


TECNOLOGÍAS WEB FRONTEND

```
2 const fetch = require('...')
3 const log = require('...')
4 let embed;
5
6
7 function transform($, ...args) {
8   // Promise.resolve to ...
9   return transform($, ...args);
10 }
11
12 function removeInitial($, ...args) {
13   return prev.then(() => {
14     $(':header').remove();
15     const children = $(children);
16     if ($(children).length > 0) {
17       $(header).text('');
18       $(children).remove();
19     }
20     return header;
21   });
22   return Promise.resolve();
23 }
```



TABLA DE CONTENIDO

- 01 Javascript básico y avanzado
 - 02 Manipulación de DOM
 - 03 Introducción a ECMAScript
 - 04 Introducción a TypeScript
- 

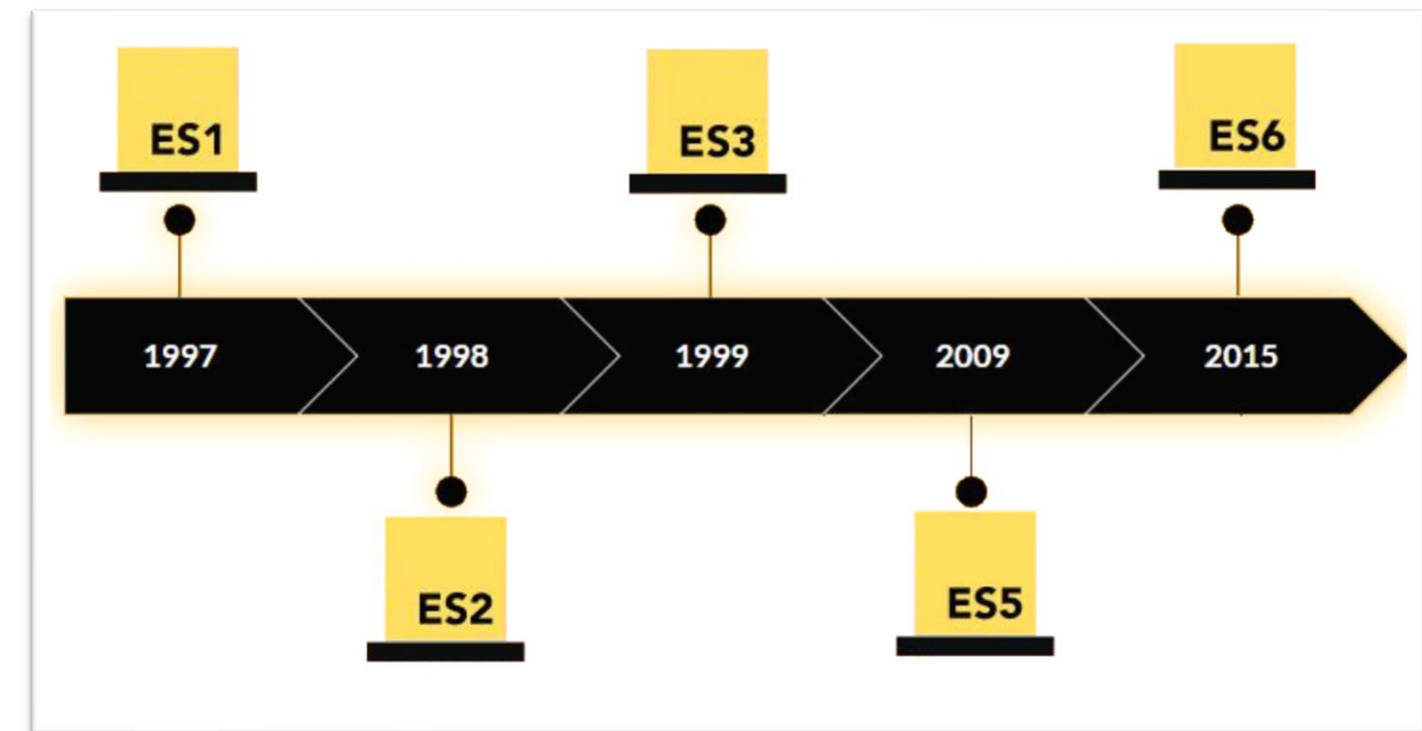
JavaScript

JavaScript es un lenguaje de programación que se utiliza principalmente para aportar dinamismo a sitios y aplicaciones web. Técnicamente, JavaScript es un **lenguaje de programación interpretado** por lo que el código escrito con JavaScript se puede probar directamente en cualquier navegador sin necesidad de procesos intermedios.

JavaScript funciona en complemento con los lenguajes web HTML Y CSS3, y permite modificar elementos dinámicamente..

Evolución de JavaScript

Creado por Brendan Eich en 1995, la primera versión de JavaScript ES1 se lanzó en 1997 y el lenguaje fue cambiando con el tiempo. Acá nos focalizamos en las versiones ES5 y ES6. A partir del año 2015, con la ES6 surgen las nuevas funcionalidades para manejo de promesas, y van surgiendo mejoras anuales de forma más rápida.



JavaScript

Sintaxis y código

JavaScript tiene sus propias reglas para la sintaxis, aunque respeta los estándares de muchos lenguajes de programación lógicos. Existen dos maneras de escribir código en JavaScript.

JavaScript

¿CÓMO ESCRIBIR CÓDIGO JS?

- Dentro de un archivo html, entre medio de las etiquetas `<html>`

```
<script>  
    // Aquí se escribe el código JS  
</script>
```

- En un archivo individual con extensión .js . Ejemplo: mi-archivo.js

```
<script src="js/main.js"></script>
```

Recuerda no utilizar espacios ni mayúsculas en los nombres de archivo.

JavaScript

Sintaxis: Palabras reservadas

Las **palabras reservadas** son las que se utilizan para construir las sentencias de JavaScript y que por tanto no pueden ser utilizadas libremente.

Las palabras actualmente reservadas por JavaScript son:

break, case, catch, continue, default, let delete, do, else, finally, for, function, if, in, instanceof, new, return, switch, this, throw, try, typeof, var, void, while, with, etc.

JavaScript

SINTÁXIS BÁSICA

- JavaScript es case-sensitive (distingue entre mayúsculas y minúsculas).
- Las instrucciones terminan con ; (opcional en muchos casos).
- Los comentarios se escriben con // para una línea o /* ... */ para múltiples líneas.
- Se pueden declarar variables con var, let, o const.



```
// Comentario de una línea
```

```
/*  
    Comentario  
    de múltiples  
    líneas  
*/
```

```
var nombre = "Juan"; // Declaración de  
variable
```


TIPOS DE DATOS

Los tipos de datos primitivos incluyen:

- **string**: Es una cadena. Las strings se definen entre comillas simples, dobles o backticks (para plantillas literales).
- **number**: Representa números, tanto enteros como decimales.
- **boolean**: Un tipo de dato lógico que puede tener uno de dos valores: true o false.
- **undefined**: Indica que la variable ha sido declarada pero aún no tiene un valor asignado.
- **null**: Es un valor que se puede asignar a una variable para indicar que está vacía o que no tiene valor.
- **symbol**: Un tipo de dato único e inmutable, utilizado como identificadores únicos para propiedades de objetos.



VARIABLES

VAR

Las variables declaradas con var tienen un ámbito de función, lo que significa que son accesibles desde cualquier parte dentro de la función en la que se declaran. Si se declara una variable con var **fuera de cualquier función**, se convierte en una **variable global**.

Las variables var pueden ser reasignadas y redeclaradas dentro del mismo ámbito.



VARIABLES

LET

Éstas variables tiene un ámbito de bloque, lo que significa que solo es accesible dentro del bloque de código donde es declarada.

Las variables let pueden ser reasignadas, pero no redeclaradas dentro del mismo ámbito.

```
let nombre = "Ana"; // String
let edad = 25;      // Number
let esEstudiante = true; // Boolean
let indefinido;     // undefined
let vacio = null;   // null
```

VARIABLES

CONST

Éstas variables tienen un ámbito de bloque, por lo que la variable solo es accesible dentro del bloque en el que se declara.

Una vez que se asigna un valor a una variable `const`, no puede ser reasignada ni redeclarada. Sin embargo, si la variable `const` es un **objeto** o un **array**, sus propiedades o elementos pueden ser modificados, pero no se puede reasignar el objeto o array en sí.



```
const PI = 3.1415;
```

OPERADORES

ARITMÉTICOS

- **+** (Suma): Suma dos números o concatena cadenas de texto.
- **-** (Resta): Resta el segundo número del primero.
- ***** (Multiplicación): Multiplica dos números.
- **/** (División): Divide el primer número por el segundo.
- **%** (Módulo o Residuo): Devuelve el resto de la división entre dos números.

```
let a = 10;
let b = 2;

let suma = a + b;
let resta = a - b;
let multiplicacion = a * b;
let division = a / b;
let modulo = a % b;
```

OPERADORES

COMPARACIÓN

- **== (Igualdad)**: Compara dos valores para verificar si son iguales en valor, sin considerar el tipo.
- **=== (Igualdad estricta)**: Compara dos valores para verificar si son iguales en valor y tipo.
- **!= (Desigualdad)**: Compara dos valores para verificar si son diferentes en valor, sin considerar el tipo.
- **!== (Desigualdad estricta)**: Compara dos valores para verificar si son diferentes en valor o tipo.

```
let x = 1;
let y = "1";

console.log(x == y);    // true
                        (igualdad de valor)
console.log(x === y);   // false
                        (igualdad estricta)
console.log(x != y);    // false
                        (desigualdad de valor)
console.log(x !== y);   // true
                        (desigualdad estricta)
```


OPERADORES

COMPARACIÓN

- **> (Mayor que)**: Verifica si el primer valor es mayor que el segundo.
- **< (Menor que)**: Verifica si el primer valor es menor que el segundo.
- **>= (Mayor o igual que)**: Verifica si el primer valor es mayor o igual que el segundo.
- **<= (Menor o igual que)**: Verifica si el primer valor es menor o igual que el segundo.



```
let x = 1;  
let y = "1";
```

```
console.log(x > 3);      // true  
console.log(x < 10);     // true  
console.log(x >= 5);     // true  
console.log(x <= 4);     // false
```

OPERADORES

LÓGICOS

- **&& (AND lógico)**: Devuelve true si ambas expresiones son verdaderas.
- **|| (OR lógico)**: Devuelve true si al menos una de las expresiones es verdadera.
- **! (NOT lógico)**: Invierte el valor lógico de la expresión.

OPERADORES

ASIGNACIÓN

- **= (Asignación)**: Asigna el valor del lado derecho a la variable del lado izquierdo.
- **+= (Asignación de suma)**: Suma el valor del lado derecho al valor de la variable y asigna el resultado a la variable.
- **-= (Asignación de resta)**: Resta el valor del lado derecho del valor de la variable y asigna el resultado a la variable.

```
let x = 10;
```

```
x += 5;
```

```
x -= 3;
```

```
x *= 2;
```

```
x /= 4;
```

```
x %= 5;
```

OPERADORES

ASIGNACIÓN

- ***= (Asignación de multiplicación):**
Multiplica el valor del lado derecho por el valor de la variable y asigna el resultado a la variable.
- **/= (Asignación de división):** Divide el valor de la variable por el valor del lado derecho y asigna el resultado a la variable.
- **%= (Asignación de módulo):** Calcula el resto de la división del valor de la variable entre el valor del lado derecho y asigna el resultado a la variable.

```
let x = 10;
```

```
x += 5;
```

```
x -= 3;
```

```
x *= 2;
```

```
x /= 4;
```

```
x %= 5;
```

ESTRUCTURAS DE CONTROL

CONDICIONALES

if, else if, else: Se utilizan para ejecutar código basado en condiciones específicas.

- **if** evalúa una condición; si es verdadera, ejecuta un bloque de código.
- **else if** permite evaluar condiciones adicionales si la anterior no es verdadera.
- **else** se ejecuta cuando ninguna de las condiciones anteriores es verdadera.

```
let age = 20;

if (age < 18) {
  console.log("Menor de edad");
} else if (age >= 18 && age < 65) {
  console.log("Adulto");
} else {
  console.log("Tercera edad");
}
```

ESTRUCTURAS DE CONTROL

CONDICIONALES

switch:

- Compara el valor de una variable con múltiples casos.
- Se utiliza para ejecutar diferentes bloques de código en función del valor de una variable.

```
let color = "rojo";

switch (color) {
  case "rojo":
    console.log("El color es rojo");
    break;
  case "azul":
    console.log("El color es azul");
    break;
  case "verde":
    console.log("El color es verde");
    break;
  default:
    console.log("Color desconocido");
}
```


ESTRUCTURAS DE CONTROL

BUCLES

for:

- Ejecuta un bloque de código un número específico de veces.
- Generalmente se utiliza cuando se conoce el número de iteraciones.

```
for (let i = 0; i < 5; i++) {  
  console.log("Iteración " + i);  
}
```

ESTRUCTURAS DE CONTROL

BUCLES

while:

- Ejecuta un bloque de código mientras una condición sea verdadera.
- Se utiliza cuando no se conoce de antemano cuántas veces se repetirá el ciclo.

```
let count = 0;

while (count < 5) {
  console.log("Cuenta: " + count);
  count++;
}
```

ESTRUCTURAS DE CONTROL

BUCLES

do...while:

- Similar al while, pero garantiza que el bloque de código se ejecute al menos una vez antes de verificar la condición.

```
let number = 0;

do {
  console.log("Número: " + number);
  number++;
} while (number < 5);
```

ESTRUCTURAS DE CONTROL

BUCLES

break y continue:

- **break:** Interrumpe la ejecución del bucle de forma inmediata y sale de él.
- **continue:** Salta la iteración actual y continúa con la siguiente iteración del bucle.

```
// Uso de break
for (let i = 0; i < 10; i++) {
  if (i === 5) {
    break; // Sale del bucle cuando i es 5
  }
  console.log(i);
}

// Uso de continue
for (let i = 0; i < 10; i++) {
  if (i % 2 === 0) {
    continue; // Salta la iteración cuando i es par
  }
  console.log(i);
}
```

FUNCIONES

Las funciones permiten agrupar bloques de código que se pueden reutilizar en diferentes partes del programa. Éstas se declaran con la palabra clave **function**.

Pueden recibir parámetros, que son valores que se pasan a la función para personalizar su comportamiento. Además pueden devolver un valor usando la palabra clave **return**. El valor devuelto puede ser utilizado donde la función es llamada.

```
function saludar(nombre) {  
    return "Hola, " + nombre + "!";  
}  
  
console.log(saludar("Juan"));
```

FUNCIONES

Las funciones pueden ser nombradas, lo que significa que se les da un nombre específico. Las funciones anónimas no tienen un nombre y se suelen utilizar como funciones de callback o en expresiones.

```
function mostrarHora() {  
    let horaActual = new  
Date().toLocaleTimeString();  
    console.log("La hora actual es: " +  
horaActual);  
}  
  
let suma = function (a, b) {  
    return a + b;  
};  
  
mostrarHora();  
console.log(suma(5, 3));  
  
function procesar(callback) {  
    let resultado = callback(10);  
    console.log("Resultado: " + resultado);  
}  
  
procesar(function (x) {  
    return x * 2;  
});
```


FUNCIONES

Las funciones pueden ser nombradas, lo que significa que se les da un nombre específico. Las funciones anónimas no tienen un nombre y se suelen utilizar como funciones de callback o en expresiones.

```
function mostrarHora() {  
    let horaActual = new  
Date().toLocaleTimeString();  
    console.log("La hora actual es: " +  
horaActual);  
}  
  
let suma = function (a, b) {  
    return a + b;  
};  
  
mostrarHora();  
console.log(suma(5, 3));  
  
function procesar(callback) {  
    let resultado = callback(10);  
    console.log("Resultado: " + resultado);  
}  
  
procesar(function (x) {  
    return x * 2;  
});
```

ARREGLOS EN JAVASCRIPT

Un arreglo es una colección de elementos, que pueden ser de cualquier tipo de dato.

Los arreglos se pueden declarar utilizando corchetes `[]` o el constructor `Array`.

Los elementos de un arreglo están ordenados y se acceden a través de índices numéricos, comenzando desde 0.

```
// Declarar un arreglo de números
let numeros = [1, 2, 3, 4, 5];

// Acceder al primer elemento
console.log(numeros[0]); // Resultado: 1

// Declarar un arreglo usando el constructor Array
let frutas = new Array("Manzana", "Banana", "Cereza");

// Acceder al último elemento
console.log(frutas[frutas.length - 1]); // Resultado: "Cereza"
```

MÉTODOS COMUNES DE ARREGLOS

Agregar y Eliminar Elementos:

- `push()`: Agrega un elemento al final del arreglo.
- `pop()`: Elimina el último elemento del arreglo.
- `unshift()`: Agrega un elemento al inicio del arreglo.
- `shift()`: Elimina el primer elemento del arreglo.

```
// Agregar elementos al final del arreglo
numeros.push(6);
console.log(numeros); // Resultado: [1, 2, 3, 4, 5, 6]

// Eliminar el último elemento
numeros.pop();
console.log(numeros); // Resultado: [1, 2, 3, 4, 5]

// Agregar elementos al inicio del arreglo
frutas.unshift("Naranja");
console.log(frutas); // Resultado: ["Naranja", "Manzana", "Banana", "Cereza"]

// Eliminar el primer elemento
frutas.shift();
console.log(frutas); // Resultado: ["Manzana", "Banana", "Cereza"]
```

MÉTODOS COMUNES DE ARREGLOS

Otros métodos:

- **length**: Devuelve el número de elementos en un arreglo.
- **splice()**: Permite agregar o eliminar elementos en una posición específica.
- **slice()**: Devuelve una copia de una parte del arreglo.
- **indexOf()**: Encuentra el índice de un elemento en el arreglo.
- **forEach()**: Ejecuta una función para cada elemento del arreglo.

```
// Usar splice para eliminar 2 elementos a partir del índice 1
frutas.splice(1, 2);
console.log(frutas); // Resultado: ["Manzana"]

// Usar slice para copiar una parte del arreglo
let otrasFrutas = frutas.slice(0, 2);
console.log(otrasFrutas); // Resultado: ["Manzana", "Banana"]

// Encontrar el índice de un elemento
let indice = frutas.indexOf("Banana");
console.log(indice); // Resultado: 1

// Recorrer un arreglo con forEach
numeros.forEach(function(numero) {
    console.log(numero * 2); // Multiplica cada número por 2 y lo
    imprime
});
```


RECORRER ARREGLOS

- `map()`: Crea un nuevo arreglo aplicando una función a cada elemento del arreglo original.
- `filter()`: Crea un nuevo arreglo con todos los elementos que cumplan una condición específica.
- `reduce()`: Aplica una función a un acumulador y cada valor del arreglo para reducirlo a un solo valor.

```
// Usar map para crear un nuevo arreglo con los números al cuadrado
let cuadrados = numeros.map(function(numero) {
  return numero * numero;
});
console.log(cuadrados); // Resultado: [1, 4, 9, 16, 25]

// Usar filter para obtener números mayores a 3
let mayores = numeros.filter(function(numero) {
  return numero > 3;
});
console.log(mayores); // Resultado: [4, 5]

// Usar reduce para sumar todos los números en el arreglo
let suma = numeros.reduce(function(acumulador, numero) {
  return acumulador + numero;
}, 0);
console.log(suma); // Resultado: 15
```



OBJETOS EN JAVASCRIPT

Un objeto es una colección de propiedades, donde cada propiedad es una asociación entre una **clave** y un **valor**. Los valores de las propiedades pueden ser de cualquier tipo, incluyendo otros objetos, arreglos o funciones.

Los objetos se pueden crear utilizando la sintaxis de llaves `{}` o el constructor `Object`.



OBJETOS EN JAVASCRIPT

Propiedades y Métodos:

- **Acceso a Propiedades:** Las propiedades de un objeto se pueden acceder usando notación de punto (.) o notación de corchetes ([]).
- **Añadir y Modificar Propiedades:** Se pueden añadir nuevas propiedades o modificar las existentes simplemente asignando un valor a una clave.
- **Métodos de Objetos:** Los métodos son funciones que están asociadas a un objeto como una propiedad. Se invocan usando notación de punto.

OBJETOS EN JAVASCRIPT



```
// Crear un objeto con propiedades
```

```
let persona = {  
  nombre: "Juan",  
  edad: 30,  
  profesion: "Desarrollador"  
};
```

```
// Acceder a las propiedades usando notación de punto
```

```
console.log(persona.nombre); // Resultado: "Juan"
```

```
// Acceder a las propiedades usando notación de corchetes
```

```
console.log(persona["edad"]); // Resultado: 30
```

OBJETOS EN JAVASCRIPT



```
// Añadir una nueva propiedad al objeto
persona.nacionalidad = "Mexicana";
console.log(persona.nacionalidad); // Resultado: "Mexicana"

// Modificar una propiedad existente
persona.edad = 31;
console.log(persona.edad); // Resultado: 31
```

OBJETOS EN JAVASCRIPT



```
// Añadir un método al objeto
persona.saludar = function() {
    console.log("Hola, mi nombre es " + this.nombre);
};

// Invocar el método del objeto
persona.saludar(); // Resultado: "Hola, mi nombre es Juan"
```

OBJETOS EN JAVASCRIPT

Iterar Sobre un Objeto

Se pueden recorrer las propiedades de un objeto utilizando el bucle `for...in`.



```
// Usar for...in para recorrer las propiedades del
objeto
for (let clave in persona) {
    console.log(clave + ": " + persona[clave]);
}
```

OBJETOS EN JAVASCRIPT

Object.keys() y Object.values():

- **Object.keys()**: Devuelve un arreglo con las claves de las propiedades de un objeto.
- **Object.values()**: Devuelve un arreglo con los valores de las propiedades de un objeto.

```
// Obtener las claves del objeto
let claves = Object.keys(persona);
console.log(claves); // Resultado: ["nombre", "edad",
"profesion", "nacionalidad"]

// Obtener los valores del objeto
let valores = Object.values(persona);
console.log(valores); // Resultado: ["Juan", 31,
"Desarrollador", "Mexicana"]
```

EVENTOS EN JAVASCRIPT

Los eventos son acciones que ocurren en el navegador, como hacer clic, mover el mouse o enviar un formulario. JavaScript puede detectar y responder a estos eventos para realizar acciones en la página web.

Algunos de los eventos más comunes incluyen `click`, `mouseover`, `keydown`, y `submit`.

```
// Detectar y manejar un evento de clic
document.getElementById("miBoton").addEventListener("click",
function () {
    console.log("Botón presionado");
});
```

EVENTOS EN JAVASCRIPT



```
// Detectar el movimiento del mouse sobre un elemento
document
  .getElementById("miElemento")
  .addEventListener("mouseover", function () {
    console.log("Mouse sobre el elemento");
  });

// Detectar la pulsación de una tecla
document.addEventListener("keydown", function (event) {
  console.log("Tecla presionada: " + event.key);
});

// Detectar el envío de un formulario
document
  .getElementById("miFormulario")
  .addEventListener("submit", function (event) {
    event.preventDefault(); // Evita el envío real del formulario
    console.log("Formulario enviado");
  });
```


Manipulación del DOM

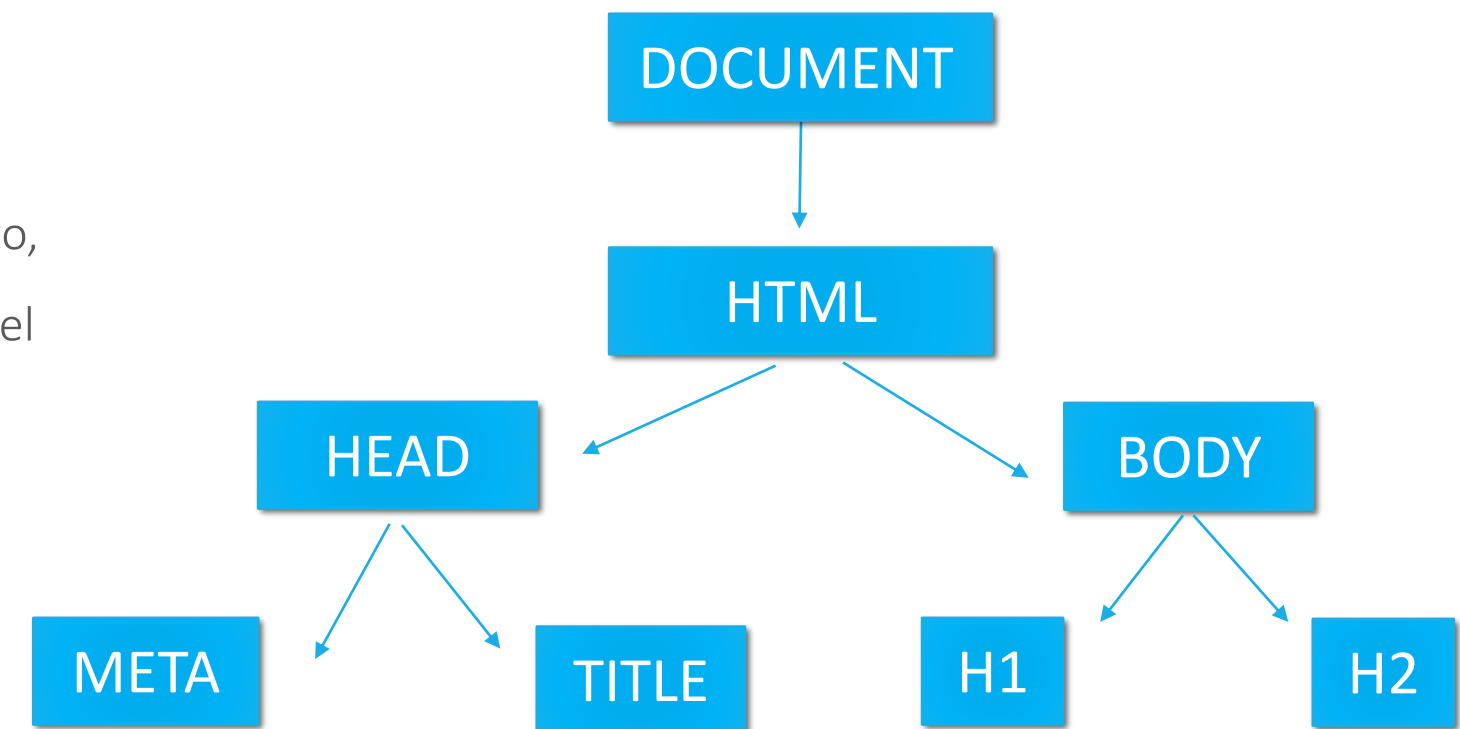
El **DOM** (Document Object Model, en español Modelo de Objetos del Documento) es la **estructura de objetos** que genera el navegador cuando se carga un documento y **se puede alterar mediante Javascript** para cambiar dinámicamente los contenidos y aspecto de la página

Estructura de árbol

El **DOM** representa un documento HTML como una estructura jerárquica de nodos, donde cada nodo corresponde a una parte del documento (por ejemplo, un elemento, atributo o texto). El DOM permite a JavaScript interactuar y manipular la estructura, el contenido y el estilo de la página web.

Cada elemento en la página web es un nodo en esta estructura, lo que permite a JavaScript interactuar y modificar el contenido, el estilo y la estructura de la página.

El nodo raíz es el documento entero (<html>), y a partir de él se derivan todos los demás nodos, como <head>, <body>, y otros elementos anidados.



DOM

NODOS

Un **nodo** es una unidad fundamental dentro del DOM. Cada nodo puede ser un elemento HTML, un atributo, un texto, un comentario, o incluso un fragmento del documento.

➤ **Elemento (Element Node)**: Representa cualquier elemento HTML, como `<div>`, `<p>`, `<a>`, etc.

➤ **Texto (Text Node)**: Representa el contenido textual de un elemento.

DOM

NODOS

- **Atributo (Attribute Node)**: Representa un atributo de un elemento, como id, class, href, etc.
- **Comentario (Comment Node)** : Representa un comentario dentro del HTML. Todo el texto dentro de `<!-- -->` es un nodo de comentario.
- **Document (Document Node)**: Representa el documento completo. Es la raiz del arbol del DOM.

DOM

Selectores

En JavaScript contamos con una serie de métodos denominados **selectores** que nos permiten seleccionar elementos de nuestro documento, y luego tratarlos como objetos: consultar y modificar sus propiedades, e invocar sus métodos.

- **getElementById()**, accede al primer elemento con un id específico.
- **getElementByName()**, accede a todos los elementos con un nombre (name) específico.
- **getElementsByTagName()**, accede a todos los elementos con un tagname (etiqueta HTML) específico.
- **querySelector()**, devuelve el primer elemento que coincide con un selector CSS especificado. Para obtener todos los elementos que coinciden (no solo el primero), se debe utilizar **querySelectorAll()**.

DOM

EJEMPLO: INTERACCIÓN CON EL DOM

JavaScript puede seleccionar elementos del DOM utilizando métodos como `getElementById`, `getElementsByClassName`, `getElementsByTagName`, y `querySelector`.

Después de seleccionar un elemento, se pueden realizar cambios en su contenido, estilo, atributos, e incluso crear o eliminar nodos.

```
// Seleccionar un elemento por su ID
let header = document.getElementById("header");

// Seleccionar elementos por su clase
let items = document.getElementsByClassName("item");

// Seleccionar elementos por su etiqueta
let paragraphs = document.getElementsByTagName("p");

// Seleccionar el primer elemento que coincida con un selector CSS
let main = document.querySelector(".main");

// Seleccionar todos los elementos que coincidan con un selector CSS
let links = document.querySelectorAll("a");
```

Manipulación del DOM

La manipulación es el objetivo principal del DOM. Es todo lo que sucede después de hacer referencia y seleccionar los elementos con los que se desea trabajar. Esto conduce a un cambio en el estado del elemento, de estático a dinámico.

- Mediante las propiedades `innerText` (o su equivalente `textContent`) e `innerHTML` podemos manipular el contenido de texto y el HTML dentro de un elemento del DOM.

```
var items = document.querySelectorAll("ul li.item");

items[1]; // → <li class="item">Item 2</li>
items[1].innerText; // → "Item 2"

items[1].innerText = "Segundo item";
items[1]; // → <li class="item">Segundo item</li>

items[2]; // → <li class="item">Item 3</li>
items[2].innerHTML; // → "Item 3"
items[2].innerHTML = "<strong>Tercero</strong>";
items[2].innerHTML; // → <strong>Tercero</strong>
items[2];
// → <li class="item"><strong>Tercero</strong></li>
```

DOM

EJEMPLO MODIFICACIÓN DEL CONTENIDO Y ESTILO

Se puede cambiar el texto dentro de un elemento usando `textContent` (`innerText`) o `innerHTML`.

El estilo de un elemento se puede modificar utilizando la propiedad `style`, permitiendo cambiar colores, tamaños, márgenes, etc.

```
// Cambiar el texto de un elemento
header.textContent = "Bienvenido a mi sitio web";

// Cambiar el contenido HTML de un elemento
main.innerHTML = "<h2>Subtítulo</h2><p>Este es un párrafo dentro de la sección principal.</p>";

// Modificar el estilo de un elemento
header.style.backgroundColor = "lightblue";
header.style.padding = "10px";
header.style.fontFamily = "Arial, sans-serif";
```


Manipulación del DOM

- Crear Nuevos Elementos: El método `createElement()` nos permite crear un nuevo elemento nodo. Recibe como argumento obligatorio el nombre del elemento a crear.

```
document.createElement(nombreDelElemento)
```

- Reemplazar un nodo: El método `replaceChild()` nos posibilita reemplazar un nodo por otro. Recibe dos argumentos , el nuevo nodo y el nodo a reemplazar.

```
nodo.replaceChild(nuevoNodo, viejoNodo)
```

DOM

Manipulación del DOM

- Eliminar elementos del DOM: `remove()` y `removeChild()` nos permiten eliminar elementos del DOM.
- El atributo `class` se utiliza frecuentemente para asociar estilos CSS a los elementos de nuestro HTML. Poder modificar este atributo de forma dinámica con JavaScript nos resultará de utilidad. La propiedad `className` nos da acceso a la clase o clases de un elemento nodo, en forma de string y separadas por espacios.

```
div.className = "seccion";  
div.className; // → "seccion"  
  
div.className += " contenido";  
div.className; // → "seccion contenido"  
  
div;  
// → <div class="seccion contenido">...</div>
```

DOM

EJEMPLO: CREAR Y ELIMINAR ELEMENTOS

Se pueden crear nuevos elementos HTML utilizando `createElement`. Los nuevos elementos se pueden agregar al DOM con `appendChild` o `insertBefore`.

Para eliminar elementos, se puede utilizar `removeChild` o `remove`.

```
// Crear un nuevo elemento de lista
let newItem = document.createElement("li");
newItem.textContent = "Elemento nuevo";

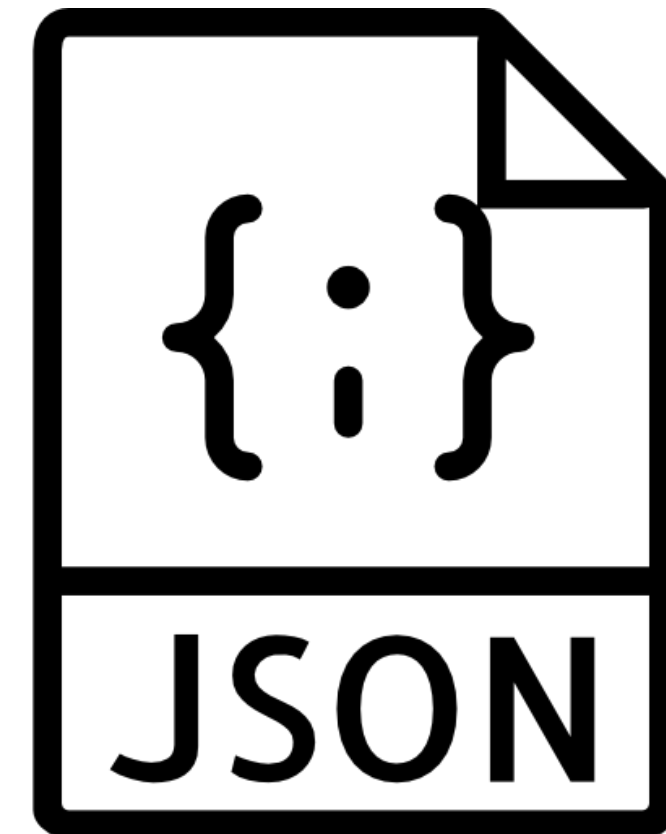
// Agregar el nuevo elemento a una lista existente
let list = document.querySelector("ul");
list.appendChild(newItem);

// Seleccionar un elemento y eliminarlo
let unwantedItem = document.querySelector(".unwanted-item");
unwantedItem.remove();

// Eliminar un nodo hijo de un elemento padre
list.removeChild(newItem); // Elimina el nuevo elemento añadido anteriormente
```

JSON

- JSON (JavaScript Object Notation) es un formato de texto ligero para el intercambio de datos.
- Los datos en JSON se organizan en pares clave-valor.
- Es similar a la sintaxis de los objetos en JavaScript, pero con algunas diferencias en las reglas de formato (por ejemplo, las claves deben estar entre comillas dobles).



JSON

Funciones JSON en JavaScript

- **JSON.stringify()**: Convierte un objeto de JavaScript en una cadena JSON.
- **JSON.parse()**: Convierte una cadena JSON en un objeto de JavaScript.

```
// Crear un objeto en JavaScript
let usuario = { nombre: "Ana", edad: 28, ciudad: "Madrid" };

// Convertir el objeto a una cadena JSON
let usuarioJSON = JSON.stringify(usuario);

console.log(usuarioJSON); // Resultado:
'{"nombre":"Ana","edad":28,"ciudad":"Madrid"}'

// Una cadena JSON
let datosJSON =
'{"producto":"Laptop","precio":1200,"disponible":true}';

// Convertir la cadena JSON a un objeto de JavaScript
let producto = JSON.parse(datosJSON);
console.log(producto);
// Resultado: {producto: "Laptop", precio: 1200, disponible:
true}
```



ECMAScript

ECMAScript es un estándar de scripting que forma la base de JavaScript. Proporcionar una base estándar para la implementación de lenguajes de scripting en navegadores y otros entornos.

La especificación es mantenida por el ECMA International y ha evolucionado con varias versiones



EVOLUCION VERSION ECMAScript

VERSION	NOVEDADES	VERSION	NOVEDADES
ECMAScript 2015 (ES6)	<ul style="list-style-type: none">• Clases: Introducción de la sintaxis de clases para la programación orientada a objetos.• Módulos: Soporte nativo para módulos, facilitando la organización del código.• Arrow Functions: Sintaxis más concisa para las funciones.• Promises: Para manejo de operaciones asíncronas.• Template Literals: Para cadenas de texto multilinea y interpolación.	ECMAScript 2020 (ES11)	<ul style="list-style-type: none">• Nullish Coalescing Operator (??): Para manejar valores nulos o indefinidos.• Optional Chaining (?.): Permite acceder a propiedades de objetos de manera segura.• BigInt: Tipo de dato para manejar enteros grandes
ECMAScript 2016 (ES7)	<ul style="list-style-type: none">• Array.prototype.includes: Método para verificar si un arreglo contiene un elemento.• Exponentiation Operator (**): Para realizar potencias de manera más sencilla.	ECMAScript 2021 (ES12)	<ul style="list-style-type: none">• Logical Assignment Operators: Nuevos operadores para asignación lógica.• String.prototype.replaceAll: Método para reemplazar todas las ocurrencias de una subcadena.• WeakRefs: Referencias débiles para la gestión de memoria.
ECMAScript 2017 (ES8)	<ul style="list-style-type: none">• Async/Await: Simplificación del manejo de promesas y código asíncrono• Object.entries() y Object.values(): Métodos para trabajar con objetos de manera más sencilla.• String.prototype.padStart y padEnd: Métodos para añadir caracteres a cadenas.	ECMAScript 2022 (ES13)	<ul style="list-style-type: none">• Class Fields: Permite definir campos directamente en clases.• Top-Level Await: Soporte para await en el nivel superior de módulos.• Private Methods y Accessors: Métodos y accesorios privados en clases.
ECMAScript 2018 (ES9)	<ul style="list-style-type: none">• Rest/Spread Properties: Mejora en la manipulación de objetos.• Asynchronous Iteration: Soporte para bucles asíncronos.• Promise.prototype.finally: Método para ejecutar código después de que una promesa se resuelve o se rechaza.	ECMAScript 2023 (ES14)	<ul style="list-style-type: none">• Array.prototype.toSorted, toSpliced, toReversed: Nuevos métodos para manipulación de arreglos sin mutar el original.• Hashbang Grammar: Soporte para scripts que comienzan con #! en la primera línea.• Change to this in Function.prototype.toString(): Mejor manejo de la función toString en el contexto de this.
ECMAScript 2019 (ES10)	<ul style="list-style-type: none">• Array.prototype.flat y flatMap: Métodos para aplanar arreglos.• Object.fromEntries(): Para convertir listas de pares clave-valor en objetos.• String.prototype.trimStart y trimEnd: Métodos para eliminar espacios en blanco al inicio y al final de cadenas.	ECMAScript 2024 (ES15)	<ul style="list-style-type: none">• Rest/Spread Properties• Asynchronous Iteration

FUNCIONES FLECHA (ARROW FUNCTIONS)

Las funciones flecha son una forma más concisa de escribir funciones en JavaScript. Utilizan una sintaxis más compacta y no tienen su propio this, lo que las hace útiles en ciertos contextos.

Características:

- No tienen su propio this, arguments, super o new.target.
- Son más cortas y no requieren la palabra clave function.

```
// Función tradicional
function sumar(a, b) {
  return a + b;
}

// Función flecha
const sumar = (a, b) => a + b;

console.log(sumar(5, 3)); // Resultado: 8

// Función flecha sin parámetros
const saludar = () => console.log('¡Hola Mundo!');

saludar(); // Resultado: ¡Hola Mundo!
```


SPREAD OPERATOR Y REST PARAMETERS

Spread Operator (...):

- Permite expandir elementos de un arreglo u objeto en otro arreglo u objeto.
- Útil para combinar arreglos o copiar propiedades de objetos.
- Útil para funciones que deben manejar una cantidad indefinida de argumentos.

```
// Combinar arreglos con spread
const numeros = [1, 2, 3];
const masNumeros = [0, ...numeros, 4];

console.log(masNumeros); // Resultado: [0, 1, 2, 3, 4]

// Copiar un objeto con spread
const persona = { nombre: 'Ana', edad: 30 };
const personaCopiada = { ...persona, ciudad: 'Madrid' };

console.log(personaCopiada);
// Resultado: { nombre: 'Ana', edad: 30, ciudad: 'Madrid' }
```

SPREAD OPERATOR Y REST PARAMETERS

Rest Parameters (...):

- Permite capturar un número variable de argumentos en una función.
- Útil para funciones que deben manejar una cantidad indefinida de argumentos.

```
function mostrarDatos(nombre, ...detalles) {  
  console.log(`Nombre: ${nombre}`);  
  console.log(`Detalles: ${detalles.join(', ')}`);  
}  
  
mostrarDatos('Luis', 'Ingeniero', 'Madrid', 'España');
```

PROMESAS

Es un objeto que representa la eventual finalización (o falla) de una operación asíncrona. Permite manejar operaciones asíncronas de manera eficiente y legible.

Estados de una Promesa:

- **Pending:** La promesa está en espera.
- **Fulfilled:** La promesa se ha completado con éxito.
- **Rejected:** La promesa ha fallado.

```
const promesa = new Promise((resolve, reject) => {
  let exito = true;

  // Simulamos una operación que toma tiempo con un for y un setTimeout
  console.log("Operación en proceso...");
  for (let i = 0; i < 1e9; i++) {} // Este bucle simula un proceso largo
  console.log("Fin del proceso...");

  setTimeout(() => {
    if (exito) {
      resolve("La operación fue exitosa");
    } else {
      reject("Hubo un error");
    }
  }, 3000); // Simulamos un tiempo de espera adicional
});

promesa
  .then((resultado) => {
    console.log(resultado); // Se ejecuta si la promesa es resuelta
  })
  .catch((error) => {
    console.log(error); // Se ejecuta si la promesa es rechazada
  })
  .finally(() => {
    console.log("Proceso terminado"); // Se ejecuta al final, sin importar el resultado
  });

console.log("El código sigue ejecutándose...");
```

FUNCIONES ASÍNCRONAS (ASYNC/AWAIT)

`async` y `await` simplifican el manejo de código asíncrono en JavaScript.

- **`async`:** Se utiliza para declarar que una función es asíncrona. Esto indica que la función puede contener operaciones que no se ejecutan de manera secuencial, como solicitudes HTTP. Las funciones asíncronas siempre devuelven una promesa.
- **`await`:** Se utiliza dentro de una función `async` para esperar a que una promesa sea resuelta. El uso de `await` hace que el código sea más legible y fácil de escribir que manejar promesas con `.then()` y `.catch()`.

```
async function obtenerDatos() {  
  try {  
    const response = await  
    fetch("https://jsonplaceholder.typicode.com/posts");  
    const data = await response.json();  
    return data;  
  } catch (error) {  
    return null;  
  }  
}
```

TYPESCRIPT

TypeScript es un lenguaje de programación basado en JavaScript que añade tipado estático opcional y otras características avanzadas.

Características Principales:

- **Tipado Estático:** Permite declarar tipos para variables, parámetros y retornos de funciones, lo que ayuda a detectar errores durante el desarrollo.
- **Interoperabilidad con JavaScript:** El código TypeScript se compila a JavaScript puro, por lo que puede integrarse fácilmente en proyectos existentes.
- **Soporte para ES6 y Más Allá:** Incluye características modernas de JavaScript y futuras versiones, como clases, módulos y async/await.



GRACIAS

```
2  const fetch = require('...')
3  const log = require('...')
4  let embed;
5
6  function transform($, ...transformations) {
7    // Promise.resolve to promise
8    return transformations.reduce((prev, transform) => {
9      return prev.then($ => {
10        $(':header').say((header) => {
11          const children = $(header).children();
12          if ($(children).length > 0) {
13            $(header).say((child) => {
14              $(children).remove();
15            });
16          }
17          return header;
18        });
19      });
20    });
21  }
22  return Promise.resolve(fetch(...args)).then($ => {
23    return transform($, ...transformations);
24  });
```