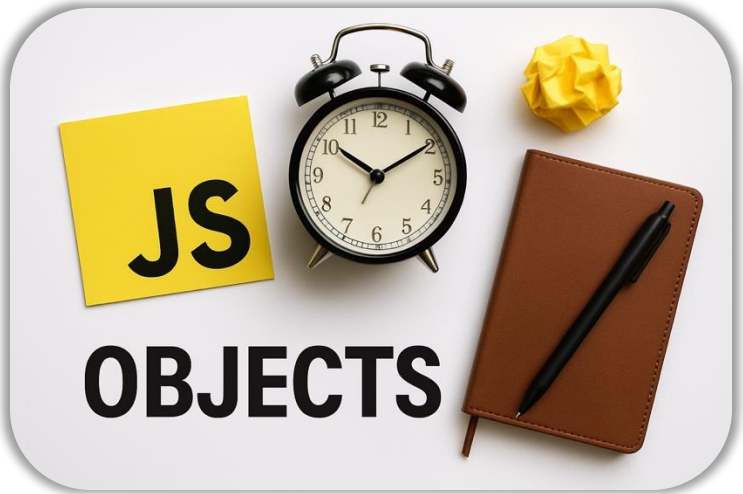
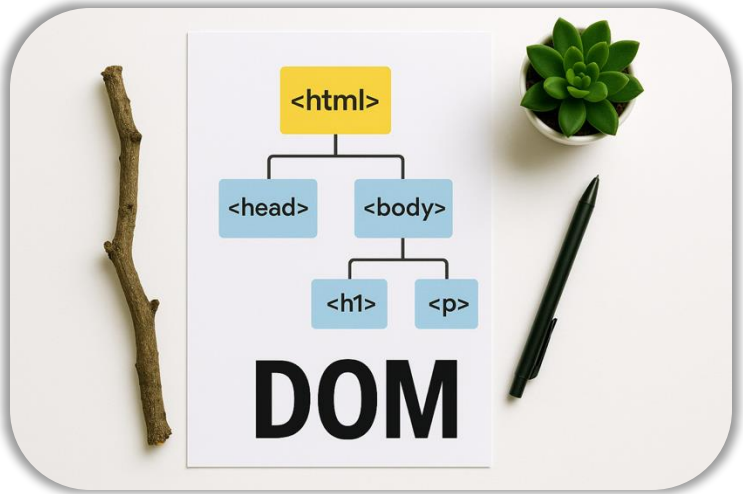
PROGRAMACIÓN WEB – SEMINARIO 6

Javascript intermedio

JavaScript

Dr. José Raúl Romero Salguero
jrromero@uco.es


JS
OBJECTS


<html>
<head>
<body>
<h1>
<p>
DOM


Click
EVENT
HANDLING

S6-1.

Objetos en Javascript

Definición de objetos

Objetos Literales

- Javascript puede ser un lenguaje orientado a objetos, pero sin definir clases
- En su lugar, se puede declarar objeto literal, que luego servirá de base al resto

```
{ property1 : value1 , property2 : value2 , . . . }
```

donde `property1`, `property2` son nombres de propiedad y *`value1`*, *`value2`* son valores (expresiones)

```
var person1 = {
  age:    (30 + 2),
  gender: 'male',
  name:   {first : 'John', last : 'Doe'},
  interests: ['music', 'sports'],
  hello:  function () {
            return 'Hello! My name is' + ' '  + this.name.first + ' '
            + this.name.last;
  } };
```

```
person1.age              --> 32
person1['gender']        --> 'male'
person1.name.first       --> 'John'
person1 ['name']['last']--> 'Doe'
```

`this` fuera de contexto no apunta al objeto actual

```
person1.fullname  --> " undefined undefined "
person1.fullname2 --> " undefined undefined "
```

Objetos Literales

```
var person1 = {
  name:   {first : 'John', last : 'Doe'},
  hello:  function () {
            return 'Hello! My name is' + ' '  + this.name.first + ' '
            + this.name.last;
          },
  fullname: this.name.first + ' ' + this.name.last,
  fullname2: name.first + ' ' + name.last
};
```

En `person1.hello()` el contexto de ejecución de `hello()` es `person1`

~ `name.first` **no** se refiere a `person1.name.first`

y `this.name.first` se refiere a `window.name.first` (que no existe)

this en Javascript

```
const persona = {
  nombre: "Laura",
  saludar() {
    console.log(this.nombre);
  }
};

persona.saludar(); // "Laura"
```

- En Javascript, `this` no depende de dónde está escrita la función, sino de cómo se invoca
- `this` se decide en el momento de la llamada
- En JavaScript moderno siempre deberíamos usar "`use strict`"; o módulos ES, donde el comportamiento de `this` es más seguro
  - En módulos ES, `this` es `undefined` por defecto
- *Evitar funciones flecha en métodos, ya que heredan el* `this` *del contexto exterior*
- Si una función se llama sin un objeto delante, `this` no es el objeto global, sino `undefined`, lo que evita errores silenciosos

¿Cómo funciona **`this`**?

Constructores de objetos

- En lugar de definir una clase, podemos definir una función que actúa como constructor de objetos
  - Las variables declaradas dentro de la función serán propiedades del objeto
    - Cada objeto tendrá su propia copia de estas variables
    - Es posible hacer que tales propiedades sean privadas o públicas
  - Las funciones internas (inner functions) serán métodos del objeto
    - Es posible hacer tales funciones / métodos privados o públicos
- Cada vez que se llama a un constructor de objetos, con el prefijo **`new`**, luego
  1. se crea un nuevo objeto,
  2. la función se ejecuta con la palabra clave `this` que está vinculada a ese objeto

Objetos: definición y usos

```
function EjemploObject () {
  var instVar2 = 'B';   //privada
  var instVar3 = 'C';   //privada
  this.instVar1 = 'A';  //publica
  this.method1 = function () {   // metodo publico
    // El uso de variables públicas precedido por 'this'
    return 'm1[' + this.instVar1 + ']' + method3();   }

  this.method2 = function () {   // metodo publico
    // El uso de un método público precedido por 'this'
    return 'm2 [' + this.method1 () + ']';  }

  var method3 = function () {    // metodo privado
    return 'm3 [' + instVar2 + ']' + method4 ();  }

  var method4 = function () {    // metodo privado
    return 'm4 [' + instVar3 + ']';          }
}

let obj = new EjemploObject ();
```

Los métodos definidos dentro del constructor **se duplican en memoria** por cada instancia

Muchos programadores prefieren utilizar *syntax sugar* como `class`

Objetos: definición y usos

```
function EjemploObject () {
  var instVar2 = 'B'; //privada
  var instVar3 = 'C';   //privada
  this.instVar1 = 'A';  //publica
  this.method1 = function () {   // metodo publico
    // El uso de variables públicas precedido por 'this'
    return 'm1[' + this.instVar1 + ']' + method3();   }

  this.method2 = function () {   // metodo publico
    // El uso de un método público precedido por 'this'
    return 'm2 [' + this.method1 () + ']';  }

  var method3 = function () {    // metodo privado
    return 'm3 [' + instVar2 + ']' + method4 ();  }

  var method4 = function () {    // metodo privado
    return 'm4 [' + instVar3 + ']';          }
}

let obj = new EjemploObject ();
```

Las variables de instancia (propiedades) pueden almacenar valores tipo cadena o funciones

Cada objeto almacena su propia copia de los métodos

Prototipo

Objetos: propiedad prototipo

- Todas las funciones tienen una propiedad `prototype` que puede contener propiedades y métodos de objetos compartidos.
  - ~ los objetos no almacenan sus propias copias de estas propiedades y métodos, sino que solo almacenan referencias a una sola copia

Código utilizado para conocer funcionamiento interno. No moderno.

```
function EjemploObject () {
  this.instVar1 = 'A'; // propiedad publica
  var instVar2 = 'B';  // propiedad privada
  var instVar3 = 'C';  // propiedad privada

  EjemploObject.prototype.method1 = function () { ... } // método publico
  EjemploObject.prototype.method2 = function () { ... } // método publico

  var method3 = function () { ... } // metodo privado
  var method4 = function () { ... } // metodo privado
}
```

- Las propiedades y métodos `prototype` son **siempre públicos**

Objetos: propiedad prototipo

```
obj1 = new EjemploObject ();
obj2 = new EjemploObject ();
document.writeln(obj1.instVar4);  // undefined
document.writeln(obj2.instVar4);  // undefined

EjemploObject.prototype.instVar4 = 'A';
document.writeln(obj1.instVar4);  // 'A'
document.writeln(obj2.instVar4);  // 'A'

EjemploObject.prototype.instVar4 = 'B';
document.writeln(obj1.instVar4);  // 'B'
document.writeln(obj2.instVar4);  // 'B'

obj1.instVar4 = 'C'; // crea una nueva propiedad para obj1

EjemploObject.prototype.instVar4 = 'D';
console.log(obj1.instVar4);  // 'C' !!
console.log(obj2.instVar4);  // 'D' !!
```

- La propiedad `prototype` se puede modificar "sobre la marcha"
  - ~ todos los objetos ya existentes obtienen nuevas propiedades/métodos
  - ~ la manipulación de propiedades / métodos asociados con la propiedad `prototype` debe hacerse con cuidado

```
obj3 = new EjemploObject ();

obj3.instVar4 == ??
```

Objetos: propiedad prototipo

```
// Se puede modificar prototype posterior a su declaracion
EjemploObject.prototype.instVar5 = 'E';
EjemploObject.prototype.setInstVar5 = function (arg) {
  this.instVar5 = arg;
}

obj1 = new EjemploObject ();
obj2 = new EjemploObject ();
obj2.setInstVar5 ('F');

console.log(obj1.instVar5);    // ??
console.log (obj2.instVar5);   //  ??
```

Tengamos en cuenta que…

- Históricamente, en **JavaScript** los métodos se añadían definiéndolos en `Function.prototype`
- Desde ECMAScript 2015 (ES6), la forma habitual para los programadores es **usar clases**, que generan automáticamente los métodos en el `prototype` de forma clara y compacta
- La sintaxis `Fn.prototype.metodo = ...` sigue siendo válida y es útil para entender **cómo funciona JavaScript internamente**, pero no es la forma preferida hoy en día (compleja y verbosa)
- El uso de *syntax sugar* (**capa sintáctica**) evita errores

```
class Rectangulo {
  constructor(ancho, alto) {
    this.ancho = ancho;
    this.alto = alto;
    this.tipo = 'Rectangulo'; }

  get area() {
    return this.ancho * this.alto; }
}

class Cuadrado extends Rectangulo {
  constructor( largo ) {
    super(largo, largo);
    this.tipo = 'Cuadrado'; }
}

var c1 = new Cuadrado(5);
document.writeln("El area de c1 es " + c1.area );
```

El area de c1 es 25

'Syntactic Sugar' para clases

`class` sirve para definir una plantilla a partir de la que crear objetos JS.

El método **`constructor`** se invoca automáticamente cuando se crea un objeto nuevo. Si no se indica, lo creará vacío.

https://www.w3schools.com/js/tryit.asp?filename=tryjs_class_method

Objetos en Javascript moderno

- Desde ES2022, los **campos privados** se añaden con `#field`
- Los **métodos** se almacenan en el `prototype` por defecto (más eficiente)

```
class Persona {
  #id;              // campo privado moderno
  constructor(nombre, id) {
    this.nombre = nombre;
    this.#id = id;
  }
  getId() { return this.#id; }
  saludo() { return `Hola, soy ${this.nombre}`; }
}
```

Los métodos privados también utilizan #

Solo son accesibles dentro de la misma clase.

No se pueden invocar desde instancias (persona.#calcChecksum() → **Error**).

No aparecen en el prototype.

```
class Persona {
 #id;                  // campo privado
 #calcChecksum() {     // método privado – presupone que #id es string
  return this.#id.split("").reduce((a, c) => a + c.charCodeAt(0), 0);
 }

 constructor(nombre, id) {
  this.nombre = nombre;
  this.#id = id;
 }

 getId() {
  return this.#id;
 }

 getChecksum() {
  return this.#calcChecksum();   // se puede usar dentro de la clase
 }

 saludo() {
  return `Hola, soy ${this.nombre}`;
 }
}
```

Objetos en Javascript moderno

En ES6, los métodos definidos dentro del bloque `class` se colocan automáticamente en `EjemploObject.prototype`, igual que hacíamos antes manualmente, pero con una sintaxis mucho más limpia.

```
class EjemploObject {
  constructor() {
    this.instVar1 = 'A';
  }

  setInstVar5(arg) {
    this.instVar5 = arg;
  }


  method1() {
    return "método 1";
  }
}
```

```
// Agregar métodos después es posible
// pero no es aconsejable mezclar notaciones

EjemploObject.prototype.extra = function () {
  return "método añadido a posteriori";
};
```

Objetos en Javascript moderno

Cuando se utiliza `extends`, la clase B extiende a A, por lo que el prototipo de B hereda del prototipo de A.

```
obj --> B.prototype --> A.prototype --> Object.prototype --> null
```

```
class A {
  metodoA() {}
}

class B extends A { // Herencia utilizando syntactic sugar
  metodoB() {}
}

const obj = new B();

console.log(obj.__proto__ === B.prototype);        // true
console.log(B.prototype.__proto__ === A.prototype); // true
```

Esta cadena de *prototypes* es lo que en realidad implementa la herencia en JavaScript

S6-2.
DOM

Objetos globales en DOM

Los seis objetos DOM globales

Cada programa Javascript puede referirse a los siguientes objetos globales:

BOM
Browser Object Model

nombre	descripción
`document`	página HTML actual y su contenido
`history`	lista de páginas que el usuario ha visitado
`location`	URL de la página HTML actual
`navigator`	información sobre el navegador web que está utilizando
`screen`	información sobre el área de la pantalla ocupada por el navegador
`window`	la ventana del navegador

El objeto `document`

Es el documento actual y punto de entrada a los elementos del DOM (*Document Object Model*)

- Propiedades:
  - `anchors, body, cookie, domain, forms, images, links, referrer, title, URL`
- Métodos:
  - `getElementById` → `element`
  - `getElementsByName, getElementsByTagName`
  - `querySelector, querySelectorAll`
  - `close, open, write, writeln`

El objeto `history`

Es la lista de sitios que el navegador ha visitado en esta ventana.

- Propiedades:
  - `length`
- Métodos:
  - `back, forward, go`

A veces –por seguridad– el navegador no permite que los *scripts* vean las propiedades del historial

https://developer.mozilla.org/es/docs/Web/API/History

El objeto `location`

Es la localización de la página web actual

- Propiedades:
  - `host, hostname, href, pathname, port, protocol, search`
- Métodos:
  - `assign(url)` – carga un nuevo documento
  - `reload(Boolean)` – recarga la página del servidor (true) o de la cache (false) enviando un GET forzado
  - `replace(url)` – como `assign` pero eliminando la URL actual del historial

El objeto `location`

```
<html>
<head>
<script>
function newDoc() {
  window.location.assign("https://www.w3schools.com")
}
</script>
</head>
<body>

<input type="button" value="Load new document" onclick="newDoc()">

</body>
</html>
```

Fuente: w3schools.com

El objeto `history`

Es la lista de sitios que el navegador ha visitado en esta ventana.

- Propiedades:
  - `length`
- Métodos:
  - `back, forward, go`

A veces –por seguridad– el navegador no permite que los *scripts* vean las propiedades del historial

https://developer.mozilla.org/es/docs/Web/API/History

El objeto `navigator`

Propiedades del objeto `navigator`:

`navigator.appName`	Nombre del navegador
`navigator.appVersion`	Versión del navegador
`navigator.language`	Idioma del navegador
`navigator.cookiesEnabled`	Indica si tiene cookies activadas

```
<html>
<head>
<title>Navigator example</ title>
<script>
if (navigator.appName == 'Netscape') {
  document.writeln('<link rel=stylesheet type="text/css" href= "netscape.css">');
} else if (navigator.appName == 'Opera') {
  document.writeln('<link rel=stylesheet type="text/css" href="Opera.css">');
} else {
  document.writeln('<link rel=stylesheet type ="text/css" href="Other.css">');
}
</ script >
</ head >
...
```

El objeto `screen`

Es información sobre la pantalla de visualización del cliente

- Propiedades:
  - `height, width`
  - `availHeight, availWidth`
  - `colorDepth, pixelDepth`

El objeto `window`

Un objeto `window` representa una ventana abierta en un navegador

- Si un documento contiene `frames`, entonces hay
  - un objeto `window` para el documento HTML
  - un objeto `window` adicional para cada marco, accesible a través de una matriz `window.frames`
- Un objeto `window` tiene **propiedades** que incluyen:
  - `innerHeight`, `innerWidth` – altura, anchura interior de la ventana del navegador en px (**no** incluye barras de herramientas ni de *scroll*)

```
var w = window.innerWidth || document.documentElement.clientWidth
|| document.body.clientWidth;

var h = window.innerHeight || document.documentElement.clientHeight
|| document.body.clientHeight;
```

  - Multitud de eventos `window.on`**X**...
  - Objetos `document`, `screen`, `navigator`, `location`, etc.

https://www.w3schools.com/jsref/met_win_open.asp

El objeto `window`

Los métodos proporcionados por un objeto `window` incluyen

- `open(url,name[,features])` abre una nueva ventana / pestaña del navegador y devuelve una referencia a un objeto de ventana
  - `url` es la URL para acceder en la nueva ventana; puede ser la cadena vacía ("`about:blank`")
  - `name` es un nombre dado a la ventana para referencia posterior (`_blank`, `_parent`, `_self`, `_top`, o un *nombre* para la ventana nueva)
  - `features` es una cadena (sin espacios) de características (atributo=valor) de la ventana:

    height=pixels, left=pixels, menubar=yes|no|1|0, scrollbars=yes|no|1|0, status=yes|no|1|0, titlebar=yes|no|1|0, top=pixels, width=pixels

NOTA: La secuencia estándar para la creación de una nueva ventana no es instanciar un nuevo objeto `window` con `new`, sino invocar a `window.open()`

El objeto `window`

Los métodos proporcionados por un objeto de ventana incluyen:

- `close`() – cierra una ventana / pestaña del navegador
- `focus`() – enfoca una ventana (acercar la ventana al frente)
- `blur`() – elimina el foco de una ventana (mueve la ventana detrás de otras)
- `scrollBy`(), `scrollTo`() – realiza un *scroll* de la ventana en ***x,y*** *px* (o desplaza a una posición dada la barra)

```
<!DOCTYPE html>
<html>
<body>

<p>Pulsar el bot&oacute;n para abrir una ventana y darle el foco.</p>

<button onclick="abrirVentana()">Pulsar aqu&iacute;</button>

<script>
function abrirVentana() {
  var vent = window.open("", "", "width=200, height=100");
  vent.document.write("<p>Nueva ventana</p>");
  vent.focus(); //Usar blur() para no darle el foco
}
</script>

</body>
</html>
```

El objeto window__ ejemplo

```
<html lang ="es-ES">
<head>
 <title>Lanzador de ventana de ayuda</title>
 <script>
function lanzarAyuda() {
 var msjAyuda = window.open('','Ayuda',
'height=300,width=150,menubar=no,scrollbars=no,status=no,titlebar=no,location=no,toolbar
=no');

 with (msjAyuda.document) {
  writeln ("<!DOCTYPE html><html><head><title> Help </title>\
            </head><body>Esto podr&iacute;a ser ayuda contextual, un mensaje u otra
alerta, seg&uacute;n el estado de la p&aacute;gina invocante.</body></html>");
  close();
 }
}
 </script>
</head>
<body>
 <form name ="ButtonForm" id ="ButtonForm" action ="">
   <p><input type ="button" value="Click for Help" onclick=" lanzarAyuda();"></p>
 </form>
</body>
</html>
```

Cajas de diálogo del objeto `window`

A menudo solo queremos abrir una nueva ventana para:

- mostrar un mensaje
- pedir confirmación de una acción
- solicitar una entrada

Para estos fines, el objeto `window` en JS proporciona mecanismos predefinidos para el manejo de "`dialog boxes`" (cuadros de diálogo simples):

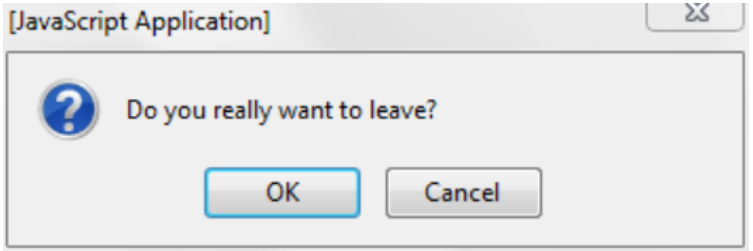
- `null alert(msg_str)`
- `bool confirm(msg_str)`
- `string prompt(msg_str, default_value)`

Cajas de diálogo del objeto `window`

Para estos fines, el objeto `window` en JS proporciona mecanismos predefinidos para el manejo de "`dialog boxes`" (cuadros de diálogo simples):

- `null alert(msg_str)`

- `bool confirm(msg_str)`



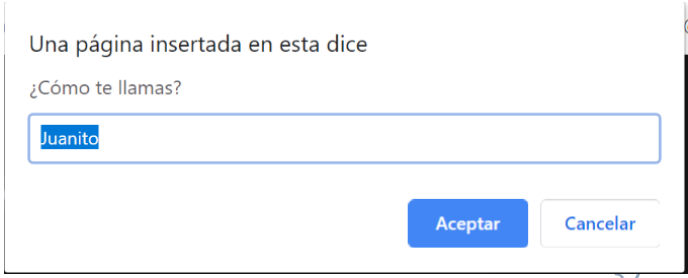
Cajas de diálogo del objeto `window`

`string prompt(msg_str, default_value)`

- Crea un cuadro de diálogo que muestra `msg_str` y un área de entrada
- Si se indica `default_value`, se mostrará en el campo de entrada
- Muestra dos botones 'Cancel' and 'OK'
- Si el usuario selecciona 'OK', el valor actual ingresado en el campo de entrada se devuelve como `String`; de lo contrario, se devuelve un valor `null`

Ejemplo:

```
var nombre = prompt("¿Cómo te llamas?",
"Juanito");
```



Exploración del árbol DOM

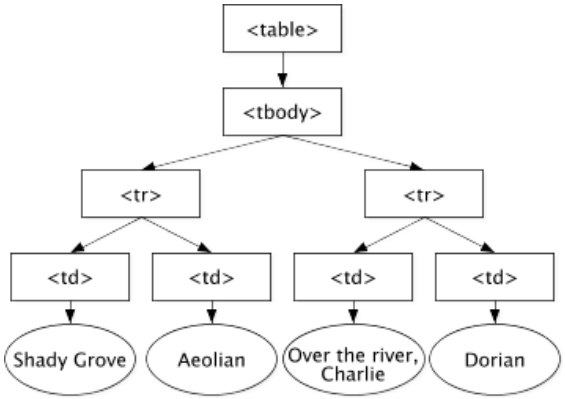
Árbol DOM

Ejemplo:

La siguiente tabla HTML

```
<table>
  <tbody>
    <tr>
      <td>Shady Grove</td>
      <td>Aeolian</td>
    </tr>
    <tr>
      <td>Over the River, Charlie</td>
      <td>Dorian</td>
    </tr>
  </tbody>
</table>
```

DOM representa la tabla como un árbol de nodos:

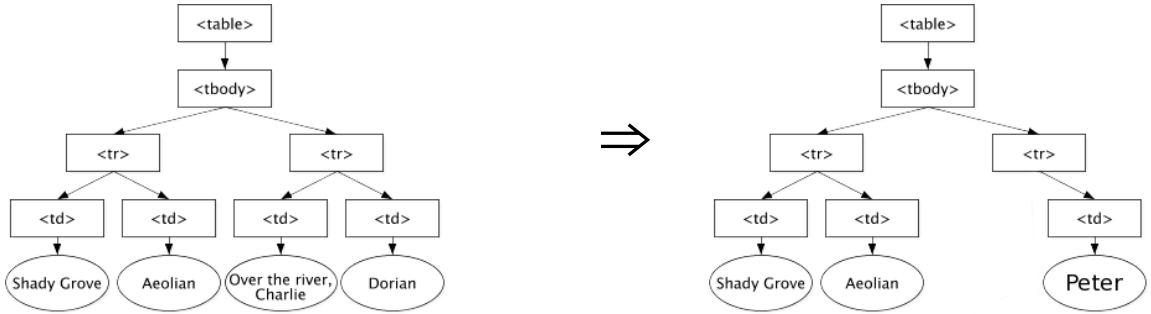


Arnaud Le Hors, et al, editors: Document Object Model (DOM) Level 3 Core Specification, Version 1.0, W3C Recommendation 07 April 2004. World Wide Web Consortium, 2004. https://www.w3.org/TR/DOM-Level-3-Core/

DOM__ acceso a elementos HTML

Métodos antiguos con limitaciones

```
// accede a elemento TBODY de la tabla
const myTbodyElement = myTableElement.firstChild;

// accede al segundo elemento TR; la lista de hijos empieza en 0
const mySecondTrElement = myTbodyElement.childNodes[1];

// elimina el primer elemento TD
mySecondTrElement.removeChild(mySecondTrElement.firstChild);

// Cambia el contenido del texto del elemento TD restante
mySecondTrElement.firstChild.firstChild.data = "Peter";
```



DOM__ acceso a elementos HTML

La navegación por DOM implica saber los nombres y utilizar `paths` en su estructura de árbol , que puede cambiar → **¡Problemático!**

~ Si esa estructura de árbol cambia, los `paths` ya no funcionan.

Ejemplo: insertamos un DIV en el formulario anterior.

```
<form name="form1" action ="">
<div class="field" name="fdiv">
  <label>Temperatura en Farenheit:</label>
    <input type ="text" name="fahrenheit" size=10 value="0" /><br/>
  <label>Temperatura en Celsius:</label>
    <input type="text" name="celsius" size="10" value ="" />
</div>
</form>
```

La implicación es que `document.form1.celsius` ya no funciona, ya que ahora hay un elemento `DIV` entre formulario y campo de texto. Ahora, habría que utilizar `document.form1.fdiv.celsius`

DOM__ acceso a elementos HTML

Hay un problema con `firstChild`, `parentNode` y `childNodes`, ya que devuelven **cualquier nodo**, lo que incluye:

- nodos de texto (espacios, saltos de línea)
- comentarios
- elementos reales → uso de métodos modernos: `firstElementChild`, `lastElementChild`, `parentElement`, `children` (solo incluyen **elementos HTML**, no nodos de texto o similares)

Esto provoca inconsistencias: si el HTML cambia de formato, aparecen nodos de texto adicionales que desplazan los índices de `childNodes`.

```
<ul id="lista">
  <li>Manzana</li>
  <li>Banana</li>
</ul>
```

HTML

```
const ul = document.getElementById("lista");

console.log(ul.firstChild);
```

JavaScript

Text {}

resultado...

DOM__ acceso a elementos HTML

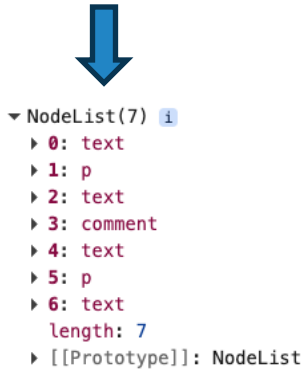
```html
<div id="contenedor">
  <p>Hola</p>
  <!-- Comentario -->
  <p>Mundo</p>
</div>
```

HTML

Selector

```
const div = document.getElementById("contenedor");

console.log(div.childNodes);
```

JavaScript



Consola

Cada salto de línea o espacio entre elementos cuenta como un **nodo de texto**, y además el comentario también aparece como nodo.

Esto provoca inconsistencias si se hace algo como:

```
div.childNodes[1].textContent = "Adiós";
```

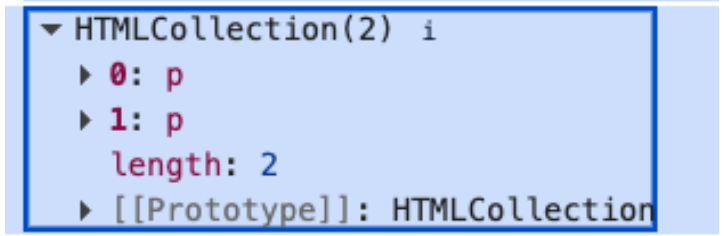
A veces funcionará, pero, con solo cambiar el formato del HTML (p.ej., quitar o agregar un salto de línea), el índice [1] dejaría de apuntar al mismo `<p>`.

DOM__ acceso a elementos HTML

Selector

Solución moderna:

```
const div = document.getElementById("contenedor");

console.log(div.children);

div.children[0].textContent = "Adiós"; // no depende de nodos de texto
```

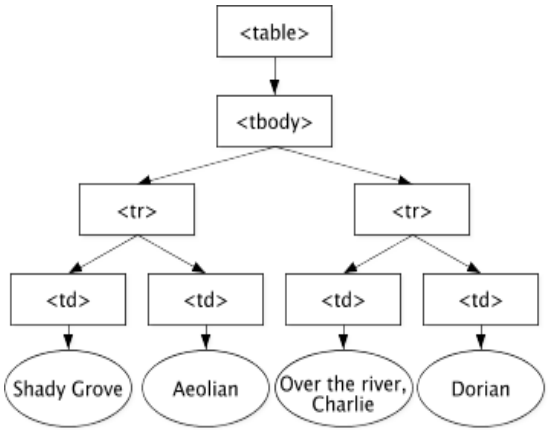


Consola

DOM__ acceso a elementos HTML

Solución anterior (propensa a errores):

```
var tbody = table.firstChild;
var secondRow = tbody.childNodes[1];
```



Selector

Solución moderna:

```
const tbody = table.querySelector("tbody");
const secondRow = tbody.children[1];
```

- `children` **solo incluye elementos**, no nodos de texto.
- `querySelector` **no depende de la estructura exacta** del árbol.

DOM__ acceso a elementos HTML

Solución anterior (propensa a errores):

```
const form = document.forms[0];
const input = form.childNodes[3]; // asume estructura fija
```

Selector

Solución moderna:

```
const input = document.querySelector("#email");
```

Cuando un fragmento **HTML cambia** (p.ej., añadiendo espacios, comentarios o nuevos DIV), la estructura del **DOM también cambia**.

Esto rompe cualquier código que dependa de índices numéricos (`childNodes[3]`).

Siempre se debe acceder a los elementos por **ID**, **clase o selectores**, no por posición.

DOM__ acceso a elementos HTML

Hoy se utilizan **selectores CSS** para la selección de elementos del árbol DOM

En un proyecto real, el HTML cambia continuamente: nuevas capas, DIVs, comentarios…

Por eso debemos **evitar depender de rutas**

El uso de selectores permite:

- Sintaxis más potente
- Independiente de la estructura del DOM
- Igual que los selectores CSS → fácil de recordar

```html
<input id="celsius" type="text">
```
HTML

```javascript
const celsiusInput = document.getElementById("celsius");
celsiusInput.value = 25;
```
JavaScript

1. Usar IDs y selectores CSS para encontrar elementos (`getElementById`, `querySelector`).
2. Evitar navegar por la estructura (`firstChild`, `childNodes`, índices).
3. Usar `children` y `firstElementChild` si necesitas recorrer hijos reales.
4. Mantener el DOM desacoplado de la estructura interna del HTML.
5. Preferir `querySelector` para código más declarativo y robusto.

DOM__ manipulando elementos HTML

No solo es posible **consultar** elementos HTML, sino también es posible **cambiar y manipular** sus valores

```
<html>
 <head>
   <title>Manipulando elementos HTML</title>
   <style>
     td.FondoRojo { background: #f00; }
   </style>
   <script>
function cambiarFondo1(id) {
 var cell=document.getElementById(id);
 // No se recomienda acceder al atributo STYLE desde JS
 cell.style.background = "#00f";
 cell.style.color = "white";
 cell.innerHTML = "azul!";
}
function cambiarFondo2(cell) {
// Se asigna clase de CSS - recomendable
 cell.className = "FondoRojo";
 cell.innerHTML = "rojo!";
}
 </script>
</head>
<body>
 <table border="1"> <tr>
   <td id="elem0" onclick="cambiarFondo1('elem0');">blanco</td>
   <td id="elem1" onclick="cambiarFondo2(this);">blanco</td></tr>
 </table>
</body>
</html>
```

Resumen: métodos clásicos de acceso

Estos métodos fueron los primeros disponibles en JavaScript para acceder al DOM

Útiles, pero limitadas en flexibilidad:

- Devuelven **`HTMLCollection`** (no array real)
- No permiten seleccionar ni por relaciones ni combinaciones CSS
- Muy rápidos y ampliamente compatibles

```
document.getElementById("id");              // 1 elemento
document.getElementsByTagName("p");         // colección de <p>
document.getElementsByClassName("item");    // colección de clase “item”
document.forms[0].elements["email"];        // acceso por nombre
```

Resumen: métodos comunes de selección

Desde **DOM Level 3** (2004), JavaScript incorpora métodos compatibles con selectores CSS

Aceptan cualquier selector CSS (`#id`, `.clase`, `tag`, `[attr=value]`) o combinaciones de ellos

Devuelven:

- `querySelector` → un solo elemento (o `null`)
- `querySelectorAll` → `NodeList` (puede iterarse con `forEach`)
- Compatibles con todos los navegadores modernos

Ventajas:

- Sintaxis uniforme (igual que en CSS)
- Evita depender de rutas o jerarquías internas del HTML.
- Utilizados en aplicaciones dinámicas y *frameworks* actuales (React, Vue, etc.)

Resumen: métodos comunes de selección

```
// primer elemento que cumple
document.querySelector("div.menu a.active");
// todos los elementos que cumplen
document.querySelectorAll(".card img");
```

```
const items = document.querySelectorAll(".item");
items.forEach(el => el.style.color = "blue");
```

```
for (const el of document.querySelectorAll(".item")) {
  el.classList.add("active");
}
```

¿Cómo convertimos un `NodeList` a `Array`?

API de manipulación DOM

Desde ES6, el DOM incorpora métodos simples y seguros para modificar la estructura sin usar `innerHTML`

Ventajas:

- No destruye eventos existentes (a diferencia de `innerHTML`)
- Permite trabajar con **nodos reales**, no solo texto HTML

```
// Inserción
element.append(nodo1, nodo2);    // al final del elemento
element.prepend(nodo);           // al inicio
element.before(nodo);            // antes del elemento
element.after(nodo);             // después del elemento

// Eliminación y reemplazo
element.remove();                // elimina el nodo
element.replaceWith(nuevoNodo);  // sustituye el nodo
```

```
const lista = document.querySelector("ul");
const nuevo = document.createElement("li");
nuevo.textContent = "Nuevo elemento";
lista.append(nuevo); // añade al final
```

API de manipulación DOM

La propiedad `classList` facilita modificar clases CSS de cualquier elemento HTML

Ventajas:

- Evita modificar `element.className` directamente
- Usar `classList` para mantener compatibilidad y evitar sobrescribir clases CSS
- Facilita la integración JS ↔ CSS (animaciones, temas, efectos visuales)

```
element.classList.add("activo");       // añade clase
element.classList.remove("oculto");    // elimina clase
element.classList.toggle("resaltado"); // añade o quita según estado
element.classList.contains("error");   // comprueba existencia
```

```
const card = document.querySelector(".card");
card.addEventListener("click", () => {
  card.classList.toggle("seleccionada");
});
```

Modificación de contenido: `innerHTML` vs `textContent`

1. `innerHTML` interpreta HTML, por lo que puede ejecutar código malicioso (XSS)
2. `textContent` inserta texto plano, sin interpretar HTML, por lo que es seguro

Consideraciones:

- Usar `textContent` para contenido generado por el usuario
- Usar `innerHTML` solo cuando el HTML sea controlado y seguro por el programador
- Si se necesita insertar elementos dinámicos, preferir `append`() y `createElement`()

Validación de entradas de usuario

Es habitual utilizar JS para la validación de entradas de usuario desde un formulario antes de que este se procese

- Se debe comprobar que los campos obligatorios no se hayan dejado vacíos
- Se debe verificar que los campos solo contengan caracteres permitidos o cumplan con una cierta gramática
- Se debe verificar que los valores estén dentro de los límites/rangos permitidos

```
<form method="post" action="XController.jsp" onsubmit="return validar(this)">
<label> User name: <input type="text" name="user"></label>
<label> Email address : <input type ="text" name="email"></label>
<input type ="submit" name ="submit" />
</form>
<script>
function validar (form) {
  fallo = validarUsuario(form.user.value);
  fallo += validarEmail(form.email.value);
  if (fallo == "") return true;
  else {
    alert(fallo); return false;
  }
}
</script>
```

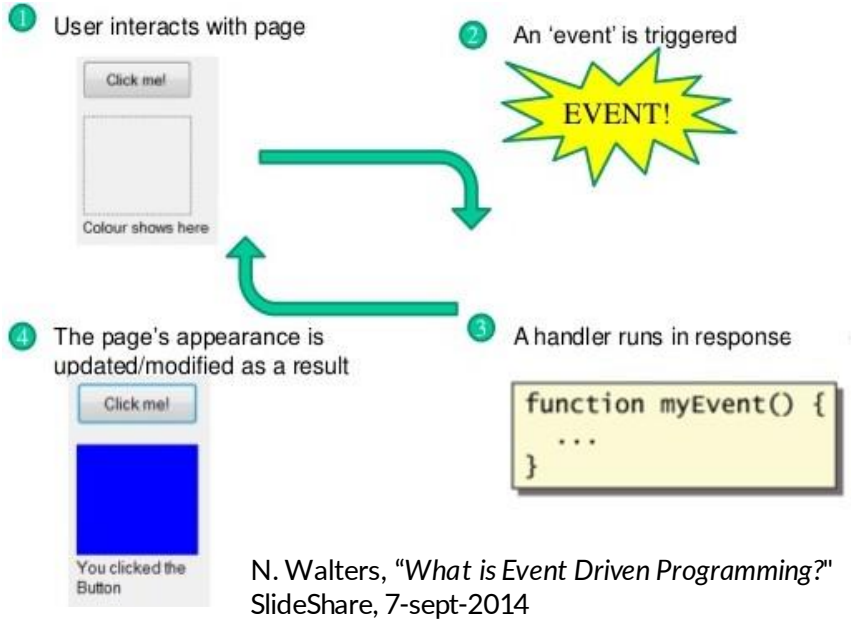
```
function validarUsuario (inp) {
  if (inp == "")
    return "No se introdujo nombre de usuario\n";
  else if (inp.length < 5)
    return "Nombre de usuario demasiado corto\n";
  else if (/[^a-zA-Z0-9_-]/.test(inp))
    return "Caracteres inválidos en nombre de usuario\n"
  else return "";
}

function validarEmail(inp) {
  if (inp == "") return "No se introdujo email\n";
  else if (!((inp.indexOf("@") > 0)) ||
       /[^a-zA-Z0-9\.\@\_\-]/.test(inp))
    return "Caracteres inválidos en email\n";
  else return "";
}
```

Se recomienda utilizar validación con HTML5 (`required`, `pattern`, `type=x`) y *Constraint Validation API*

S6-3.
Eventos

Programación dirigida por eventos en JS

Las aplicaciones web son dirigidas por eventos (event-driven), esto es, reaccionan a eventos del usuario como *clics*, pulsaciones, movimientos de ratón, etc.



N. Walters, "*What is Event Driven Programming?*" SlideShare, 7-sept-2014
https://tinyurl.com/ya58xbs9

- Podemos definir funciones manipuladoras de eventos (*event **handler** functions*) para una amplia variedad de eventos
- Estas funciones pueden manipular el objeto `document` (cambiando la página)

https://www.w3schools.com/jsref/obj_event.asp

Tipos de eventos__ Interfaz `UIEvent`

- `UIEvent` (hereda de `Event`) recoge eventos sencillos de la interfaz de usuario
- Cada evento tiene su *handler*:

error	**onerror**
load	**onload**
unload	**onunload**
resize	**onresize**
Scroll	**onscroll**

- De `UIEvent` heredan las interfaces:
  - **MouseEvent**, **KeyboardEvent**, **InputEvent**, **FocusEvent**, **TouchEvent**, **WheelEvent**

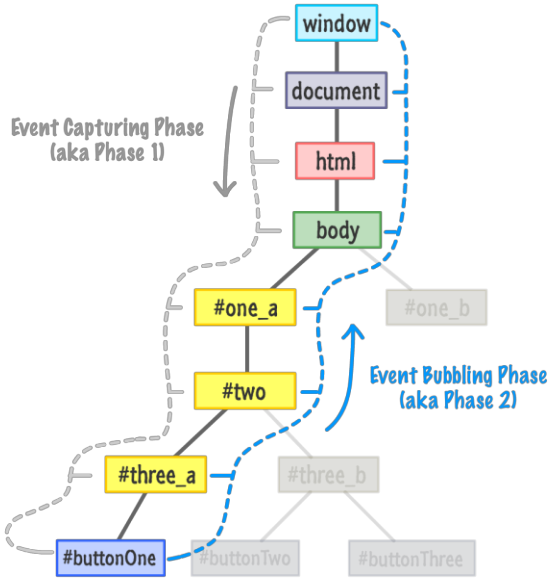
Tipos de eventos__ Interfaz `UIEvent` Ejemplo `onload`

- Se produce un evento `(on)load` cuando se carga un objeto
- Normalmente los *handlers* para eventos onload están asociados con el objeto `window` o el elemento `body` de un documento HTML


```
<! DOCTYPE html>
<html lang="es-ES">
  <head>
    <title>Ejemplo onload</title>
    <script>
      function hola() {alert("Bienvenido!"); }
    </script>
  </head>
  <body onload="hola()">
    <p>Contenido aqu&iacute;</p>
  </body>
</html>
```

Vinculando un evento a un elemento

```
<style>
  body * {
      margin: 10px;
      border: 1px solid blue;
  }
</style>
<form onclick="alert('form')">FORM
  <div onclick="alert('div')">DIV
    <p onclick="alert('p')">P</p>
  </div>
</form>
```



Fuente: kirupa.com

https://www.w3schools.com/jsref/obj_mouseevent.asp

Tipos de eventos__ Interfaz `MouseEvent`

- `MouseEvent` ocurre cuando el ratón del usuario interactúa con el documento
- Cuenta con unas propiedades propias del evento: `buttons`, `clientX`, `clientY`, `pageX`, `pageY`, etc.
- Además, maneja una serie de eventos:

onclick	**ondblclick**
oncontextmenu	**onmousedown**
onmouseenter	**onmouseleave**
onmouseover	**onmouseout**
onmouseup	

https://www.w3schools.com/jsref/obj_keyboardevent.asp

Tipos de eventos__ Interfaz `KeyboardEvent`

- `KeyboardEvent` ocurre cuando el usuario pulsa una tecla
- Cuenta con unas propiedades propias del evento: `charCode`, `ctrlKey`, `key`, `location`, `shiftKey`, etc.
- Además, maneja una serie de eventos:

onkeydown	**onkeyup**
onkeypress	

https://www.w3schools.com/jsref/obj_inputevent.asp

Tipos de eventos__ Interfaz `InputEvent`

- `InputEvent` ocurre cuando el usuario cambia el contenido de algún elemento de entrada de un formulario
- Cuenta con unas propiedades propias del evento: `inputType`, `data`, etc.
- Además, maneja el evento `oninput`

Tipos de eventos__ Interfaz `FocusEvent`

- Un evento de foco (`onFocus`) ocurre cuando un campo de formulario recibe foco de entrada presionando con el teclado o haciendo clic con el mouse
- Un evento de desenfoque (`onBlur`) cuando un campo de formulario pierde el foco por clic o presión de teclado por parte del usuario en otro elemento
- Un evento de cambio (`onChange`) ocurre cuando un campo de selección, texto o área de texto pierde el foco y su valor ha sido modificado

```
<form name="form1 "method="post" action="Xcontroller.jsp">
   <select name="select" required onChange="document.form1.submit();">
          <option value=""> Indique su juego favorito: </option>
          <option value="200812345"> The Last of Us </option>
          <option value="200867890"> Red Dead Redemption </option>
   </select>
</form>
```

```
<html>
 <head>
  <title>Ejemplo onchange</title>
  <script type="text/javascript">
function FahrenheitToCelsius(tFahr) {
 return (5/9)*(tFahr - 32);
}
  </script>
 </head>
<body>
 <form>
 Indique la temperatura en Fahrenheit:
  <input type="text" id="fahr" size="10" value="0“
     onchange="document.getElementById("celsius").value ="
FahrenheitToCelsius(parseFloat(document.getElementById('fahr').value)).toFixed(1);"/> <br/>
 La temperatura en Celsius:
  <input type="text" id="celsius“ size="10" value="" onfocus="blur();" />
</form>
</body>
</html>
```

blur() hace perder el foco al elemento; de este modo, #celsius nunca estará focalizado

https://www.w3schools.com/jsref/dom_obj_event.asp

Otros tipos de eventos de `Event`

- Interfaz **DragEvent:** hereda de **MouseEvent**
  - **`ondrag, ondragend, ondragenter, ondragleave, ondragover, ondragstart, ondrop`**
- **Form Events:** ocurre cuando un usuario interactúa con el elemento de formulario.
  - **`input, change, submit, reset, cut, copy, paste, select`**

https://developer.mozilla.org/en-US/docs/Web/Guide/Events/Mutation_events

https://www.w3.org/TR/DOM-Level-3-Events/#events-mutationevents https://dom.spec.whatwg.org/

Manejadores de eventos y elementos HTML

- Los eventos HTML indican que algo le ha ocurrido a algún elemento(s) HTML
- Los manejadores (*event handlers*) son funciones de JS que procesan eventos
- Los *handlers* **deben estar asociados** con elementos HTML para eventos específicos
- Esto se puede hacer a través de los atributos de los elementos HTML

```
<input type="button" value="Ayuda" onclick ="ayuda()">
```

- Preferiblemente, se debe usar una función de JS para agregar un controlador a un elemento HTML

```
// La mayoría de navegadores
window.addEventListener("load", ayuda)
```

Manejadores de eventos y elementos HTML


```
<head> <title>DOM Event Example</title>
  <style>
    #t { border: 1px solid red }
    #t1 { background-color: pink; }
  </style>
  <script>
  // Function to change the content of t2
  function modifyText(new_text) {
    var t2 = document.getElementById("t2");
    t2.firstChild.nodeValue = new_text; }

  // Function to add event listener to t
  function load() {
    var el = document.getElementById("t");
    el.addEventListener("click", function(){modifyText("four")}, false);
}
  </script>
</head>
<body onload="load();">
  <table id="t">
    <tr><td id="t1">one</td></tr>
    <tr><td id="t2">two</td></tr>
  </table>
</body>
```

Fuente: Mozilla.org

Manejadores de eventos y elementos HTML

- Se puede cancelar el comportamiento por defecto de un evento:

```
element.addEventListener("evento", function(event) {
  event.preventDefault();
});
```

- Los *handlers* pueden eliminarse:

```
window.removeEventListener("load", fn_evento);
```

- Se puede detener la propagación del evento:

```
element.addEventListener("click", (e) => {
  e.preventDefault();
  e.stopPropagation();
});
```

Eventos habituales

"click"	Clic simple con el botón principal del ratón.
"dblclick"	Doble clic del ratón.
"mousedown"	Pulsar un botón del ratón (sin soltarlo).
"mouseup"	Soltar un botón del ratón.
"mousemove"	Mover el puntero del ratón sobre un elemento.
"mouseenter"	El puntero entra en el área del elemento (sin burbujeo).
"mouseleave"	El puntero sale del área del elemento (sin burbujeo).
"mouseover"	El puntero entra en el elemento o en sus hijos (burbujea).
"mouseout"	El puntero sale del elemento o de uno de sus hijos.
"contextmenu"	Clic derecho del ratón (muestra menú contextual).
"wheel"	Desplazamiento de la rueda del ratón o trackpad.
"keydown"	Presionar una tecla (mantener pulsada cuenta como repetición).
"keyup"	Soltar una tecla.
"keypress"	(Obsoleto) Detectaba teclas imprimibles — usar keydown/keyup.

"input"	Cambio de valor en un campo de formulario mientras el usuario escribe (en tiempo real).
"change"	Cambio de valor confirmado (tras perder el foco o presionar Enter).
"submit"	Envío de un formulario.
"reset"	Reinicio de los campos de un formulario.
"focus"	Cuando un elemento recibe el foco (no burbujea).
"blur"	Cuando un elemento pierde el foco (no burbujea).
"focusin"	Similar a "focus", pero **sí burbujea** (útil para delegación).
"focusout"	Similar a "blur", pero **sí burbujea**.
"select"	Cuando se selecciona texto dentro de un <input> o <textarea>.
"load"	Cuando un recurso o toda la página se ha cargado completamente.
"DOMContentLoaded"	Cuando el DOM está listo (antes de que se carguen imágenes).
"beforeunload"	Antes de que el usuario abandone la página (permite mostrar aviso).
"unload"	Cuando la página se descarga (obsoleto en algunos contextos).
"resize"	Cambio de tamaño de la ventana o iframe.
"scroll"	Desplazamiento de un elemento o de la ventana.

"error"	Error al cargar un recurso (imagen, script, etc.).
"abort"	Cancelación de la carga de un recurso.
"animationstart"	Inicio de una animación CSS.
"animationend"	Fin de una animación CSS.
"animationiteration"	Una animación CSS repite su ciclo.
"transitionstart"	Comienza una transición CSS.
"transitionend"	Termina una transición CSS.
"dragstart"	Comienzo de una operación de arrastrar (Drag & Drop).
"drag"	Mientras se arrastra un elemento.
"dragenter"	El elemento arrastrado entra en una zona de destino.
"dragover"	El elemento arrastrado se mantiene sobre la zona de destino.
"dragleave"	El elemento arrastrado sale de la zona de destino.
"drop"	El elemento arrastrado se suelta en la zona de destino.
"dragend"	Fin del arrastre (suelta o cancela).

"touchstart"	Cuando un dedo toca la pantalla (dispositivos táctiles).
"touchmove"	Cuando un dedo se mueve sobre la pantalla.
"touchend"	Cuando se levanta el dedo de la pantalla.
"touchcancel"	Interrupción del toque (p. ej. alerta o multitarea).
"pointerdown"	Inicio de interacción con puntero (ratón, stylus o toque).
"pointermove"	Movimiento del puntero (unifica ratón + táctil).
"pointerup"	Fin de la interacción del puntero.
"pointerenter"	Puntero entra en un elemento (sin burbujeo).
"pointerleave"	Puntero sale del elemento (sin burbujeo).
"visibilitychange"	Cuando el documento pasa a estar visible u oculto (cambio de pestaña).
"fullscreenchange"	Cambio de modo pantalla completa.
"online"	El navegador recupera conexión.
"offline"	El navegador pierde conexión.

Recursos y lecturas

- Javascript Reference – *API Completa* (MDN)

  https://developer.mozilla.org/es/docs/JavaScript

  https://developer.mozilla.org/es/docs/Web/API

- DOM (MDN)

  https://developer.mozilla.org/es/docs/Web/API/Document_Object_Model

  https://dom.spec.whatwg.org/

- ECMAScript Language Specification

  https://tc39.es/ecma262/

- Tutoriales de JS

  https://javascript.info/

- Validación de formularios (MDN)

  https://developer.mozilla.org/es/docs/Learn/Forms/Form_validation

Recursos y lecturas

- Eventos (MDN)

  https://developer.mozilla.org/es/docs/Learn/JavaScript/Building_blocks/Events

- Seguridad y XSS (MDN)

  https://developer.mozilla.org/es/docs/Glossary/Cross-site_scripting

- Playground (MDN) – Editor en MDN

  https://developer.mozilla.org/es/play