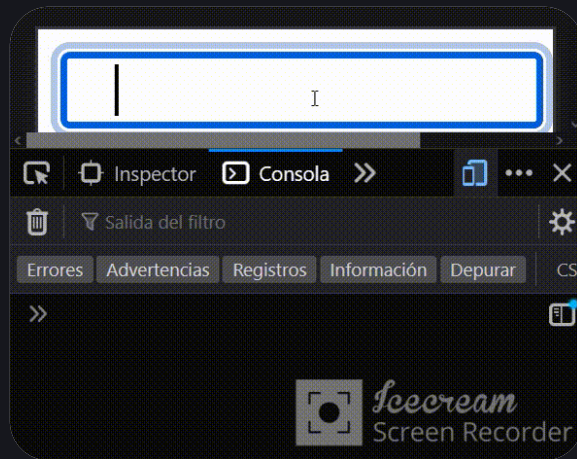
EJEMPLO:

HTML:

```
<input id="campo" type="text" placeholder="ingrese texto">
```

JavaScript:

```
campo = document.getElementById("campo");  
campo.addEventListener("keyup", ()=>{console.log("Soltaste la tecla")})
```



Bubbling:

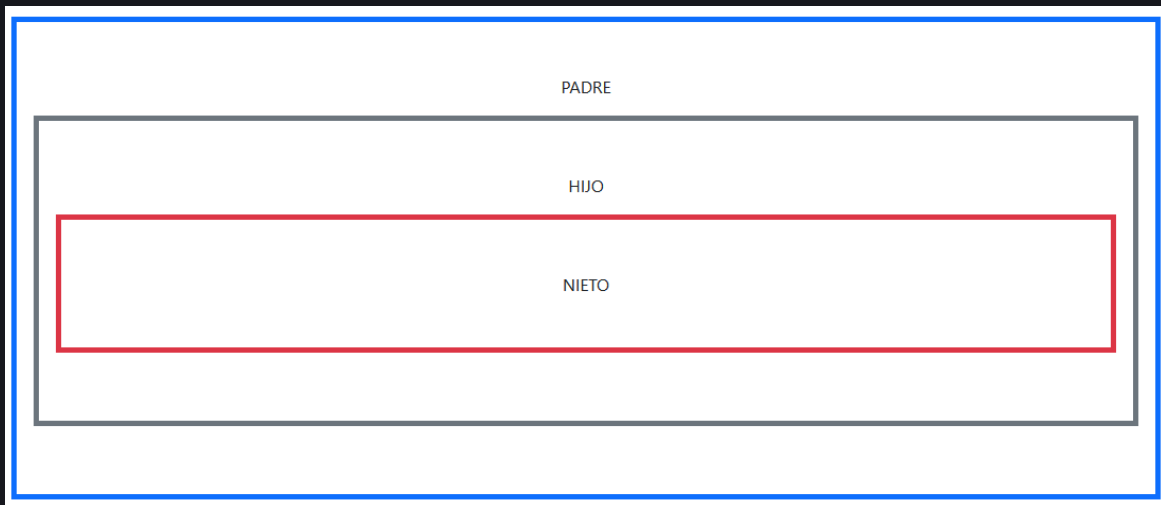
El bubbling o burbujeo, es la escucha de un evento en varios contenedores cuando el evento solo se dispara en uno solo.

puede traer problemas, hay que conocer sobre el mismo, y como tratarlo.

Supongamos:

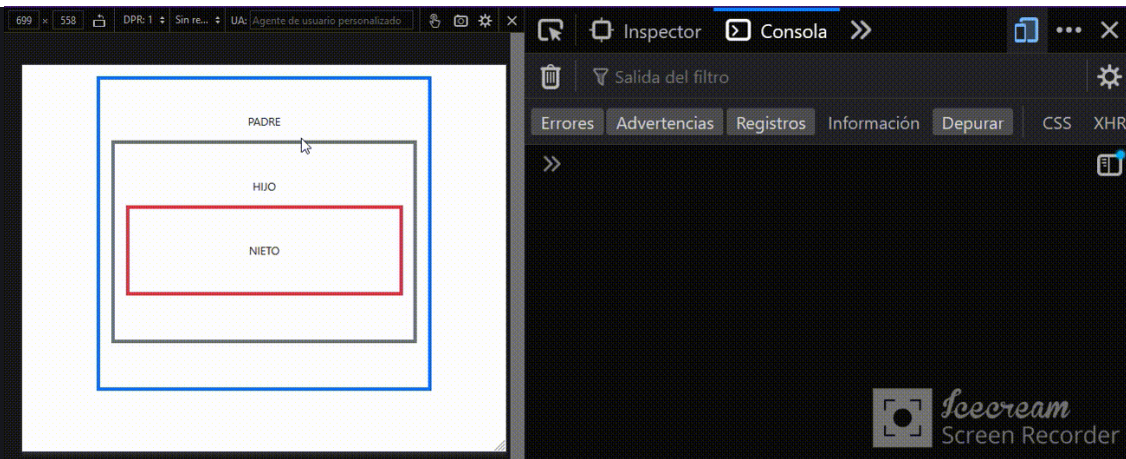
HTML:

```
<div class="container text-center">  
  <main id="padre" class="border border-primary border-5 py-5 m-3 ">  
    PADRE  
    <section id="hijo" class="border border-secondary border-5 py-5 m-3">  
      HIJO  
      <article id="nieto" class="border border-danger border-5 py-5 m-3">  
        NIETO  
      </article>  
    </section>  
  </main>  
</div>
```



JavaScript:

```
padre = document.getElementById("padre");  
hijo = document.getElementById("hijo");  
nieto = document.getElementById("nieto");  
  
padre.addEventListener("click" ,()=>{console.log("CLICKEASTE PADRE");});  
hijo.addEventListener("click" ,()=>{console.log("CLICKEASTE HIJO");});  
nieto.addEventListener("click" ,()=>{console.log("CLICKEASTE NIETO");});
```



Como vemos al clickear el elemento anidado en otros elementos se propaga el evento a todos los contenedores.

Esto se solucionar con el siguiente método.

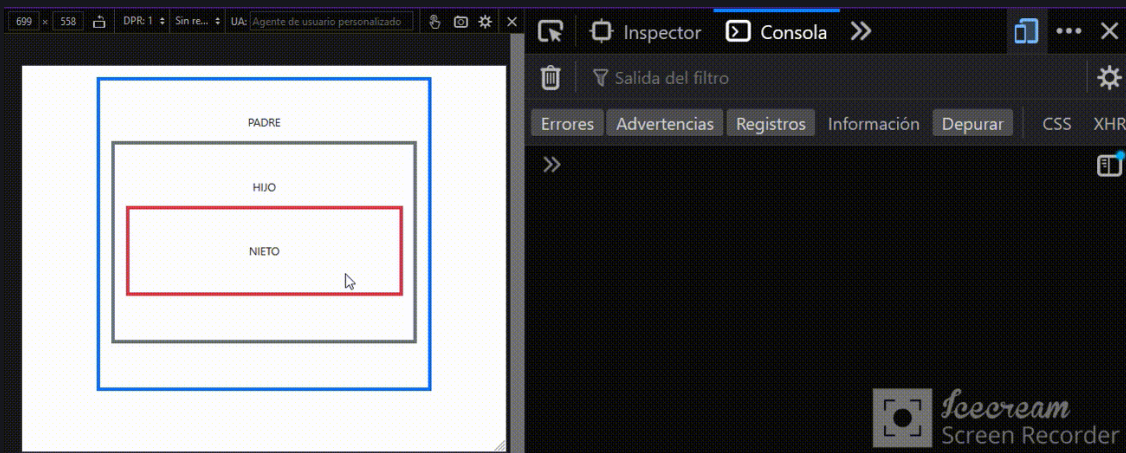
()

stopPropagation:

Evita la propagación de eventos ocasionada por el efecto bubbling.

JavaScript:

```
padre.addEventListener("click" ,(e)=>{
  e.stopPropagation()
  console.log("CLICKEASTE PADRE");
});
hijo.addEventListener("click" ,(e)=>{
  e.stopPropagation()
  console.log("CLICKEASTE HIJO");
});
nieto.addEventListener("click" ,(e)=>{
  e.stopPropagation()
  console.log("CLICKEASTE NIETO");
});
```



Y de este modo el efecto bubbling ya no sucede y podemos trabajar eventualmente con cada elemento particular.

()

preventdefault:

En ciertos eventos puede presentarse un comportamiento nativo no deseado del navegador.

Obstante es preferible usar alguna prevención como....

```
preventDefault();
```

método que previene cualquier comportamiento nativo por default que tenga configurado el navegador, y luego realiza nuestras operaciones.

HTML:

```
<form id="formulario">
  <input name="name" type="text" placeholder="Nombre">
  <input name="email" type="text" placeholder="Email">
  <input name="pass" type="password" placeholder="Password">
  <button id="btn" type="submit">ENVIAR</button>
</form>
```

JavaScript:

```
const form = document.getElementById("formulario");
const enviar = (evn) => {
  evn.preventDefault();
  console.log(evn.target.name, evn.target.email, evn.target.pass)}

form.addEventListener("submit", enviar);
```

al ver lo que muestra el console.log veremos:

<input type="text" value="carlos"/>	<input type="text" value="pepe"/>	<input type="password" value="••••"/>	<input type="button" value="ENVIAR"/>
-------------------------------------	-----------------------------------	---------------------------------------	---------------------------------------

► <input name="name" type="text" placeholder="Nombre"> undefined ► <input name="pass" type="password" placeholder="Password">

Si agregamos la propiedad "value"

```
console.log(evn.target.name.value, evn.target.email.value, evn.target.pass.value)}
```

<input type="text" value="Carlitos"/>	<input type="text" value="pepu@correito.com"/>	<input type="password" value="••••"/>	<input type="button" value="ENVIAR"/>
---------------------------------------	--	---------------------------------------	---------------------------------------

Podremos ver los valores contenidos

Carlitos pepu@correito.com 1234

...Mucho ojo ahí....

Delegación de eventos:

Permite identificar un evento en todo un contenedor para posteriormente dar un tratamiento particular dependiendo de donde específicamente ocurrió el evento.

Sigamos trabajando con la estructura anterior...

HTML:

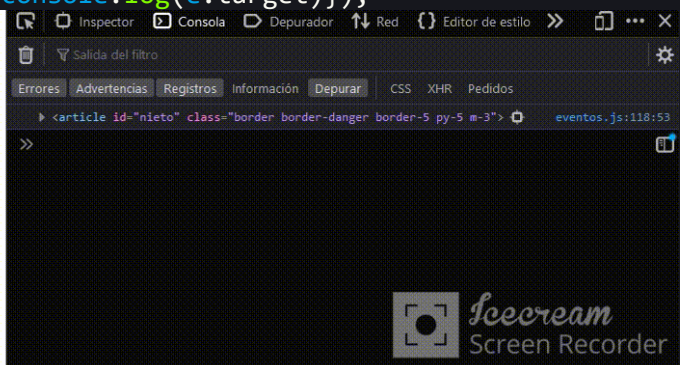
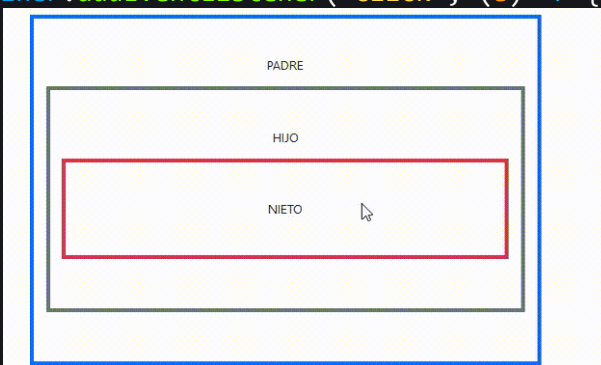
```
<div class="container text-center">
  <main data-info="father" id="padre" class="border border-primary border-5 py-5 m-3 ">
    PADRE
    <section data-info="son" id="hijo" class="border border-secondary border-5 py-5 m-3">
      HIJO
      <article data-info="grandson" id="nieto" class="border border-danger border-5 py-5 m-3">
        NIETO
      </article>
    </section>
  </main>
</div>
```

Target:

Target es una propiedad que nos permite devolver el objeto en el que se disparó el evento.

JavaScript:

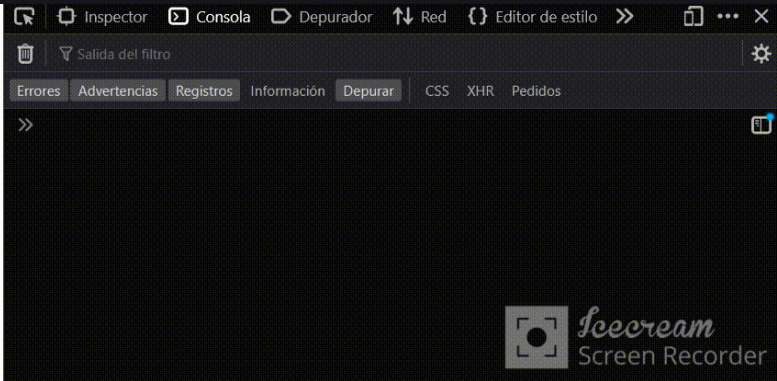
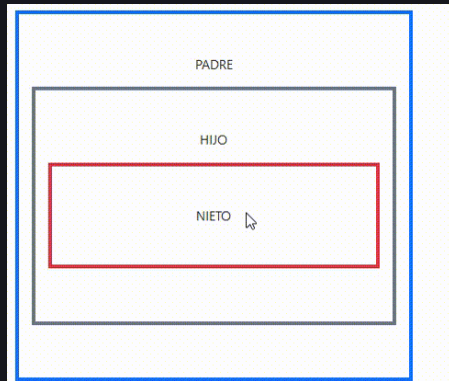
```
const container = document.querySelector(".container");
container.addEventListener("click", (e) => {console.log(e.target)});
```



```
container.addEventListener("click", (e) => {console.log(e.target.id)});
```

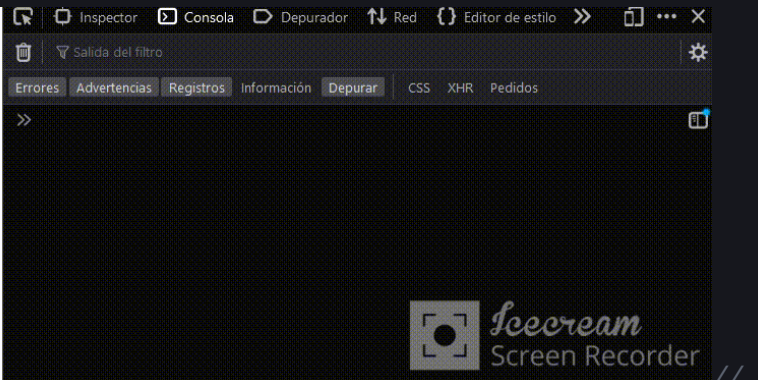



Funciona también si seleccionamos todo el DOM:



USANDO TARGET CON ID:

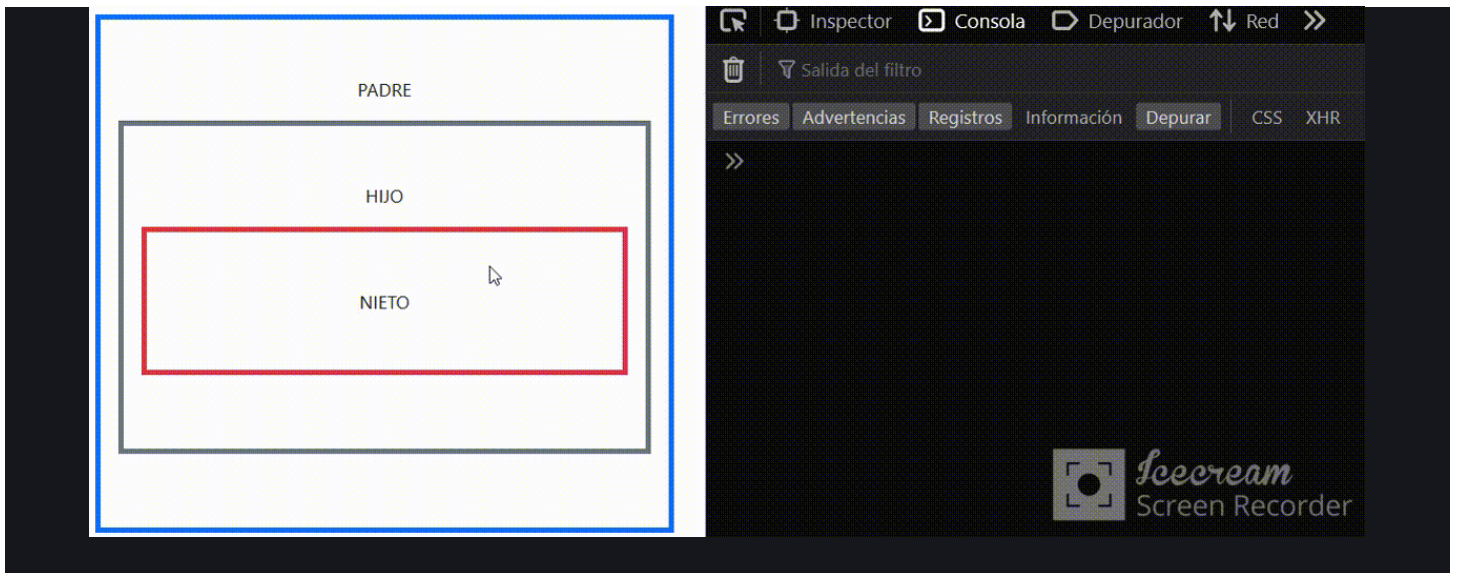
```
container.addEventListener("click", (e) => {
  if (e.target.id === "padre"){console.log("diste click en el padre")}
  if (e.target.id === "hijo"){console.log("diste click en el hijo")}
  if (e.target.id === "nieto"){console.log("diste click en el nieto")}
});
```



USANDO TARGET CON DATASET:

```
// sintaxis 1:
container.addEventListener("click", (e) => {
  if (e.target.dataset["info"] === "father"){console.log("clickeaste dentro del padre")}
  if (e.target.dataset["info"] === "son"){console.log("clickeaste dentro del hijo")}
  if (e.target.dataset["info"] === "grandson"){console.log("clickeaste dentro del nieto")}
});

// sintaxis 2:
container.addEventListener("click", (e) => {
  if (e.target.dataset.info === "father"){console.log("clickeaste dentro del padre")}
  if (e.target.dataset.info === "son"){console.log("clickeaste dentro del hijo")}
  if (e.target.dataset.info === "grandson"){console.log("clickeaste dentro del nieto")}
});
```



matches:

“Matches” es un método que permite seleccionar elementos que puedan ser seleccionables con selectores de CSS.

Si el elemento no es seleccionable, retorna, *false*

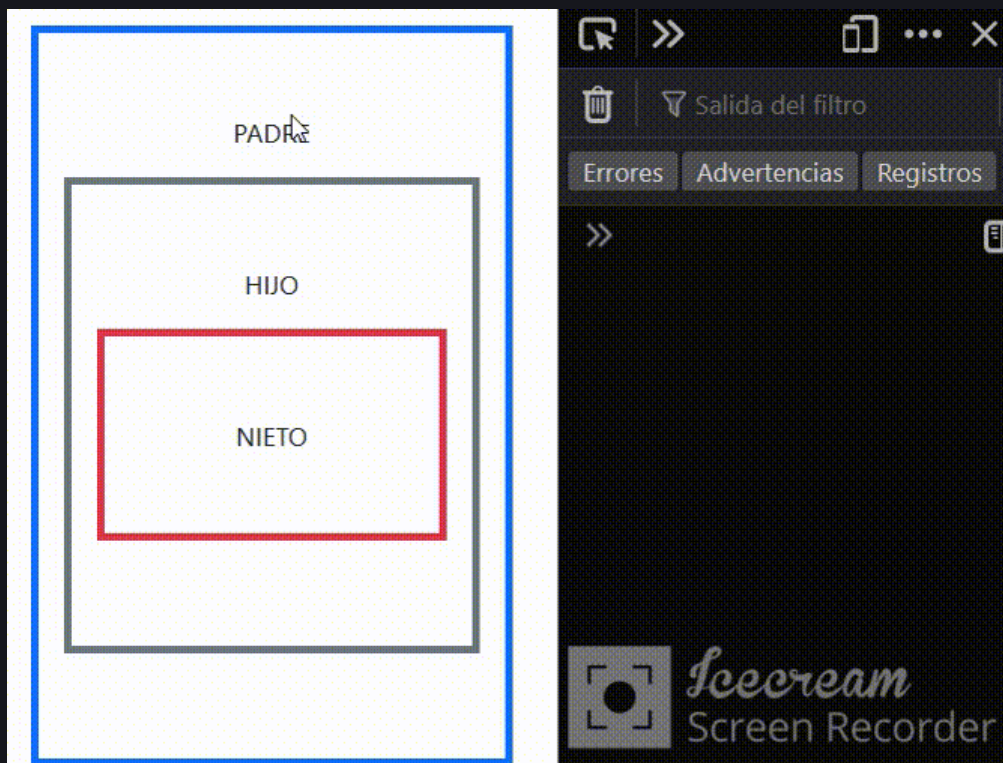
Sintaxis:

```
element.target.matches("SELECTOR_TYPE");
```

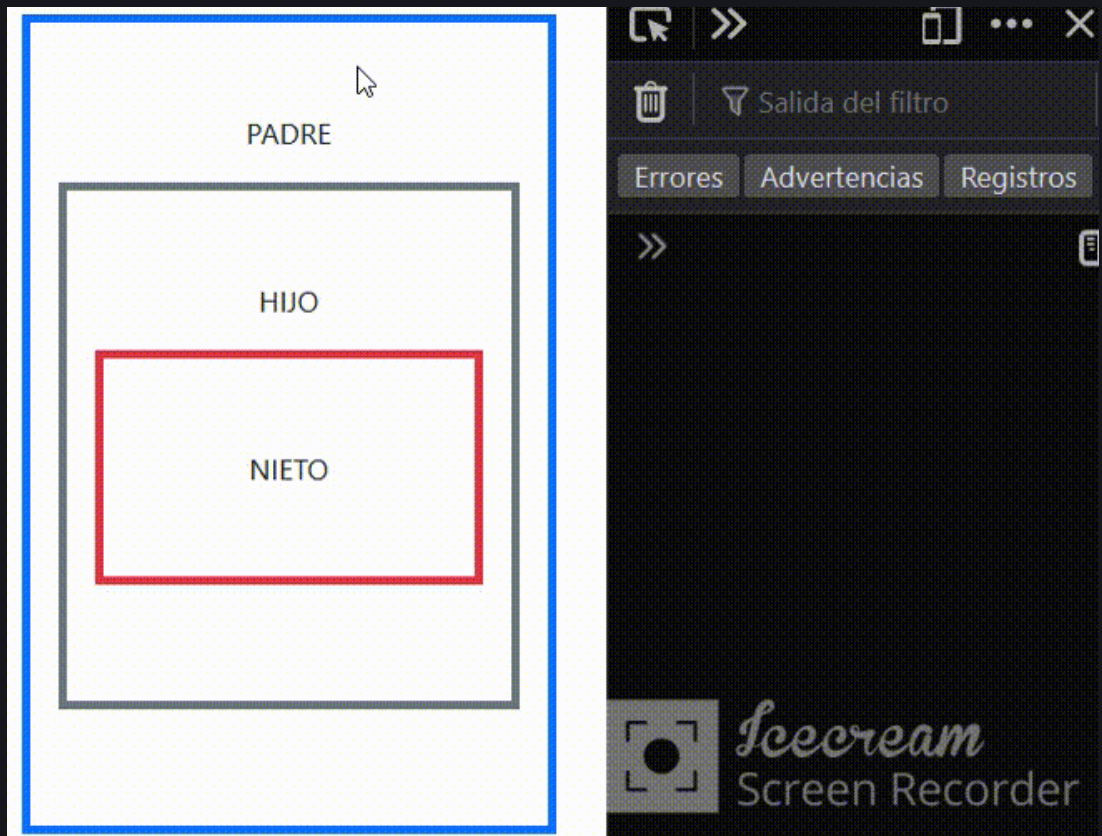
EJEMPLO:

```
container.addEventListener("click", (e) => {"selector"})
```

```
console.log(e.target.matches("#cualquiera"));
console.log(e.target.matches(".cualquiera"));
```



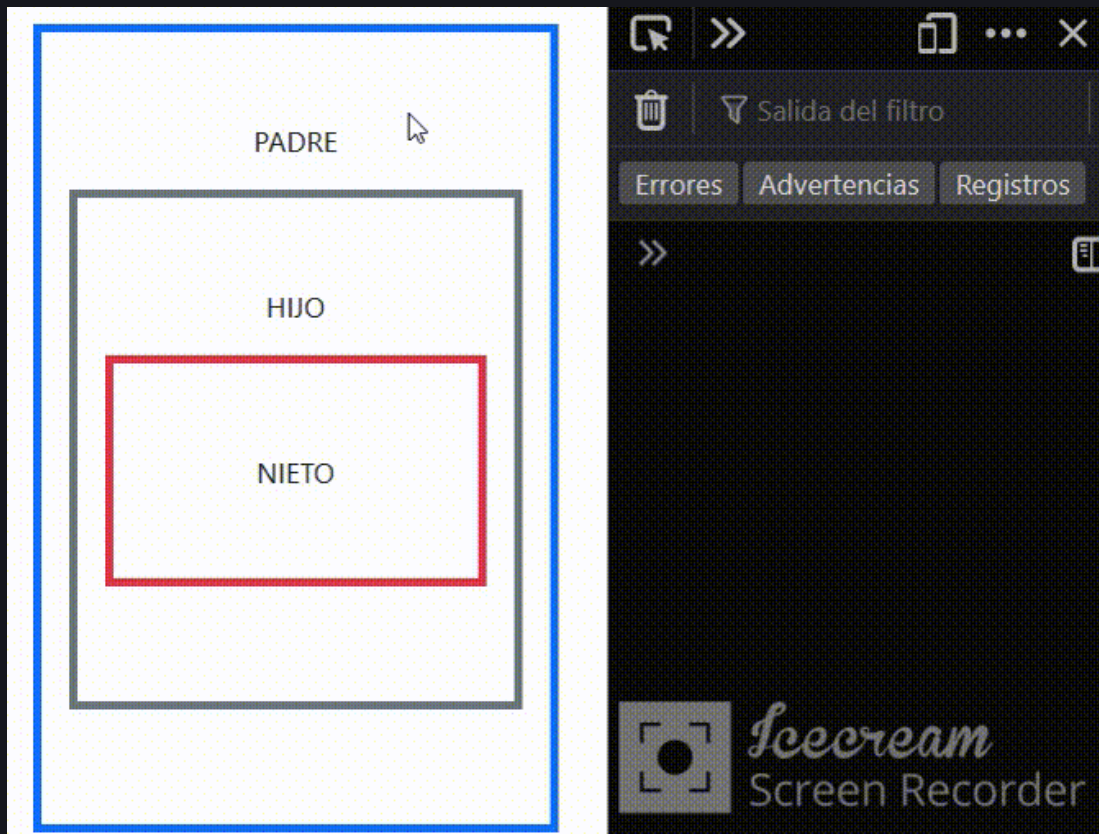

```
console.log(e.target.matches("#padre"));
console.log(e.target.matches(".border-primary"));
```



```
console.log(e.target.matches("#hijo"));
console.log(e.target.matches(".border-secondary"));
```

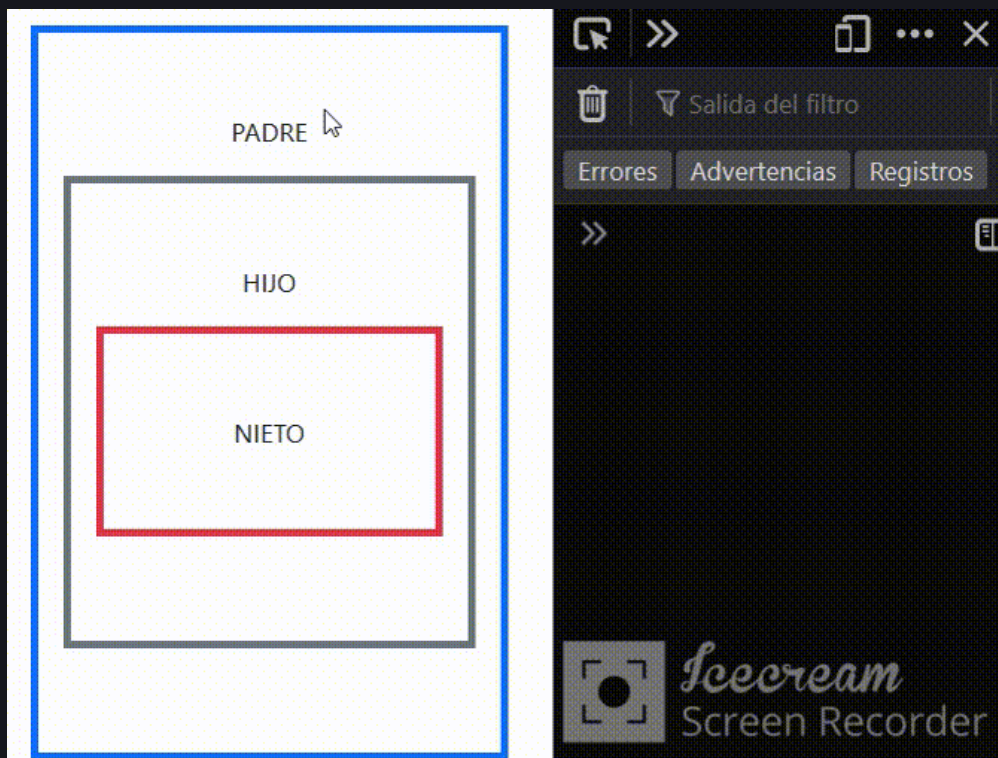


```
console.log(e.target.matches("#nieto"));
console.log(e.target.matches(".border-danger"));
```



```
if (e.target.matches("#padre")){console.log("diste click en el padre")}
if (e.target.matches("#hijo")){console.log("diste click en el hijo")}
if (e.target.matches("#nieto")){console.log("diste click en el nieto")}

if (e.target.matches(".border-primary")){console.log("diste click en el padre")}
if (e.target.matches(".border-secondary")){console.log("diste click en el hijo")}
if (e.target.matches(".border-danger")){console.log("diste click en el nieto")}
```



FAMILIA DE NODOS:

No existe oficialmente el termino familia de nodos en ningún tipo de documentación de JS.

Pero al tener en cuenta que a continuación trataremos con nodos que se llaman:

Childs-siblings-parents

Que literalmente traducidos significan:

Hijos-hermanos-padres

(Nombres que, por supuesto van a tener sentido)

decidí englobar en un mismo título este contenido, algo que por defecto no existe en la documentación oficial.

así que lo llamé familia de nodos y no por mero capricho.

Esta familia de nodos abarca nodos que usualmente están relacionados entre si.

childs:

Los childs vienen a representar cualquier elemento que este contenido en otro elemento.

Por Ejemplo

cuando tenemos varias etiquetas `<p></p>` dentro de etiquetas `<div></div>`

Su nombre viene dado porque la etiqueta que contiene a los childs se le llama parent, por lo tanto los childs serian hijos de la etiqueta padre.

A su vez, a los diferentes childs se los conoce como siblings (hermanos)

Crear childs:

()

appendChild:

Permite crear un elemento hijo a un elemento padre.

Agrega un nuevo nodo al final de la lista de un elemento hijo de un elemento padre especificado.

Si el hijo (Child) es una referencia (hace referencia) hacia un nodo existente en el documento actual, este es quitado del padre actual para ser puesto en el nodo padre nuevo. La clave está en si el (Child) es una referencia a un nodo existente en el documento.

Sintaxis:

```
variable_name.appendChild(child_element);
```

EJEMPLO:

EJEMPLO 1:

HTML:

```
<div id="contenedor"></div>
```

JavaScript:

```
contenedor = document.getElementById("contenedor");  
elemento_nuevo = document.createElement("li");  
contenedor.appendChild(elemento_nuevo);  
console.log(contenedor);
```

```
▼ <div id="contenedor">  
  ► <li> ... </li>  
</div>
```

En este caso podemos insertar elementos "li" en el contenedor "div"

```
elemento_nuevo = document.createElement("ul");  
contenedor.appendChild(elemento_nuevo);  
console.log(contenedor);
```

```
▼ <div id="contenedor">  
  <ul></ul>  
</div>
```

EJEMPLO 2:

HTML:

```
<div id="contenedor"><li>Elemento estático en HTML</li></div>
```

JavaScript:

```
contenedor = document.getElementById("contenedor");  
elemento_nuevo = document.createElement("li");  
elemento_nuevo.textContent = "Elemento desde js"  
contenedor.appendChild(elemento_nuevo);
```

Elemento estatico en HTML
Elemento desde js

EJEMPLO 3:

HTML:

```
<div id="contenedor"></div>
```

JavaScript:

```
const contenedor = document.getElementById("contenedor");  
const frutas = ["Pera", "Fruta del dragon", "manzana", "banana"];  
frutas.forEach(fruta =>{  
    const elemento = document.createElement("li");  
    elemento.textContent = fruta;  
    contenedor.appendChild(elemento)})
```

Pera
Fruta del dragon
manzana
banana

EJEMPLO 4:

Cuando se trata de las etiquetas head y body. Podemos usar el método sin identificarlas previamente.

Esto no es una buena práctica solo una simple dato curioso.

```
parrafo = document.createElement("p");  
parrafo.textContent = "texto de parrafo"  
document.body.appendChild(parrafo);
```

texto de parrafo

```
document.head.appendChild(parrafo);
```

```
<html>  
  ▼ <head>  
    <title>Creacion de Elementos en JavaScript</title>  
    <link rel="shortcut icon" href="#">  
    <p>texto de parrafo</p>  
  </head>  
  ► <body> ... </body>  
</html>
```

se ve por consola ya que el head no es visible para el usuario

Si quisiéramos crear varios childs podríamos tranquilamente utilizar un ciclo u otra solución, pero esto no es una buena práctica ya que genera “reflow”.

El Reflow sucede cuando un navegador debe procesar y pintar parte de, o toda una página web nuevamente,

Como después de actualizar un sitio web interactivo.

Esto enlentece el tiempo de carga y hace más pesada una página innecesariamente,

Y por esta ocasión especial sirve el siguiente método:

Obtener y modificar childs

SINTAXIS GENERAL:
`Variable_name.property;`

firstChild:

selecciona el primer hijo de un elemento padre.

La propiedad firstChild de solo lectura de la interfaz Node devuelve el primer elemento secundario del nodo en el árbol, o nulo si el nodo no tiene elementos secundarios

Sintaxis:

`Variable_name.firstChild;`

EJEMPLO:

HTML:

```
<div id="contenedor">
  <h1>TITULASO</h2>
  <h3>TITULONGO</h3>
  <p>parrafito</p>
</div>
```

TITULASO
TITULONGO
parrafito

JavaScript:

```
contenedor = document.getElementById("contenedor");
primer_hijo = contenedor.firstChild;
```

```
document.write(primer_hijo);
```

[object Text]

```
console.log(primer_hijo);
```

► #text "\n" ⚙

Muestra esto por el espacio que hay...

```
<div id="contenedor">
  <h1>TITULASO</h2>
  <h3>TITULONGO</h3>
  <p>parrafito</p>
</div>
```


Pero si lo elimináramos

```
<div id="contenedor"><h1>TITULASO</h2>  
  <h3>TITULONGO</h3>  
  <p>parrafito</p>  
</div>
```

[object HTMLHeadingElement]

▶ <h1> ⚙

Nos mostraría el primer hijo directo, ya que el texto es también un nodo que puede contar como un hijo.

lastChild:

Nos selecciona el *ultimo* hijo de un elemento padre.

La propiedad lastChild de solo lectura de la interfaz Node devuelve el último elemento secundario del nodo. Si su padre es un elemento, entonces el hijo es generalmente un nodo de elemento, un nodo de texto o un nodo de comentario. Devuelve nulo si no hay nodos secundarios.

Sintaxis:

```
Variable_name.lastChild;
```

EJEMPLO:

HTML:

```
<div id="contenedor">  
  <h1>TITULASO</h2>  
  <h3>TITULONGO</h3>  
  <p>parrafito</p>  
</div>
```

JavaScript:

```
contenedor = document.getElementById("contenedor");  
ultimo_hijo = contenedor.lastChild;  
document.write(ultimo_hijo);
```

[object Text]

```
console.log(ultimo_hijo);
```

▶ #text "\n" ⚙

*Lo mismo pasa aquí.
Muestra esto por el espacio que hay...*

```
<div id="contenedor">  
  <h1>TITULASO</h2>  
  <h3>TITULONGO</h3>  
  <p>parrafito</p>  
</div>
```

Pero si lo elimináramos

```
<div id="contenedor">  
  <h1>TITULASO</h2>  
  <h3>TITULONGO</h3>  
  <p>parrafito</p></div>
```

[object HTMLParagraphElement]

► <p> ⚙

Obtendríamos el ultimo hijo.

firstElementChild:

A diferencia de firstChild, nos devuelve el primer ELEMENTO y no el ~~primer hijo~~.

La propiedad de solo lectura Element.firstElementChild devuelve el primer elemento secundario de un elemento o nulo si no hay elementos secundarios. Element.firstElementChild incluye solo nodos de elementos. Para obtener todos los nodos secundarios, incluidos los nodos que no son elementos, como los nodos de texto y comentarios, use Node.firstChild.

La propiedad de solo lectura Document.firstElementChild devuelve el primer elemento secundario del documento o nulo si no hay elementos secundarios. Para los documentos **HTML**, este suele ser el único elemento secundario, el elemento raíz. Consulte Element.firstElementChild para conocer el primer elemento secundario de elementos específicos dentro de un documento.

La propiedad de solo lectura DocumentFragment.firstElementChild devuelve el primer elemento secundario del fragmento de documento, o nulo si no hay elementos secundarios.

Sintaxis:

Variable_name.firstElementChild;

EJEMPLO:

HTML:

```
<div id="contenedor">  
  <h1>TITULASO</h2>  
  <h3>TITULONGO</h3>  
  <p>parrafito</p>  
</div>
```

JavaScript:

```
contenedor = document.getElementById("contenedor");  
primer_elemento = contenedor.firstElementChild;  
document.write(primer_elemento);
```

[object HTMLHeadingElement]

```
console.log(primer_elemento);
```

► <h1> ⚙

lastElementChild:

A diferencia de lastChild, nos devuelve el último ELEMENTO y no el último hijo.

La propiedad de solo lectura **Element.lastElementChild** devuelve el último elemento secundario de un elemento, o nulo si no hay elementos secundarios. Element.lastElementChild incluye solo nodos de elementos. Para obtener todos los nodos secundarios, incluidos los nodos que no son elementos, como los nodos de texto y comentarios, use Node.lastChild.

La propiedad de solo lectura **Document.lastElementChild** devuelve el último elemento secundario del documento, o nulo si no hay elementos secundarios. Para los documentos **HTML**, este suele ser el único elemento secundario, el elemento raíz.

La propiedad de solo lectura **DocumentFragment.lastElementChild** devuelve el último elemento secundario del fragmento de documento, o nulo si no hay elementos secundarios.

Sintaxis:

```
Variable_name.lastElementChild;
```

EJEMPLO:

HTML:

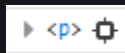
```
<div id="contenedor">
  <h1>TITULASO</h2>
  <h3>TITULONGO</h3>
  <p>parrafito</p>
</div>
```

JavaScript:

```
contenedor = document.getElementById("contenedor");
ultimo_elemento = contenedor.lastElementChild;
document.write(ultimo_elemento);
```

[object HTMLParagraphElement]

```
console.log(ultimo_elemento);
```



childNodes:

La propiedad de solo lectura Node.childNodes devuelve una colección de hijos nodes del elemento dado donde el primer nodo hijo es asignado un índice 0.

Sintaxis:

```
Variable_name.childNodes;
```

EJEMPLO:

HTML:

```
<div id="contenedor">
  <h1>TITULASO</h2>
  <h3>TITULONGO</h3>
  <p>parrafito</p>
</div>
```

JavaScript:

```
contenedor = document.getElementById("contenedor");
nodos_hijos = contenedor.childNodes;
document.write(nodos_hijos);
```

[object NodeList]

```
console.log(nodos_hijos);
```

► NodeList(7) [#text, h1, #text, h3, #text, p, #text]

Cabe aclarar que esto es una "node list" y no un array

children:

muestra la colección de elementos de un padre.

Sintaxis:

```
Variable_name.children
```

EJEMPLO:

HTML:

```
<div id="contenedor">
  <h1>TITULASO</h2>
  <h3>TITULONGO</h3>
  <p>parrafito</p>
</div>
```

JavaScript:

```
contenedor = document.getElementById("contenedor");
hijos = contenedor.children;
document.write(hijos);
```

[object HTMLCollection]

```
console.log(hijos);
```

nos devuelve las etiquetas del contenedor padre pero no su contenido

```
HTMLCollection { 0: h1 , 1: h3 , 2: p , length: 3 }  
  ▶ 0: <h1>  
  ▶ 1: <h3>  
  ▶ 2: <p>  
    length: 3
```

Métodos con childs:

SINTAXIS GENERAL:

```
father_name.method_name();
```

()

replaceChild:

El método **Node.replaceChild()** reemplaza un nodo hijo del elemento especificado por otro.

Sintaxis:

```
father_name.replaceChild(new_child,old_child);
```

EJEMPLO:

HTML:

```
<div id="contenedor">  
  <h1 id="tit1" >TITULO 1</h1>  
  <h2 id="tit2" >TITULO 2</h2>  
  <p id="Parrafo">parrafo</p>  
</div>
```

TITULO 1

TITULO 2

parrafo

JavaScript:

```
contenedor = document.getElementById("contenedor");  
titulo_nuevo = document.createElement("h3");  
titulo_nuevo.innerHTML = "TITULO 3 NUEVO";  
titulo_viejo = document.getElementById("tit2");  
contenedor.replaceChild(titulo_nuevo,titulo_viejo);
```

TITULO 1

TITULO 3 NUEVO

parrafo

De esta forma podemos reemplazar un hijo de algún elemento padre.

()

removeChild:

El método **Node.removeChild()** elimina un nodo hijo del DOM y puede devolver el nodo eliminado.

Sintaxis:

```
father_name.removeChild(child_name);
```

EJEMPLO:

HTML:

```
<div id="contenedor">
  <h1 id="tit1" >TITULO 1</h1>
  <h2 id="tit2" >TITULO 2</h2>
  <p id="Parrafo">parrafo</p>
</div>
```

TITULO 1

TITULO 2

parrafo

JavaScript:

```
contenedor = document.getElementById("contenedor");
parrafo_a_remover = document.getElementById("Parrafo")
contenedor.removeChild(parrafo_a_remover);
```

TITULO 1

TITULO 2

()

hasChildNodes:

El método Node.hasChildNodes() devuelve un valor Boolean indicando si el Node (nodo) actual tiene nodos hijos o no.

Sintaxis:

```
Father_name.hasChildNodes();
```

EJEMPLO:

HTML:

```
<div id="contenedor">
  <h1 id="tit1" >TITULO 1</h1>
  <h2 id="tit2" >TITULO 2</h2>
  <p id="Parrafo">parrafo</p>
</div>
```

JavaScript:

```
contenedor = document.getElementById("contenedor");
consulta = contenedor.hasChildNodes();
document.write(consulta);
```

true

```
console.log(consulta);
```

true

HTML:

```
<div id="elemento_vacio"></div>
```

JavaScript:

```
elemento_vacio = document.getElementById("elemento_vacio");
consulta2 = elemento_vacio.hasChildNodes();
document.write(consulta2);
```

false

```
console.log(consulta2);
```

false

Importante recordar que hay una diferencia entre esto:

```
<div id="elemento_vacio"></div> ** ¿hasChildNodes? → false
```

Y esto:

```
<div id="elemento_vacio"> </div> ** ¿hasChildNodes? → true
```

Y la diferencia esta en el espacio dentro de los cierres de etiqueta "><"

Un espacio: " " cuenta como un nodo texto y eso cuenta como un hijo de un elemento.

Algo que podríamos hacer a partir de esto es establecer condicionales:

JavaScript:

```
contenedor = document.getElementById("contenedor");
consulta = contenedor.hasChildNodes();
if (consulta) {
    document.write("El elemento tiene hijos");
}
else{ document.write("El elemento no tiene hijos");
}

document.write("<br>");

elemento_vacio = document.getElementById("elemento_vacio");
consulta2 = elemento_vacio.hasChildNodes();

if (consulta2) {
    document.write("El elemento tiene hijos");
}
else{ document.write("El elemento no tiene hijos");
}
```

El elemento tiene hijos
El elemento no tiene hijos

siblings:

Cuando varios childs están contenidos en un contenedor
estos se denominan siblings.

Existen propiedades que nos permiten trabajar con esta característica.

SINTAXIS GENERAL:

`Variable_name.property_name;`

nextSibling:

La propiedad de sólo lectura **Node.nextSibling** devuelve el siguiente nodo con respecto al indicado en la lista de nodos ([childNodes](#)) a la que este pertenece o null si el nodo especificado es el último en dicha lista.

Sintaxis:

`Variable_name.nextSibling;`

EJEMPLO:

HTML:

```
<div id="DIV_PADRE">
  <p id="p1">PARRAFO1</p>
  <p id="p2">PARRAFO2</p>
  <p id="p3">PARRAFO3</p>
</div>
```

JavaScript:

```
parrafo1 = document.getElementById("p1");
parrafo2 = document.getElementById("p2");
parrafo3 = document.getElementById("p3");
console.log(parrafo1.nextSibling);
console.log(parrafo2.nextSibling);
console.log(parrafo3.nextSibling);
```

▶ #text "\n"	⚙
▶ #text "\n"	⚙
▶ #text "\n"	⚙

Ya que el nodo más próximo después de cada párrafo es texto en el código

```

  <div id="DIV_PADRE">
    <p id="p1">PARRAFO1</p>
    <p id="p2">PARRAFO2</p>
    <p id="p3">PARRAFO3</p>
  </div>
```

previousSibling:

La propiedad de sólo-lectura `Node.previousSibling` devuelve el nodo inmediatamente anterior al especificado en la lista de nodos `childNodes` de su padre, o null si el nodo especificado es el primero en dicha lista.

Sintaxis:

```
Variable_name.previousSibling;
```

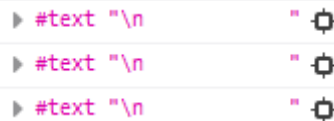
EJEMPLO:

HTML:

```
<div id="DIV_PADRE">
  <p id="p1">PARRAFO1</p>
  <p id="p2">PARRAFO2</p>
  <p id="p3">PARRAFO3</p>
</div>
```

JavaScript:

```
parrafo1 = document.getElementById("p1");
parrafo2 = document.getElementById("p2");
parrafo3 = document.getElementById("p3");
console.log(parrafo1.previousSibling);
console.log(parrafo2.previousSibling);
console.log(parrafo3.previousSibling);
```



```
▶ #text "\n"
▶ #text "\n"
▶ #text "\n"
```

Lo mismo pasa qui porque el primer nodo anterior es texto.

```
<div id="DIV_PADRE">
  <p id="p1">PARRAFO1</p>
  <p id="p2">PARRAFO2</p>
  <p id="p3">PARRAFO3</p>
</div>
```

nextElementSibling:

La propiedad de sólo lectura `NonDocumentTypeChildNode.nextElementSibling` devuelve el elemento inmediatamente posterior al especificado, dentro de la lista de elementos hijos de su padre, o bien `null` si el elemento especificado es el último en dicha lista.

Sintaxis:

`Variable_name.nextElementSibling;`

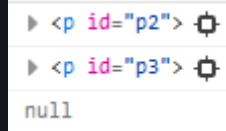
EJEMPLO:

HTML:

```
<div id="DIV_PADRE">
  <p id="p1">PARRAFO1</p>
  <p id="p2">PARRAFO2</p>
  <p id="p3">PARRAFO3</p>
</div>
```

JavaScript:

```
parrafo1 = document.getElementById("p1");
parrafo2 = document.getElementById("p2");
parrafo3 = document.getElementById("p3");
console.log(parrafo1.nextElementSibling);
console.log(parrafo2.nextElementSibling);
console.log(parrafo3.nextElementSibling);
```



```
> <p id="p2">
> <p id="p3">
null
```

Recordemos que aunque después del último P hay un elemento DIV, éste es PADRE Y no es hermano de los elementos P

Por eso obtenemos null al consultar por su siguiente elemento hermano

previousElementSibling:

La propiedad de sólo lectura `NonDocumentTypeChildNode.previousElementSibling` retorna el Element predecesor inmediato al especificado dentro de la lista de hijos de su padre, o bien `null` si el elemento especificado es el primero de dicha lista.

Sintaxis:

`Variable_name.previousElementSibling;`

EJEMPLO:

HTML:

```
<div id="DIV_PADRE">
  <p id="p1">PARRAFO1</p>
  <p id="p2">PARRAFO2</p>
  <p id="p3">PARRAFO3</p>
</div>
```

JavaScript:

```
parrafo1 = document.getElementById("p1");
parrafo2 = document.getElementById("p2");
parrafo3 = document.getElementById("p3");
console.log(parrafo1.previousElementSibling);
console.log(parrafo2.previousElementSibling);
console.log(parrafo3.previousElementSibling);
```



Recordemos que, aunque antes del primer P hay un elemento DIV, éste es PADRE Y no es hermano de los elementos P

Por eso obtenemos null al consultar por su siguiente elemento hermano

parents:

Como mencionamos antes, los contenedores de los childs son conocidos como parents(padres)

Existen maneras de poder trabajar con estos tambien.

parentElement:

La propiedad de sólo lectura de `Nodo.parentElement` devuelve el nodo padre del DOM Element, o null, si el nodo no tiene padre o si el padre no es un Element DOM .

Sintaxis:

`Element_name.parentElement`

EJEMPLO:

HTML:

```
<div id="DIV_PADRE">
  <p id="PARRAFO_HIJO"></p>
</div>
```

JavaScript:

```
cuerpo = document.getElementById("BODY");
div = document.getElementById("DIV_PADRE");
parrafo = document.getElementById("PARRAFO_HIJO");

console.log(div.parentElement)
console.log(parrafo.parentElement)
console.log(cuerpo.parentElement)
```

```
▶ <body id="BODY"> ☐
```

```
▶ <div id="DIV_PADRE"> ☐
```

```
▶ <html lang="en"> ☐
```

parentNode:

La propiedad de sólo lectura `node.parentNode` devuelve el padre del nodo especificado en el árbol.

Sintaxis:

`Element_name.parentNode`

EJEMPLO:

HTML:

```
</head>
<body id="BODY">
  <div id="DIV_PADRE">
    <p id="PARRAFO_HIJO"></p>
  </div>
  <script src="propiedades de padres.js"></script>
</body>
</html>
```

JavaScript:

```
cuerpo = document.getElementById("BODY");
div = document.getElementById("DIV_PADRE");
parrafo = document.getElementById("PARRAFO_HIJO");

console.log(div.parentNode);
console.log(parrafo.parentNode);
console.log(cuerpo.parentNode);
```



*Si bien el resultado es el mismo que mostraría el parentElement
No siempre va a ser así ya que se puede dar la situación en la que el elemento padre de un
elemento es un nodo y no un elemento.*

Creación de elementos:

Existen diversos métodos para crear o editar elementos *HTML* desde *JavaScript*

SINTAXIS GENERAL:

```
document.method_name();  
( )
```

createElement:

crea un elemento *HTML* especificado por su tagName o un [HTMLUnknownElement \(en-US\)](#) si su tagName no se reconoce.

Sintaxis:

```
document.createElement("tagName");
```

EJEMPLOS:

HTML:

```
<div id="contenedor"></div>
```

EJEMPLO 1:

JavaScript:

```
contenedor = document.getElementById("contenedor");  
elemento_nuevo = document.createElement("LI");  
document.write(elemento_nuevo);
```

```
[object HTMLLIElement]
```

```
console.log(elemento_nuevo);
```

```
▶ <li>
```

Se puede aplicar claramente a cualquier elemento HTML.

EJEMPLO 2:

```
elemento_nuevo = document.createElement("head");  
document.write(elemento_nuevo);  
console.log(elemento_nuevo);
```

```
[object HTMLHeadElement]
```

```
▶ <head>
```

[Referencia de Elementos HTML:](#)

<https://developer.mozilla.org/es/docs/Web/HTML/Element>

()

createTextNode:

crea un nodo de texto

Sintaxis:

```
document.createTextNode("text");
```

EJEMPLO:

JavaScript:

```
texto = document.createTextNode("este es un nodo texto creado");  
document.write(texto);
```

[object Text]

```
console.log(texto);
```

▶ #text "este es un nodo texto creado"

createDocumentFragment:

Crea un nuevo [DocumentFragment](#) vacío, dentro del cual un nodo del DOM puede ser adicionado para construir un nuevo árbol DOM fuera de pantalla.

Esto ahorra muchos recursos en el navegador ya que al agregar elementos no se adhieren arriba del DOM creado, sino que se vuelve a crear desde 0 y ahora agregando el elemento.

Con este método nosotros agregamos el elemento sin tener que crear el DOM de nuevo y evitando reflow

Sintaxis:

```
document.createDocumentFragment();
```

EJEMPLO:

HTML:

```
<div id="contenedor"></div>
```

JavaScript:

```
contenedor = document.getElementById("contenedor");  
fragmento = document.createDocumentFragment();
```



```
for (let i = 1; i <= 5; i++) {  
  parrafo = document.createElement("p");  
  parrafo.innerHTML = `soy el parrafo ${i}`;  
  fragmento.appendChild(parrafo);  
}  
contenedor.appendChild(fragmento);  
console.log(fragmento);
```

soy el parrafo 1
soy el parrafo 2
soy el parrafo 3
soy el parrafo 4
soy el parrafo 5

► DocumentFragment []

Otra forma de crear elementos:

```
const contenedor = document.getElementById("contenedor");  
const fragmento = document.createDocumentFragment();  
const colores = ["rojo", "verde", "azul", "amarillo"];  
  
colores.forEach((color) =>{  
  const li = document.createElement("li");  
  li.textContent = color;  
  fragmento.appendChild(li);  
})
```

rojo
verde
azul
amarillo

Nodos:

()

closest:

El método `closest()` de la interfaz `Element` devuelve el ascendiente más cercano al elemento actual (o el propio elemento actual) que coincida con el selector proporcionado por parámetro. Si no existe dicho ascendiente, devuelve `null`.

Sintaxis:

```
selector_name.closest("selector");
```

EJEMPLO:

EJEMPLO 1:

HTML:

```
<div class="clase">
  <ul class="clase">
    <p class="clase">
      <b class="b"></b>
    </p>
  </ul>
</div>
```

JavaScript:

```
negrita = document.querySelector(".b")
console.log(negrita.closest(".b"));
console.log(negrita.closest(".clase"));
console.log(negrita.closest("claseInexistente"));
```

```
▶ <b class="b">
▶ <p class="clase">
null
```

en su defecto si no encuentra ningún elemento con el selector especificado como parámetro devuelve `null`.

O devuelve el mismo elemento en caso de que sea el único con ese selector

EJEMPLO 2:

HTML:

```
<div class="clase">
  <ul class="clase">
    <p><b class="b"></b></p>
  </ul>
</div>
```

JavaScript:

```
negrita = document.querySelector(".b")
console.log(negrita.closest(".clase"));
```

```
▶ <ul class="clase"> 🔍
```

EJEMPLO 3:

HTML:

```
<div class="clase">
  <ul><p><b class="b"></b></p></ul>
</div>
```

JavaScript:

```
negrita = document.querySelector(".b")
console.log(negrita.closest(".contenedor"));
```

```
▶ <div class="clase"> 🔍
```

()

cloneNode:

Crea una copia de un nodo con sus valores y atributos; y lo retorna

Si se le da el parámetro `true` también clona los hijos del elemento

Sintaxis:

```
selector_name.cloneNode(value);
```

EJEMPLO:

HTML:

```
<div id="contenedor">
  <h1>Titulo h1</h1>
  <h2>Titulo h2</h2>
  <h3>Titulo h3</h3>
  <p>parrafo</p>
</div>
```

JavaScript:

```
const contenedor = document.getElementById("contenedor");
const clon = contenedor.cloneNode();
document.write(clon);
console.log(clon);
```

[object HTMLDivElement]

▶ <div id="contenedor">

```
const clonTrue = contenedor.cloneNode(true);
```

asignando el valor `true` también se copian los hijos del elemento clonado.

[object HTMLDivElement]

```
▼ <div id="contenedor">
  <h1>TITULASO</h1>
  <h3>TITULONGO</h3>
  <p>parrafito</p>
</div>
```

Puede ser que dependiendo del navegador podamos ver los hijos o no

Validaciones con JavaScript:

Las validaciones son un aspecto muy importante dentro de un código.

Las mismas pueden ocurrir:

Del lado del *cliente (frontend)* o del *servidor (backend)*.

Las validaciones del lado del cliente, son imprescindibles; por ejemplo, cuando el cliente debe ingresar datos específicos.

Como vimos anteriormente en el apartado de atributos de inputs

Las validaciones del lado del servidor son cruciales y nunca deben de ser erróneas, pues estas se registran generalmente en bases de datos donde la información es tratada posteriormente, obstante ingresar información indebida es equivalente a desatar un posible efecto domino de errores.

HTML 5 tiene validaciones incorporadas a expensas de *JavaScript*

Que son las validaciones de los atributos de input, pero no son una forma segura de validar datos rigurosamente.

Por otro lado, es posible darle más rigurosidad a estas validaciones agregando código *JavaScript*.

EJEMPLO:

En estos ejemplos se trabajará con expresiones regulares, si no está familiarizado con el uso de las mismas puede informarse aquí:

<https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Regular Expressions>

HTML:

```
<form id="formulario">
  <input id="input" type="text">
  <button type="submit" >SEND</button>
</form>
```

JavaScript:

```
const formulario = document.getElementById("formulario");
const input = document.getElementById("input");
const regUserName = /^[A-Za-zÑñÁáÉéÍíÓóÚúÜü\s]+$/;

formulario.addEventListener("submit", (e)=>{
  e.preventDefault();
  if (!regUserName.test(input.value)) {
    console.log('Formulario no valido');
    return;}
  console.log("formulario enviado");
});
```

Probemos varios ejemplos:

<input type="text" value="hola"/> <input type="button" value="SEND"/>	<input type="text" value="hola buenas tardes"/> <input type="button" value="SEND"/>	<input type="text" value="khomo ézstas"/> <input type="button" value="SEND"/>
<input type="text" value="Todo bien?"/> <input type="button" value="SEND"/>	<input type="text" value="._. que paso????!!"/> <input type="button" value="SEND"/>	

formulario enviado

3

Formulario no valido

2

cómo se puede observar hay tres formularios enviados y dos formularios no validos

Ya que la expresión regular declarada anterior solo acepta letras y no símbolos.

FormData:

Permite de forma sencilla representar parejas clave valor de un formulario

Donde la clave hace referencia al atributo **name** y valor se lo da el usuario al llenar el campo del input

No es necesario el ID pero si necesario el atributo name.

HTML:

```
<form id="formulario">
<input name="nombre" type="text">
<input name="email" type="email">
<button type="submit">SEND</button>
</form>
```

JavaScript:

```
const formulario = document.getElementById("formulario");
const campos = new FormData(formulario);

formulario.addEventListener("submit", (e)=>{
  e.preventDefault();
  for(let item of campos){
    console.log(item);
  }
});
```

```
console.log(campos.get("nombre"));
console.log(campos.get("email"));
});
```

► Array ["nombre", "Pedro"]

► Array ["email", "Pedro@jorje.com"]

Pedro

Pedro@jorje.com