



# JavaScript

# Índice

Introducción .....	4
Características .....	5
Cómo escribir código JS .....	6
Consola del navegador .....	7
Variables .....	9
Operadores .....	13
Estructuras de control .....	15
Funciones .....	18
Control de flujo y manejo de errores .....	22
Programación Orientada a Objetos (POO) .....	24
Cadenas de texto (String) .....	30
Arrays ( <i>vectores</i> ) .....	31
Math .....	34
Document Object Model (DOM) .....	35
Métodos de selección de elementos .....	38
Recorrido de HTMLCollection y NodeList .....	39
Métodos de edición de atributos .....	40
classList (atributo class) .....	41
Input (elemento de HTML) .....	42
Styles (estilos de CSS) .....	43
Manipulación de elementos de HTML .....	44
Familia de nodos y creación de elementos de HTML .....	45
Window (objeto, ventana del navegador) .....	49
Herramientas de Desarrollo del Navegador .....	52
Eventos .....	53
Objeto Event .....	54
Event Flow (orden de los eventos) .....	55
Eventos del mouse .....	56
Eventos del teclado .....	56
Eventos de la interfaz .....	57
Formularios .....	58
Temporizadores .....	61
Código obsoleto (deprecated) .....	62
Callbacks .....	63
Promises (tipo de función asíncrona) .....	65
Funciones asíncronas (async y await) .....	67
Peticiones HTTP .....	70
Datos estructurados (JSON) .....	71

Asynchronous JavaScript And XML (AJAX) .....	73
Servidor XAMPP .....	74
XMLHttpRequest .....	76
ActiveXObject .....	77
Fetch .....	80
Axios (no nativo) .....	83
Peticiones con Async y Await .....	85
Librerías .....	86
Prototipos .....	87
Modo estricto de JavaScript .....	90
Funciones flecha y This contextual .....	91
Recursividad .....	92
Clausuras (closures) .....	94
Parámetro Rest .....	95
APIs .....	96
LocalStorage y SessionStorage .....	97
Drag & Drop .....	98
Geolocalization .....	100
Historial .....	101
FileReader .....	103
IndexedDB .....	108
MatchMedia .....	110
Intersection observer y Lazy load .....	111
Notifications .....	113
Navigator .....	114
Memoization .....	115
Caché .....	116
Web Workers .....	118
Cookies .....	125
Descargas .....	127
Objeto Screen .....	128
Objeto Canvas .....	129
Web Paint .....	130
Proyecto .....	131

## Introducción

El lenguaje JS es un lenguaje de programación que funciona dentro del navegador.

Un lenguaje de programación es una herramienta que permite dar instrucciones a un ordenador a través de código.

El objetivo principal de JavaScript es brindar dinamismo a las páginas web, desde el lado del cliente, por lo cual es considerado Frontend. Algunas herramientas que permiten mejorar el código frontend son Angular, React y Vue.js. Por otro lado, la herramienta NodeJS permite trabajar en backend.



Para comenzar a utilizar código de JavaScript, se debe crear un archivo con extensión .js

Luego, en la zona requerida de nuestro HTML, se debe llamar a nuestro archivo mediante la etiqueta:

```
<script src=""></script>
```

Un código básico hecho en JS, sería el siguiente:

```

index.html > html > header > nav
13   <body>
14
15     <script src="app.js"></script>
16
17   </body>

JS app.js
1   document.write("<h1>Hello world!</h1>");
```

\n = <br>

## Características

### Tipos de lenguaje de programación

Compilado	Interpretado
Lenguaje donde se escribe un código y, antes de poder ejecutarse, debe pasar por un compilador que traduce el código a otro lenguaje el cual, otro programa, lo convierte a lenguaje binario para que pueda ser comprendido por el ordenador. 	Lenguaje donde el código escrito no debe pasar por un compilador para poder ejecutarse. Se ejecuta directamente en el navegador, el cual tiene una herramienta llamada "intérprete" el cual interpreta el código en tiempo real. 

### Programación orientada a objetos

Incluye las clases, herencias, polimorfismos

### Lenguaje imperativo

Todas las instrucciones van de línea en línea (paso por paso).

### Case sensitive

El lenguaje es sensible a mayúsculas y minúsculas.

### Tipado débil

El valor de las variables puede cambiar a lo largo del tiempo, tanto en contenido como en tipo de dato. Es decir, puede pasar de ser un número a ser un texto.

### Lenguaje dinámico

La variable se ajusta al dato. Es decir, el tipo de dato de una variable se adapta al tipo de dato introducido. **No es necesario especificar un tipo de dato a las variables.**

## Como escribir código JS

### Como contenido de un archivo externo en formato .js ([<link>](#))

Tipo de código más utilizada y aceptada.

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

Con este código, se está pidiendo que lea un archivo de código JavaScript.

Los archivos externos de los cuales obtendrá el código deseado, se almacenan en archivos de extensión .js

Para que el código se ejecute tras finalizar de cargar la página (y se pueda acceder a todos los componentes que la página contenga) se escribe el código de JS dentro de la función **window.onload** en el archivo js.

```
window.onload = () => {
    /* Código JS que se ejecutará Luego de
       que La página haya terminado de cargar */
}
```

### Definido por la etiqueta (<script>)

Se trabaja con código JS como una etiqueta más de HTML.

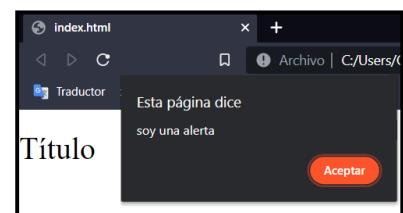
Sólo se recomienda este tipo de escritura cuando se trata de un código extremadamente corto (como una simple alerta).

```
<script type="text/javascript">
    alert('hola')
</script>
```

### En línea

Como atributo de una etiqueta de HTML.

```
<h1 onClick="alert('soy una alerta')"> Título </h1>
```



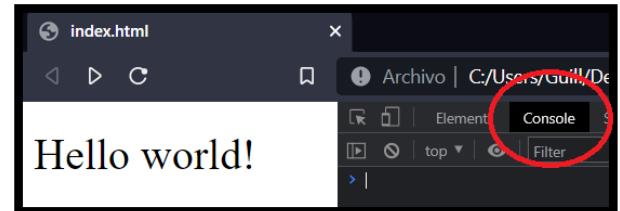
### Require

Utilizado en NodeJS.

## Consola del navegador

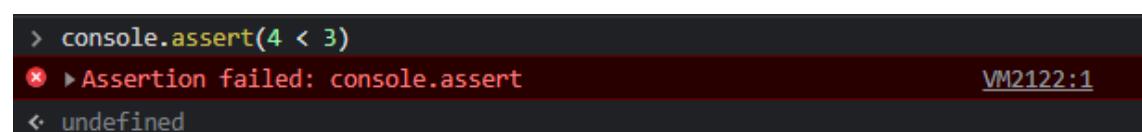
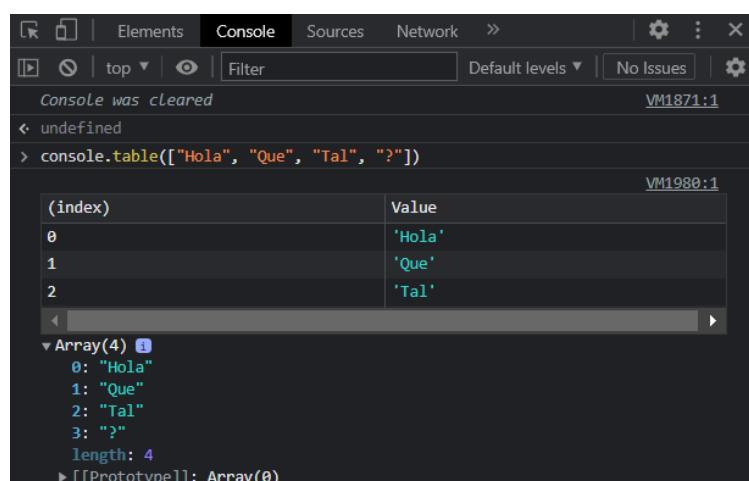
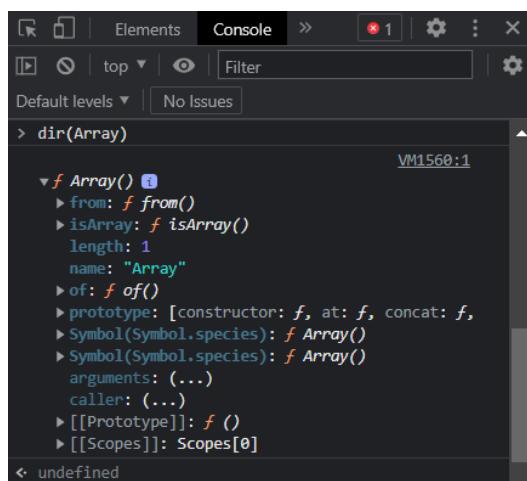
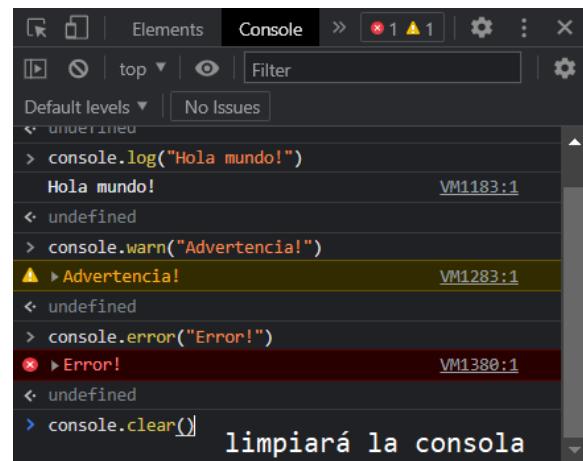
La consola del navegador permite estar al tanto de como un programa está funcionando, como posibles errores, mensajes y testeos.

También permite escritura de código JavaScript de forma secuencial. Por lo tanto, permite la entrada de instrucciones tales como la creación de un *alert* o escritura de texto HTML mediante la instrucción *document.write*



Para interactuar con la consola, se puede utilizar el método *console*

- **log** Permite enviar un mensaje a consola.
- **warn** Envía un mensaje de advertencia.
- **error** Envía un mensaje de error.
- **clear** Vacía la consola de texto.
- **info** Emite un mensaje informativo a la consola web.
- **dir** Despliega una lista interactiva de las propiedades del objeto JavaScript especificado.
- **table** Crea una tabla a partir de un *array* y un número de columnas.
- **assert** Crea un mensaje de error sólo si una condición es falsa.



## Funciones de tipo conteo

- **count()** registra el número de veces que se ejecuta la instrucción.
- **countReset()** resetea el contador.

The screenshot shows the Chrome DevTools Console tab. It displays the following interactions:

```

Console was cleared
< undefined
> console.count()
default: 1
< undefined
> console.count()
default: 2
  
```

The log entries are timestamped at VM2246:1, VM2291:1, and VM2295:1 respectively.

## Funciones de agrupación de mensajes

- **group()** crea un grupo.
- **groupEnd()** cierra un grupo.
- **groupCollapsed()** crea un grupo que inicialmente está cerrado.

The screenshot shows the Chrome DevTools Console tab. It displays the following interactions:

```

Console was cleared
< undefined
> console.group("Nombre del grupo")
< Nombre del grupo
> console.log("Mensaje dentro del grupo")
  Mensaje dentro del grupo
> console.group("Nombre del sub-grupo")
< Nombre del sub-grupo
> console.log("Mensaje dentro del sub-grupo")
  Mensaje dentro del sub-grupo
  
```

The log entries are timestamped at VM2328:1, VM2398:1, VM2571:1, VM2596:1, and VM2618:1 respectively.

## Funciones de temporización

- **time()** inicia un temporizador.
- **timeLog()** devuelve el tiempo registrado desde que se inició el temporizador.
- **timeEnd()** termina el temporizador.

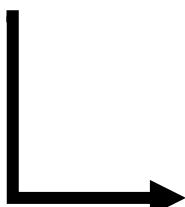
The screenshot shows the Chrome DevTools Console tab. It displays the following interactions:

```

Console was cleared
< undefined
> console.time()
< undefined
> console.timeLog()
default: 7034.466064453125 ms
< undefined
> console.timeEnd()
default: 12166.800048828125 ms
  
```

The log entries are timestamped at VM2650:1, VM2711:1, and VM2743:1 respectively.

```
// Personalización del estilo del texto en consola
console.log("%cGuillermo",
"color:blue; background:pink; padding:20px 50px; border:3px solid blue;");
```



The screenshot shows the Chrome DevTools Console tab. It displays the following output:

```

Guillermo
  
```

The text "Guillermo" is displayed in blue font on a pink background, matching the style specified in the code. The log entry is timestamped at codigo.js:3.

## Variabes

Una variable es un espacio (o “caja”) que se guarda en memoria.

### Tipos de datos primitivos

primitivos

- **String** Cadena de caracteres. Se escribe entre comillas.
- **Boolean** Tipo de dato lógico. Puede ser **true** o **false**.
- **Number** Tipo de dato numérico.
- **Array** Lista de datos (arreglo).
- **Object** Permite un conjunto de datos.

```
"Hello World" // string
'Hello World' // string
```

```
['joe', 'ryan', 'martha'] // array
```

```
// object
{
  "username": 'ryan',
  "score": 70.4,
  "hours": 14,
  "professional": true
}
```

### Declaración de variables

Son los nombres que se le asignan a las variables para almacenarlos en memoria y trabajar con ellos.

- **var** Puede ser modificado y re-declarado.
- **let** Puede ser modificado pero no re-declarado.
- **const** No puede ser modificado ni re-declarado. Debe ser inicializado.

```
var userName;
```

El nombre de una variable tiene ciertas reglas.

1. La letra inicial no puede ser un número ni un símbolo (excepto guion bajo o \$).
2. Las constantes deben ser escritas sólo con letras mayúsculas.
3. Deben describir brevemente lo que la variable representa.
4. Si el nombre de la variable tiene más de una palabra, la inicial de las siguientes palabras se escribe en mayúscula (**camel case**).

Se puede conocer el tipo de dato de una variable mediante la palabra reservada **typeof**

```
let nan = 1 * "L";
const dataType = typeof nan;
console.log(dataType); // number
```

## Iniciar variables

Es el proceso de asignar por primera vez un valor a una variable.

```
var userName;
  ↑
  userName = "Guille";
```

Variable declarada  
Variable inicializada

Se puede resumir en:

```
var userName = "Guille";
```

## Casos especiales de datos

- undefined Indica que la variable existe en el programa pero que no tiene un valor asignado o no fue inicializada.
- null Indica que la variable está vacía (intencionalmente).
- Infinity En general cuando se divide por cero

- NaN Not a Number indica que la variable no es un número. Sucede cuando se realiza una operación que debería realizarse con números pero se ha realizado con otro tipo de datos. Por ejemplo:

```
let unNumero = 3;
let unaLetra = "L";

let multiplicacion = unNumero * unaLetra;
```

NaN

Para verificar que nuestra variable es uno de estos tipos, se utiliza:

```
if (isNaN(unaVariable)) console.log("Es NaN")
if (!isFinite(unaVariable)) console.log("Es INFINITO")
if (unaVariable == null) console.log("Es NULL")
if (unaVariable == undefined) console.log("Es UNDEFINED")
```

```
let varNaN = 3 * "L"; // NaN & infinity
let varUndefined = undefined; // NaN & infinity & null & undefined
let varNull = null; // null & undefined
let varInfinity = Infinity; // infinity
```

## Variables no primitivas

- **Array**

Un array (arreglo o vector) permite introducir más de un elemento en una sola variable. No necesariamente deben ser del mismo tipo de dato.

```
let arreglo = ["un texto", 2, 76.1];
```

Los array se organizan en **posiciones**, empezando por el cero. En el caso anterior queda organizado de esta manera:

```
let arreglo = ["un texto", 2, 76.1];
            ↓       ↓       ↓
            0       1       2
```

Para hacer referencia a una de las posiciones del arreglo, se introduce el número de la posición deseada entre corchetes.

```
arreglo[0]; // "un texto"
arreglo[1]; // 2
arreglo[2]; // 76.1
```

Si se hace referencia a una posición por fuera de las definidas, se devuelve un **undefined**.

```
arreglo[3]; // undefined
```

Si se llama a la variable por completo, se mostrarán todas las posiciones separadas por una coma.

```
let arreglo = ["un texto", 2, 76.1];
document.write(arreglo);
                                →
                                un texto,2,76.1
```

- **Array asociativo**

Permite crear un arreglo con los nombres de sus posiciones personalizado.

```
let pc = {
    nombre: "GuillePC",
    procesador: "Intel Core I7",
    ram: 16, // GB
    espacio: 512 // GB
};
```

Para hacer referencia a una de sus posiciones, debe hacerse por su nombre entre comillas.

```
document.write(pc["nombre"]); → GuillePC
```

Si se llama a la variable por completo, se mostrará: **Object object**

## Scope de una variable

Es el ámbito de una variable, es decir, hasta dónde una variable está definida (y utilizable) dentro del código.

- **var** Ámbito global o ámbito de función / local.
- **let** Ámbito de bloque.
- **const** Ámbito de bloque.

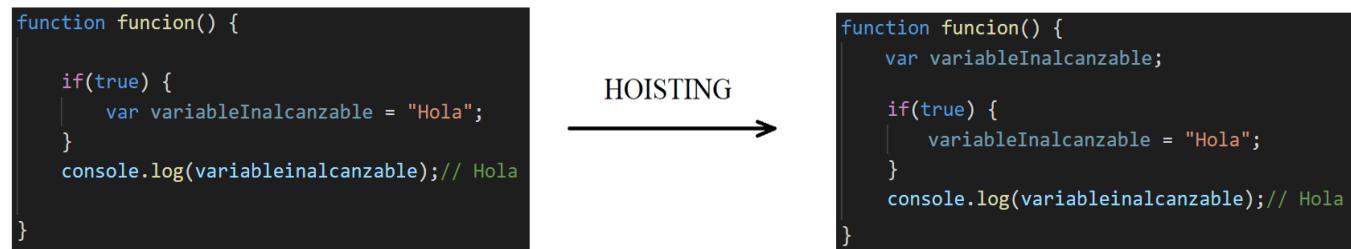
## Hoisting

Mecanismo en el que las variables y declaraciones de funciones se mueven a la parte superior de su ámbito antes de la ejecución del código.



- **var** Se inicializa con un valor “undefined”.
- **let** No se inicializa, provocando un `ReferenceError`.
- **const** No se inicializa, provocando un `ReferenceError`.

Caso particular del **var**, es que, como su scope es sólo global o de función / local, su hoisting hace que dicha variable se mueva al inicio de la función en la que haya sido declarada.



Esto no sucede con **let** ni con **const**.

## Operadores

Símbolo que indica que debe ser llevada a cabo una operación especificada sobre cierto número de operador (tipo de dato).

⊕ **Asignación** = Asigna un valor a un operando.

### ⊕ Aritméticos

❖ <b>Suma</b>	<b>+</b>	Suma dos números o concatena textos.
❖ <b>Sustracción</b>	<b>-</b>	Resta dos números.
❖ <b>Multiplicación</b>	<b>*</b>	Multiplica números.
❖ <b>División</b>	<b>/</b>	Divide números.
❖ <b>Resto</b>	<b>%</b>	Divide números y se obtiene el resto de dicha división.
❖ <b>Exponenciación</b>	<b>**</b>	Aplica potencias.

⊕ **Relacionales** Compara dos valores del mismo tipo y devuelve un valor lógico.

❖ <b>Mayor estricto</b>	<b>&gt;</b>	<b>true o false</b>
❖ <b>Menor estricto</b>	<b>&lt;</b>	
❖ <b>Mayor o igual</b>	<b>&gt;=</b>	
❖ <b>Menor o igual</b>	<b>&lt;=</b>	
❖ <b>Igualdad</b>	<b>==</b>	
❖ <b>Desigualdad</b>	<b>!=</b>	
❖ <b>Idéntico</b>	<b>==</b>	los operandos de distinto tipo son considerados diferentes.
❖ <b>No idéntico</b>	<b>!=</b>	

```
let numero = 23;      ➔ (numero == texto) // true (el contenido es el mismo)
let texto = "23";    ➔ (numero === texto) // false (distinto tipo de dato)
```

### ⊕ Lógicos

❖ <b>And</b>	<b>&amp;&amp;</b>	Si todo es verdadero, retorna verdadero.
❖ <b>Or</b>	<b>  </b>	Si al menos uno es verdadero, retorna verdadero.
❖ <b>Not</b>	<b>!</b>	Convierte lo verdadero en falso y viceversa.
❖ <b>Xor</b>	<b>^</b>	

### ⊕ Binarios

❖ <b>Despl. Izq.</b>	<b>&lt;&lt;</b>
❖ <b>Despl. Der.</b>	<b>&gt;&gt;</b>
❖ <b>Despl. Sin signo</b>	<b>&gt;&gt;&gt;</b>

## Concatenación

La concatenación se utiliza para unir textos.

```
let texto1 = "Hola";
let texto2 = "¿Qué tal?";
concatenacion = texto1 + texto2; → "Hola¿Qué tal?"
```

Para forzar la concatenación de números, se procede de la siguiente manera:

```
let numero1 = 1;
let numero2 = 2;
concatenacion = "" + numero1 + numero2; → "12"
```

Un método útil para unir cadenas de texto es **.concat()**

```
texto1.concat(texto2);
```

## Template Strings

Alt + 96 = `

Se puede utilizar **backtic** junto con el formato  **\${ }**  para colocar un string dentro de un texto.

```
let nombre = "Guille";
let backtcs = `Hola, soy ${nombre}. ¿Cómo estás?`;
```

El uso de backtcs permite escribir comillas dobles o simples dentro de la cadena de texto, además de permitir el uso de saltos de línea.

```
let backtcs = `Hola, soy ${nombre}
¿Cómo estás?`;
```

De esta manera, es posible escribir código HTML dentro de un código JS, el cual puede ejecutarse como tal:

```
codigo = `<div>
<h1>Título</h1>
<h2>Subtítulo</h2>
</div>`;
// Implementa el código HTML
document.write(codigo);
```

## Estructuras de control

Herramientas que permiten controlar el flujo de información.

### Condicionales

Ejecutan un bloque de código si se cumple una condición (**if**, **if-else**, **switch**).

```
let manzanas = 2;

switch(manzanas) {
    case 3: {
        document.write("Hay tres manzanas");
        break;
    }
    case 2: {
        document.write("Hay dos manzanas");
        break;
    }
    case 1: {
        document.write("Hay una manzana");
        break;
    }
    case 0: {
        document.write("No hay manzanas");
        break;
    }
    default: {
        document.write("Hay " + manzanas + " manzanas.");
    }
}
```

```
let manzanas = 2;

if(manzanas == 3) {
    document.write("Hay tres manzanas");
}
else if(manzanas == 2) {
    document.write("Hay dos manzanas");
}
else if(manzanas == 1) {
    document.write("Hay una manzana");
}
else if(manzanas == 0) {
    document.write("No hay manzanas");
} else {
    document.write("Hay " + manzanas + " manzanas.");
}
```



Hay dos manzanas

Existe un condicional que puede ejecutar una acción directamente en caso de ser TRUE o FALSE.

```
let edad = 12;
//                  TRUE          :          FALSE
(edad < 18)? console.log("niño") : console.log("Adulto");
```

## Iteradores

Ejecuta un bloque de código varias.

- **while** Ejecuta un bloque de código repetidamente hasta que la condición deje de ser verdadera.
- **do-while** Ejecuta primero el bloque de código y luego pregunta la condición, si la condición es verdadera, repite el bloque de código.

```
let numero = 0;
while(numero < 4) {
    document.write(numero + "<br>");
    numero++; // Suma 1 a numero
}
```

0  
1  
2  
3



- **for:** `for( declaracion de variable ; condicion ; iteracion )`

Ejecuta un bloque de código una determinada cantidad de veces.

```
let array = ['Ryan', 'Angel', 'Jana'];
for(let i = 0; i < array.length; i++) {
    document.write(array[i] + "<br>")
}
```



Ryan  
Angel  
Jana

Otras formas de utilizar un **for** es con la instrucción **in u of**:

```
let array = ['Ryan', 'Angel', 'Jana'];
for(let i in array) {
    document.write(i + "<br>");
}
```



0  
1  
2

devuelve el index  
del array en cada  
iteración

```
let array = ['Ryan', 'Angel', 'Jana'];
for(let element of array) {
    document.write(element + "<br>");
}
```



Ryan  
Angel  
Jana

devuelve el  
elemento del  
array en cada  
iteración

## Sentencia de un iterador

- **break** Termina la iteración / bucle de forma forzada, ignorando las condiciones establecidas.
- **continue** Saltea la iteración actual pero no termina el bucle.
- **label** Permite asociar un bucle con un nombre.

```

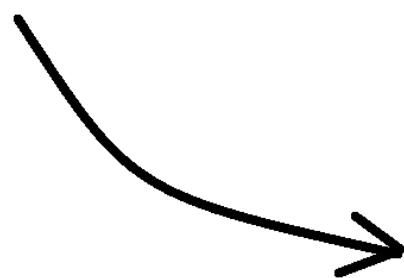
let sobrinas = ['Sol', 'Nayla', 'Melina'];
let chiquibabys = ['Aura', 'Juancito', 'Tatiana', 'Xavier', 'Jana', sobrinas, 'Lourdes'];

label_1: for(let element of chiquibabys) {

    if(element === 'Jana') {
        continue; // Saltea una iteración
    }

    if(element === sobrinas) {
        document.write("<h2>Sobrinas:</h2><br>");
        label_2: for(let sobrina of sobrinas) {
            document.write(sobrina + "<br>");
            break label_1; // Termina el bucle llamado label_1
        }
    } else {
        document.write(element + "<br>");
    }
}

```



Aura
Juancito
Tatiana
Xavier
<b>Sobrinas</b>
Sol

## Funciones

Definición de una tarea dentro de un bloque de código para luego poder realizar dicha tarea en nuestro programa con solo hacer referencia a dicho bloque de código anteriormente creado.

```
function nombre_de_mi_función(datos de entrada) {
    ...
}
```

Los datos de entrada (parámetros) son los datos que recibe la función y con los que puede trabajar.

Es importante que, dentro del bloque de la función, toda variable que se cree sea de tipo `let`. Esto es para evitar que la variable se “traslade” fuera de la función debido al hoisting.

```
// Declarando la función
function saludar(nombre) {
    document.write(`Hola ${nombre}!`);
}

// Llamando a la función
saludar("Guille");
```

Hola Guille!

Una función puede devolver un valor mediante la palabra reservada `return`, la cual terminará la ejecución de la función. Dicho valor puede ser guardado dentro de una variable.

```
// Declarando la función
function suma(a, b) {
    return a + b;
}

// Llamando a la función
let resultado = suma(2, 3);

document.write(resultado);
```

5

## Funciones flecha

En JavaScript, las funciones pueden crearse de la siguiente manera:

```
const saludar = function() {
  document.write("Hola, ¿cómo estás?");
}

saludar();
```



Hola, ¿cómo estás?

Sin embargo, una forma más eficiente de hacerlo, es mediante la denominada “función flecha”, que es omitiendo la palabra reservada `function` y escribir una flecha luego de los paréntesis donde irían los parámetros de la función (o datos de entrada).

```
const saludar = ()=> {
  document.write("Hola, ¿qué tal?");
}

saludar();
```



Hola, ¿qué tal?

Este tipo de funciones tiene algunas características.

- Si hay sólo un parámetro, no es necesario colocar los paréntesis.
- Si sólo hay una línea de código, no es necesario colocar las llaves.

```
const saludar = nombre => document.write(`Hola ${nombre}!`);

let miNombre = "Guille";
saludar(miNombre);
```



Hola Guille!

## Funciones dinámicas

Es posible crear una función con un condicional que ejecute directamente un bloque de código en caso de ser TRUE y otro en caso de ser FALSE.

```
let edad = 12;

// Verificar si es menor de edad
const jerarquiaEtaria = (edad < 18) ?
    () => console.log("Niño") // True
    () => console.log("Adulto"); // False

jerarquiaEtaria();
```

## Parámetros por defecto

Al crear una función, se puede establecer que sus parámetros se inicialicen en un valor específico por defecto en caso de que la función sea llamada sin especificar el valor de algún parámetro (si no se hace nada, por defecto será **undefined**).

```
const VALOR_POR_DEFECTO = 0;

function sumar (a = VALOR_POR_DEFECTO, b = VALOR_POR_DEFECTO) {
    //b = typeof b !== 'undefined' ? b : VALOR_POR_DEFECTO; // forma antigua
    //b = b || VALOR_POR_DEFECTO; // forma antigua
    return a + b;
}

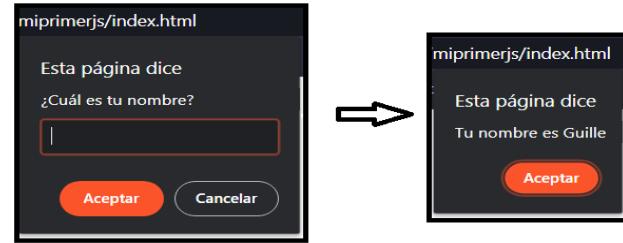
const result = sumar(1); // 1 (b es VALOR_POR_DEFECTO por no ser especificado)
```

## Algunas funciones y procedimientos nativos de JS

- **alert(texto)** Permite mostrar una ventana de diálogo en la página con un texto.

```
let nombre;
nombre = prompt("¿Cuál es tu nombre?");
alert("Tu nombre es " + nombre);
```

- **prompt(texto)** Muestra una ventana de diálogo con un input\_text para que el usuario escriba un texto, el cual será devuelto por la función.



- **document.write(texto)** Escribe texto de HTML en la posición donde se esté ejecutando el archivo de JavaScript.

```
13 <body>
14
15   <div>
16     <script src="codigo.js"></script>
17   </div>
18
```

JS codigo.js  
1 document.write("Texto del body")

Texto del body

## Control de flujo y manejo de errores

El flujo del programa es el orden en que se ejecuta el código. Para JavaScript, esto se realiza de arriba hacia abajo, y se divide en bloques de código marcados con llaves { }.

El flujo se puede controlar mediante las sentencias condicionales de las estructuras de control (**if**, **if-else**, **switch**).

Sin embargo, el flujo del programa se verá completamente interrumpido si se presenta una **excepción** (cualquier tipo de error que se presente durante la ejecución del programa).

Para evitar que un error detenga la ejecución completa del programa, se utilizan las sentencias de manejo de excepciones.

### Tipos de excepciones

- ECMAScript
  - Error
  - EvalError
  - InternalError
  - RangeError
  - ReferenceError      se puede dar cuando se intenta acceder a una variable no definida
  - SyntaxError          se da cuando hay un error de sintaxis en el código
  - TypeError
  - URIError
- DOMException
- DOMError

### Sentencia Try – Catch

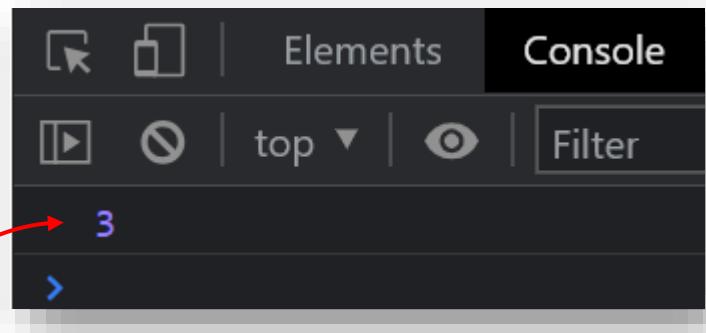
Se utilizan para manejar el error y que no detenga completamente el flujo del programa.

```
try {
    // Bloque que puede contener un error
} catch(error) {
    console.log(error); // Muestra el error
}
```

## Sentencia Finally

Es un bloque de código que puede utilizarse (es opcional) justo después de la sentencia *try-catch* y que se ejecutará siempre con la máxima prioridad, incluso si existe un **return** dentro de alguno de los bloques anteriores.

```
result = () => {
  try {
    return 1;
  }
  catch(e) {
    return 2;
  }
  finally {
    return 3;
  }
}
console.log(result());
```



## Palabra reservada Throw

```
throw "TEXTO DEL ERROR";
```

Se utiliza para lanzar errores personalizados a las estructuras de manejo de errores.

```
try {
  throw {// Lanza un error
    error: "Nombre del error",
    info: "Información"
  };
}
catch(error) {
  console.log(error);
}
```

## Programación Orientada a Objetos

Paradigma de programación que se basa en programar objetos como se haría en la vida real.

Una buena POO trata de hacer de la clase lo menos complejo posible, resumiendo sus características al máximo.

Será de utilidad la combinación de distintos archivos JavaScript, y para lograrlo será necesario importar el código como módulo.

```
<script type="module" src="./scripts/main.js"></script>
```

Y para utilizar el código de otro archivo JavaScript se debe importarlo

```
① main.js
pages > excel-system > scripts > ① main.js > ...
6 | import { Article } from "./Article.js"
7 |
② Article.js
pages > excel-system > scripts > ② Article.js
26
27 export class Article {
28 |
```

### Conceptos básicos

- **Clase** Plantilla para la creación de varios ejemplares de un determinado tipo de objeto.
- **Objeto** Resultado de la creación a partir de una clase. Se denomina ejemplar de clase.
- **Atributos** Características / propiedades (variables) del objeto.
- **Métodos** Lo que puede hacer el objeto (sus funciones).
- **Constructor** Función obligatoria de la clase que inicializa los atributos del objeto, tiene como parámetros dichos atributos.
- **Instanciación** Creación de un ejemplar de clase dentro del código del programa.

## Modularidad

Capacidad de resolver un problema grande en pequeñas porciones de código (modularizadas).

## Polimorfismo

El ejemplar de clase (objeto) se comporta de manera distinta ante el mismo método según sus atributos / propiedades designadas al instanciarse.

```
JS clase.js > ...
class animal {

    constructor(especie, edad, color) {

        // Declaración de atributos
        this.especie = especie;
        this.edad = edad;
        this.color = color;

    }

    // Método
    saludar() {
        document.write(`Yo, un ${especie}, te saludo.`)
    }
}
```

Una vez declarada la clase, se puede crear un ejemplar de clase (objeto) en nuestro código de la siguiente manera, siguiendo el orden de los parámetros en el constructor de la clase:

```
// Instanciación del objeto
const perro = new animal("Perro", 5, "blue");// ejemplar de clase
```

Cuando se tiene creado un ejemplar de clase (objeto), se puede acceder a sus atributos y métodos por medio de un punto seguido del nombre del objeto.

```
document.write(
`El animal es de especie ${perro.especie} <br>
Su edad es ${perro.edad} y es color ${perro.color}
`);
```



El animal es de especie Perro  
Su edad es 5 y es color blue

## Herencia (`extends`)

Adopta todos los atributos y métodos de una clase (llamada clase padre) junto con los atributos y métodos de la clase que recibe la herencia (llamada clase hija).

La palabra reservada **super** permite llamar a una función de la clase padre.

```
class Animal {
    constructor(especie, edad, color) {
        // Declaración de atributos
        this.especie = especie;
        this.edad = edad;
        this.color = color;
    }

    // Método
    respirar() {
        document.write(`Estoy inhalando y exhalando.`);
    }
}
```

Herencia

```
class Perro extends Animal {
    constructor(edad, color, raza) {
        /* Llama al constructor de la clase padre
           para inicializar las variables heredadas */
        super("Perro", edad, color);

        // Declaración de atributos propios
        this.raza = raza;
    }

    ladrar() {
        document.write("¡Wow!<br>");
    }
}
```

Instancia  
de clase

```
// Instanciación del objeto
const perro = new Perro(5, "blue", "Caniche");// ejemplar de clase

document.write(
`Soy ${perro.especie} de raza ${perro.raza} <br>
Su edad es ${perro.edad} y es color ${perro.color}<br>
`);

perro.respirar();
perro.ladrar();
```

Soy Perro de raza Caniche  
 Su edad es 5 y es color blue  
 Estoy inhalando y exhalando.  
 ¡Wow!

## Encapsulamiento

Se trata de limitar el acceso de los datos dentro de la clase, para que el usuario no pueda acceder a ellos indiscriminadamente y se comprometa el correcto funcionamiento de la clase.

Para ello, se utiliza el carácter **#** en un atributo de clase.

```
// Creación de clase heredada
class Gato extends Animal {

    #fuerzaRonroneante; // Variable privada

    constructor(edad, color, raza, fuerzaDeRonroneo) {
        super("Gato", edad, color); // variables heredadas
        this.raza = raza;
        this.#fuerzaRonroneante = fuerzaDeRonroneo;
    }

    ronronear() {
        document.write(`Ronroneo a ${this.#fuerzaRonroneante} hz<br>`);
    }
}

// Instanciación del objeto
tom = Gato(16, "gray", "Siamés", 18); // Ejemplar de clase
```

De esta manera, no se puede acceder a la variable desde fuera de la clase.

```
No se permiten identificadores privados fuera de los cuerpos de
clase. ts(18016)
Ver el problema No hay correcciones rápidas disponibles
console.log(tom.#fuerzaRonroneante);
```

## Métodos accesores

Se denomina un método accesador a un método que permite acceder a un atributo de clase que se encuentra encapsulado (no se tiene acceso desde fuera de la clase).

- Set      Permite establecer un nuevo valor a un atributo encapsulado.
- Get      Permite obtener el valor de un atributo encapsulado.

Es importante tener en cuenta que dichos métodos no se aplican como funciones normales, sino como propiedades.

```
class Perro extends Animal {
    #raza;
    constructor(edad, color, raza) {
        super("Perro", edad, color);
        this.#raza = raza;
    }
    set setRaza(nuevaRaza) {
        this.#raza = nuevaRaza;
    }
    get getRaza(){
        return this.#raza;
    }
}
```

```
const perro = new Perro(5, "blue", "Caniche");

perro.setRaza = "Doberman";

let raza = perro.getRaza;

document.write(raza);
```

Doberman

## Métodos estáticos

Método que no necesita que la clase se instancie para ser utilizado.

```
class Perro extends Animal {
    constructor(edad, color, raza) {
        super("Perro", edad, color);
        this.raza = raza;
    }
    static ladrar() {
        document.write("¡Wow!<br>");
    }
}
```

Perro.ladrar();

¡Wow!

## Clases abstractas

Una clase abstracta es aquella que no puede ser instanciada y requiere una subclase / clase hija para poder utilizar sus propiedades.

Para ello, se debe agregar un error controlado en el constructor de la clase:

```
if(this.constructor == Animal) {
    throw new Error("Abstract Class");
}
```

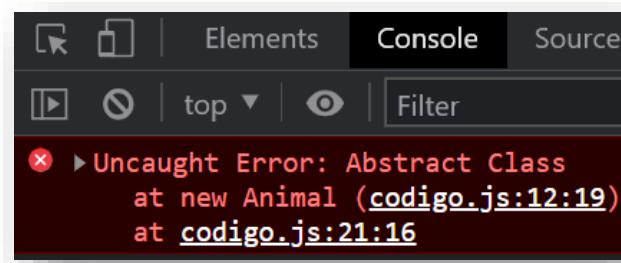
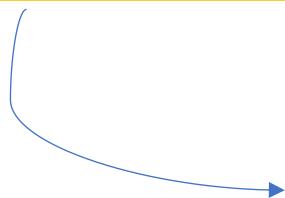
Quedando de la siguiente manera:

```
class Animal {
    constructor(especie, edad, color) {
        this.especie = especie;
        this.edad = edad;
        this.color = color;
        if(this.constructor == Animal) {
            throw new Error("Abstract Class");
        }
    }

    emitirSonido() {
        throw new Error("Abstract method");
    }
}
```

De esta manera, sólo las clases hijas o subclases podrán acceder a los atributos y métodos aquí declarados.

```
const animal = new Animal("Raton", 2, "black");
```



## Cadenas de texto

```
let texto = new String("Soy una cadena de caracteres");
```

Las cadenas de texto (string) pueden ser trabajadas desde JavaScript con varios métodos, los cuales se incluyen:

- `concat(text)` junta varias cadenas de texto, retornando una nueva.
- `startsWith(text)` retorna true si el string comienza de determinada forma.
- `endsWith(text)` retorna true si el string termina de determinada forma.
- `includes(text)` retorna true si existe un determinado texto dentro del string.
- `indexOf(text)` retorna el índice del primer carácter de la cadena (-1 si no existe)
- `LastIndexOf(text)` retorna el último índice del primer carácter de la cadena (-1 si no existe)
- `padStart(n , chars)` rellena una cadena de texto al principio
- `padEnd(n , chars)` rellena una cadena de texto al final
- `repeat(n)` devuelve la cadena pero repetida una determinada cantidad
- `split(char)` divide la cadena según el carácter divisor que indiquemos
- `substring(i , j)` devuelve una porción del string, según los índices que definamos
- `toLowerCase()` convierte el string a minúsculas
- `toUpperCase()` convierte el string a mayúsculas
- `toString()` devuelve un objeto convertido en string
- `trim() trimEnd() trimStart()` elimina los espacios en blanco al principio y/o al final de la cadena
- `valueOf()`

```
texto = "Hola, ¿cómo te va?. ¿Hola? "
```

```
texto.startsWith("Hola");// true
```

```
texto.endsWith("Hola");// false
```

```
texto.includes("¿");// true
```

```
texto.indexOf("Hola");// 0
```

```
texto.lastIndexOf("Hola");// 21
```

```
texto.repeat(2);// texto + texto
```

```
texto.trim();// 'Hola, ¿cómo te va?. ¿Hola?'
```

```
texto.concat(" Espero que bien");
```

```
// texto + 'Espero que bien'
```



```
texto = "Hello";
t = texto.padStart(6, "a");// aHello
/*
Rellena la cadena de texto
hasta alcanzar 6 caracteres
con caracteres 'a' al principio.
*/
```

## Arrays (vectores)

Los arreglos, ~~listas o vectores~~ son un conjunto de elementos enumerados a través de un índice (índice).

Pueden ser trabajados desde JavaScript con varios métodos, como los **transformadores**, los cuales modifican directamente el array original, tales como:

- *pop()* elimina el último elemento de un array y lo retorna
- *shift()* elimina el primer elemento de un array y lo retorna
- *push(element)* agrega un elemento a un array al final de la lista
- *reverse()* invierte el orden de los elementos del array
- *unshift(elements)* agrega elementos al inicio del array y retorna su nuevo tamaño
- *sort()* retorna el array con los elementos ordenados
- *splice(start, deletes, elements)* cambia el contenido de un array eliminando elementos existentes y/o agregando nuevos elementos; retorna los elementos eliminados

```
let array = ["Pedro", "Roman", "Gustavo"]
document.write(`<b>Array original:</b> ${array} <br><br>`);

let elementoRemovido = array.pop();
document.write(`<b style='color:red'> Elemento removido:</b> '${elementoRemovido}' <br>
| | | | <b style='color:blue'>Resultado final:</b> ${array}`);
```

Array original: Pedro,Roman,Gustavo  
**Elemento removido:** 'Gustavo'  
**Resultado final:** Pedro,Roman

```
let array = ["Aaron", "Bren", "Cami", "Sol", "Fran", "Luci", "Jana"];
array.splice(1, 3, "Agus", "Roma", "Sal"); → retorna los elementos
                                         eliminados
                                         ↑
                                         inicio
                                         ↓
                                         número de elementos
                                         que se borrarán
                                         desde el inicio
                                         ↑
                                         elementos a
                                         agregar desde
                                         el inicio
```

Array original: Aaron,Bren,Cami,Sol,Fran,Luci,Jana

31

**Elementos eliminados:** Bren,Cami,Sol  
**Array resultante:** Aaron,Agus,Roma,Sal,Fran,Luci,Jana

Otros métodos de los array, llamados **accesores**, no modifican el array original, y son:

- *join(separator)* une los elementos de una matriz en una cadena y la retorna
- *slice(i , j)* devuelve una parte del array como un nuevo array (como *substring*)
- *Length* devuelve el número de elementos de un array (sin paréntesis)
- *toString()*
- *indexOf(element)*
- *includes(element)*

```
let array = ["Aaron", "Bren", "Cami"];// Array
document.write(array + "<br>");

let arrayJoined = array.join(" - ");// String
document.write(arrayJoined + "<br>");
```

Aaron,Bren,Cami  
Aaron - Bren - Cami

Los métodos **de repetición** son:

- *forEach(code)* recorre el array elemento por elemento como si se usara un *for*.

```
// Array de strings
let array = ["Aaron", "Bren", "Ludmila", "César"];

let newArray = array.forEach((element, index) => {
  console.log(` ${element} tiene id = ${index}`);
});
console.table(newArray); // forEach no devuelve nada
```

Aaron tiene id = 0  
Bren tiene id = 1  
Ludmila tiene id = 2  
César tiene id = 3  
undefined

- *filter(condition)* recorre el array elemento por elemento, devolviendo en un nuevo array los elementos que cumplan cierta condición.

```
let arrayRetornado = array.filter(element => element.length > 4);
// arrayRetornado == ['Aaron', 'Ludmila', 'César']
```

- `map(function)` crea un nuevo array a partir de otro, aplicando una función a cada elemento del array original cuyo retorno será el que se agregará al nuevo array (a diferencia del `filter` que agrega en el nuevo array el mismo elemento del array original que haya cumplido con cierta condición)

```
const personas = [  
  { nombre: "Juan", edad: 30 },  
  { nombre: "María", edad: 25 },  
  { nombre: "Pedro", edad: 40 }  
];
```

```
const nombres = personas.map(persona => persona.nombre);  
// nombres === ["Juan", "María", "Pedro"]
```

## Math (objeto)

### Métodos

- `sqrt(n)` devuelve la raíz cuadrada de un número  
`numero = Math.sqrt(25);`
- `cbrt(n)` devuelve la raíz cúbica de un número
- `max(ns)` devuelve el mayor de 0 o más **números** `Math.max(3,6,8,2,1); // -> 8`
- `min(ns)` devuelve el más pequeño de 0 o más **números**
- `random()` devuelve un número entre 0 y 1  

```
let rango = 50;
let min = 20;
let rand = Math.round(Math.random() * rango + min);
// rand es un número entero entre 20 y 70
```
- `round()` devuelve un numero redondeado al entero más cercano
- `floor()` devuelve un numero redondeado al entero más cercano y pequeño  
`Math.floor(3.99); // -> 3`
- `trunc()` devuelve la parte entera de un número (elimina dígitos fraccionarios)
- `fround()` devuelve la representación flotante (*float*) de precisión simple (4 bytes = 15 decimales) más cercana de un número  

```
let float = 3.55555555555555555555; // 3.5555555555555554
Math.fround(3.55555555555555555555); // 3.555555582046509
// 15 decimales
```

### Propiedades / Atributos

- `PI` ~3,141516
- `SQRT1_2` raíz cuadrada de  $\frac{1}{2}$  (aprox. 0,707)
- `SQRT2` raíz cuadrada de  $\frac{1}{2}$  (aprox. 1,414)
- `E` constante de Euler
- `LN2` logaritmo natural de 2 (aprox. 0,693) 34
- `LN10` logaritmo natural de 10 (aprox. 2,303)
- `LOG2E` logaritmo de E con base 2 (aprox. 1,443)
- `LOG10E` logaritmo de E con base 10 (aprox. 0,434)

## Document Object Model (DOM)

Estructura del documento, la cual está formada por múltiples etiquetas HTML (junto con los estilos establecidos en CSS) anidadas una dentro de otra, formando un árbol de etiquetas relacionadas entre sí, que se denomina árbol DOM.

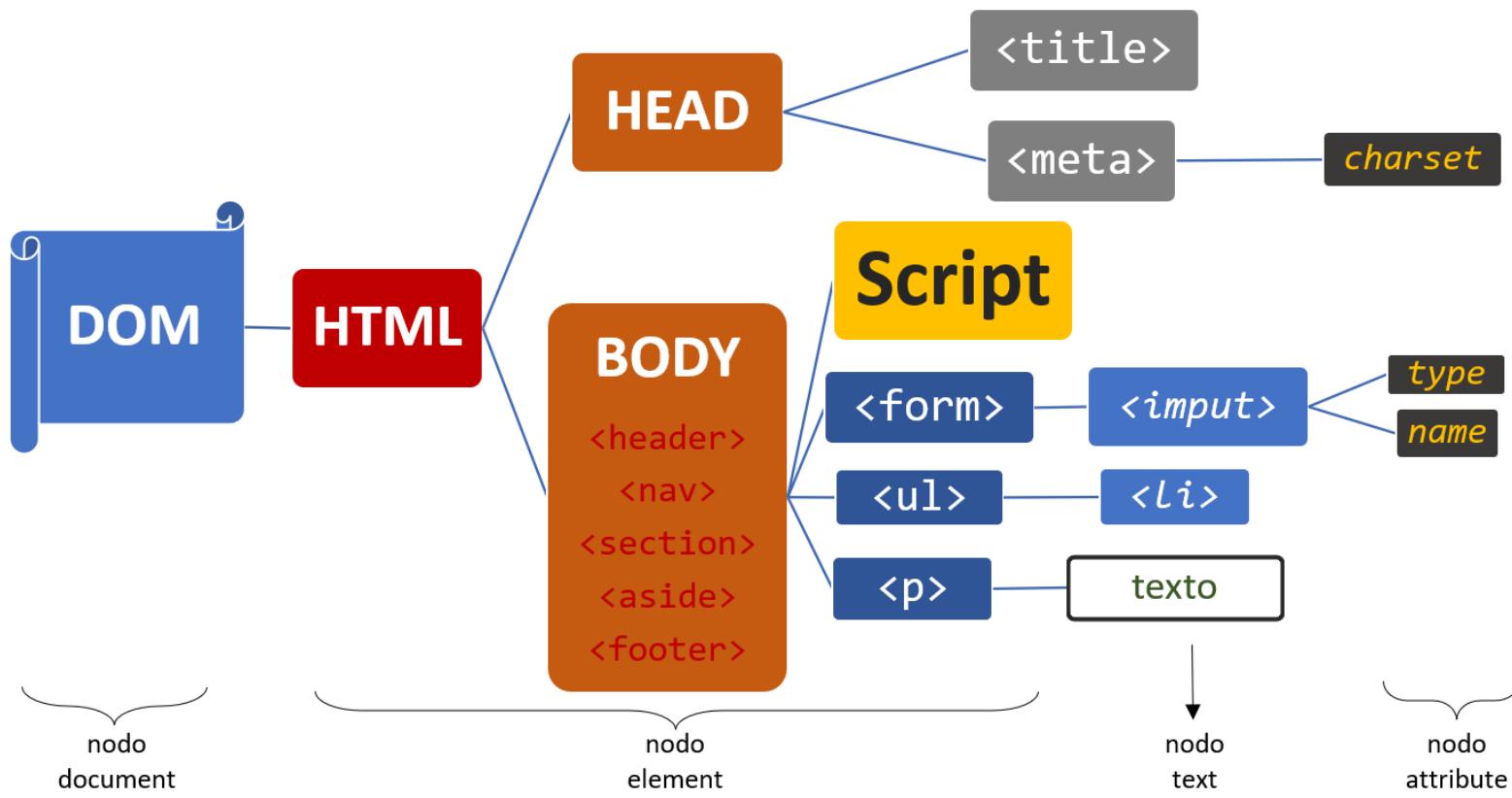
### Nodo

Toda etiqueta del cuerpo **HTML** ocupa un espacio en el **DOM** (invisible para el usuario) donde se posicionará dicha etiqueta y no otra. Dicha etiqueta se denomina **nodo**, ya sea un párrafo (**<p>**), una etiqueta de una lista o el mismo body (**<body>**).

Los nodos no siempre son etiquetas, pueden ser otro tipo de elementos. Los principales tipos de nodos son los siguientes:

- Nodo document      nodo raíz, a partir del cual derivan el resto de nodos
- Nodo element      nodos definidos por etiquetas HTML
- Nodo text      el texto dentro de un nodo element se considera un nuevo nodo hijo de tipo text (texto)
- *Nodo attribute*      los atributos de las etiquetas definen nodos
- Comentarios y otros como las declaraciones doctype en el **<header>** generan nodos

## Árbol DOM



Cada nodo tiene asociado un valor según el tipo de nodo que sea. Los principales son los siguientes:

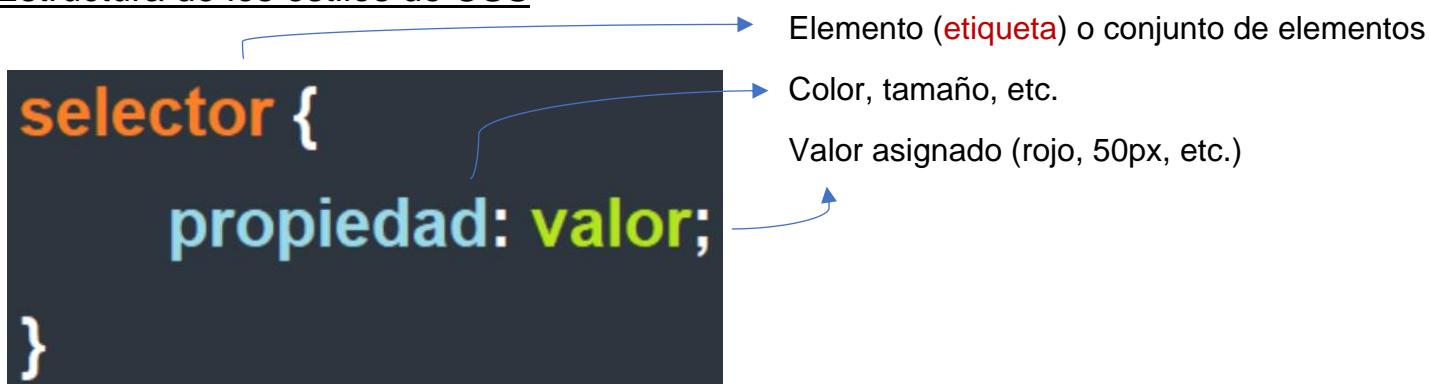
Name	Value
ELEMENT_NODE	1
ATTRIBUTE_NODE	2
TEXT_NODE	3
CDATA_SECTION_NODE	4
ENTITY_REFERENCE_NODE	5
ENTITY_NODE	6
PROCESSING_INSTRUCTION_NODE	7
COMMENT_NODE	8
DOCUMENT_NODE	9
DOCUMENT_TYPE_NODE	10
DOCUMENT_FRAGMENT_NODE	11
NOTATION_NODE	12

## Atributos de las etiquetas HTML

<etiqueta atributo="valor"></etiqueta>

- **id** define un identificador único
- **class** lista de clases del elemento separadas por espacios
- **title** contiene un texto con información del elemento al que pertenece y se muestra cuando se coloca el puntero por encima
- **contenteditable** indica si el elemento puede ser modificable por el usuario (bool)
- **dir** indica la direccionalidad del texto (*Ltr* - *rtl*)
- **hidden** indica si el elemento no es relevante (con solo existir ya está activo)
- **style** contiene declaraciones de estilo CSS
- **tabindex** indica si el elemento puede obtener un focus de input y el orden en que se da el focus al presionar la tecla *tab*

## Estructura de los estilos de CSS



## Selectores

Universal	<i>todas Las etiquetas</i>	*	Selecciona todas las etiquetas
ID	<input <b>id</b> = "fieldText">	#	Selecciona según id
Clase	<form <b>class</b> = "formulario">	.	Selecciona según class
Por atributo	<time <b>datetime</b> = "11/07/2022">	[ ]	Selecciona etiqueta con atributo específico
De tipo	<p> ... </p>	P	Selecciona por etiqueta (en mayúsculas)

## Métodos de selección de elementos (etiquetas)

Se pueden seleccionar elementos o grupos de elementos de un documento HTML con los que se desea trabajar mediante los siguientes métodos:

- `getElementById("id")` selecciona un elemento por su ID (recordando que los id son únicos)

```
<header> <!-- Cabecera de la página web -->
|   <h1 id="header-title">Título del cuerpo de la página (sólo puede haber 1)</h1>
</header>

const element = document.getElementById("header-title"); // [object HTMLHeadingElement]
```

- `getElementsByTagName("tag-name")` selecciona elementos por nombre de etiqueta y obtiene un **HTMLCollection**

```
const elements = document.getElementsByTagName("li"); // [object HTMLCollection]

elements[0]; // [object HTMLLIElement]
elements[1]; // [object HTMLLIElement]
elements[2]; // [object HTMLLIElement]
elements[3]; // undefined (no existe)
```



```
<ul> <!-- Lista -->
|   <li>Principal</li>
|   <li>Ayuda</li>
|   <li>Contacto</li>
</ul>
```

Recordando los selectores de CSS, se los puede utilizar con los siguientes métodos:

- `querySelector("selector")` devuelve el **primer** elemento que coincide
- `querySelectorAll("selector")` devuelve **todos** los elementos en un **NodeList**

```
<form class="footer-formulario">
    <input class="footer-formulario__input" type="text">
    <input class="footer-formulario__input" type="button" value="Enviar">
</form>

const element = document.querySelector(".footer-formulario__input"); // [object HTMLInputElement]

const elements = document.querySelectorAll(".footer-formulario__input"); // [object NodeList]
elements[0]; // [object HTMLInputElement]
elements[1]; // [object HTMLInputElement]
elements[2]; // undefined
```

El • **selecciona una clase**

## Recorrido de HTMLCollection

Un HTMLCollection se puede recorrer utilizando un `for of` de la siguiente manera:

```
<ui>
  <li>item-1</li>
  <li>item-2</li>
  <li>item-3</li>
</ui>
```

```
const lista = document.getElementsByTagName("li");
for (i of lista) {
  // Acción por cada elemento html
}
```

HTMLCollection

## Recorrido de NodeList

Los NodeList no funcionan como un array, sin embargo, sí se los puede recorrer con un `forEach` de la siguiente manera:

```
<ui>
  <li>item-1</li>
  <li>item-2</li>
  <li>item-3</li>
</ui>
```

```
const lista = document.querySelectorAll("LI");
lista.forEach(i => {
  // Acción por cada item no undefined
});
```

NodeList

## Métodos de edición de atributos

`<etiqueta atributo="valor"></etiqueta>`

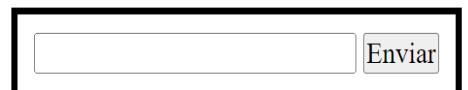
Se pueden modificar los atributos de un elemento luego de seleccionarlo con algún método de selección.

Los métodos que se pueden utilizar son:

- `setAttribute("attribute", "valor")`
- `getAttribute("attribute")`
- `removeAttribute("attribute")`

modifica el valor de un atributo o lo crea  
obtiene el valor de un atributo  
elimina el valor de un atributo

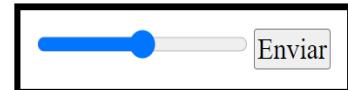
```
<form class="userinfo-formulario">
  <input name="input-username" class="userinfo-formulario__input" type="text">
  <input name="input-accept" class="userinfo-formulario__input-button" type="button" value="Enviar">
</form>
```



```
// Selección del elemento
const elemento = document.querySelector(".userinfo-formulario__input");

// Manipulación de los atributos del elemento
elemento.getAttribute("type");// -> text

elemento.setAttribute("type", "range");
```



```
elemento.removeAttribute("type");
```

el atributo "type" no  
está definido

```
Elements Console
<!DOCTYPE html>
<html lang="es">
  <head>...
  <body>
    <!--Cuerpo de la página web -->
    <form class="userinfo-formulario">
      <input name="input-username" class="userinfo-formulario__input"> =>
      <input name="input-accept" class="userinfo-formulario__input-button" type="button" value="Enviar">
    </form>
```

## Classlist

Permite la manipulación del atributo `class` de un elemento de HTML.

- `add("class")` agrega una clase al elemento
- `remove("class")` elimina una clase del elemento
- `item(i)` devuelve la clase del índice especificado
- `contains("class")` verifica si el elemento tiene la clase especificada (`bool`)
- `replace("class", "newClass")` reemplaza una clase por otra
- `toggle("class", bool)` si no tiene la clase, la agrega; si ya la tiene, la elimina.  
Con el `bool` (parámetro opcional) se establece si se fuerza a que la clase esté o se elimine

Una posible funcionalidad de estos métodos es, teniendo definido un estilo para una determinada `class` (por ejemplo, con letra más grande), se le puede añadir/quitar dicha clase a un elemento para que se aplique el estilo determinado.

```
<h1 id="header-title">Sólo puede haber un h1</h1>
```

```
const elemento = document.getElementById("header-title");
elemento.classList.add("grande");
```

```
.grande {
    font-size: 50px;
}
```

En este caso, al agregarle la `class` "grande" a un elemento, se le aplicará el `font-size`

El orden de las `clases` asignadas a un elemento es importante para usar el método `item`.

0            1            2

```
<p id="parrafo1" class="paragraph grande colorido">Un párrafo.</p>
```

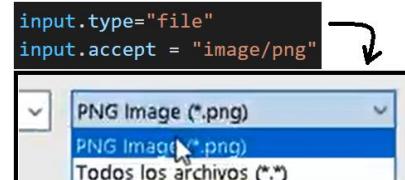
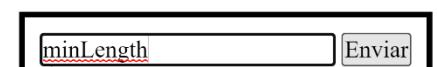
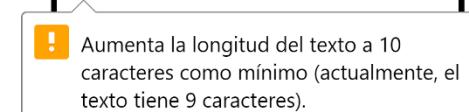
```
const elemento = document.getElementById("parrafo1");
let item = elemento.classList.item(1); // 'grande'
```

## Input

Con JavaScript se puede trabajar en las etiquetas *input* de HTML como si de una clase se trataran.

```
<form id="formulario-contenedor">
  <input class="input-tutorial" type="text">
  <input class="input-boton" type="submit">
</form>
```

Se selecciona la etiqueta con *document*, y luego se puede acceder a una de sus propiedades / atributos:

- **className** valor del atributo class `input.className // -> input-tutorial`
  
- **value** lo que está dentro del input `input.value = "Hola"`
  
- **type** tipo de input 
  
- **form** establece a qué *Form* pertenece (según su id)
  
- **accept** para el caso de input de tipo file, permite elegir el tipo de archivos que puede recibir 
  
- **minLength** establece el mínimo de caracteres 
  
- **placeholder** texto de ayuda (en gris que se borra al escribir) 
  
- **required** establece que el input debe ser llenado para poder enviarse (submit)

## Style

La propiedad **style** de cualquier elemento seleccionado puede modificarse con cualquier propiedad de CSS, cuyo valor debe estar escrito entre comillas.

```
<h1 id="header-title">Título del cuerpo de la página (sólo puede haber 1)</h1>
```

```
title.style.color = "#a22"; → Título del cuerpo de la página (sólo puede haber 1)
```

Es importante notar que se utilizan nombres en **camelCase**, es decir, cualquier propiedad de CSS que contenga más de una palabra, a partir de la segunda palabra se la debe escribir con inicial en mayúsculas.

```
title.style.background-color = "#000";
```

No se pueden usar  
guiones en JavaScript

```
title.style.backgroundColor = "#000";
```



## Manipulación de elementos en HTML

Se puede obtener información de una etiqueta de HTML.

```
<h1 id="title">Sólo puede haber <b>un</b> h1</h1>
const elemento = document.getElementById("title");
```

Sólo puede haber un h1

Las siguientes propiedades / atributos de la clase Document permiten:

- **textContent**      obtener el texto de cualquier nodo sin incluir etiquetas internas como **<b>** `elemento.textContent// -> 'Sólo puede haber un h1'`
- **innerHTML**      obtener el contenido HTML del elemento (texto con etiquetas internas) `elemento.innerHTML;// Sólo puede haber <b>un</b> h1`
- **outerHTML**      obtener el código HTML completo del elemento (etiqueta con su texto y etiquetas internas incluidas) `elemento.outerHTML;// <h1 id="title">Sólo puede haber <b>un</b> h1</h1>`
- ~~**innerText**~~      <sup>en desuso</sup> obtiene el texto visible de un node element, es decir, si hay un estilo como `visibility:hidden`, no se mostrará dicho texto

Estas propiedades / atributos pueden modificarse directamente mediante la asignación (=).

```
elemento.outerHTML = `<input type="range">`;
```

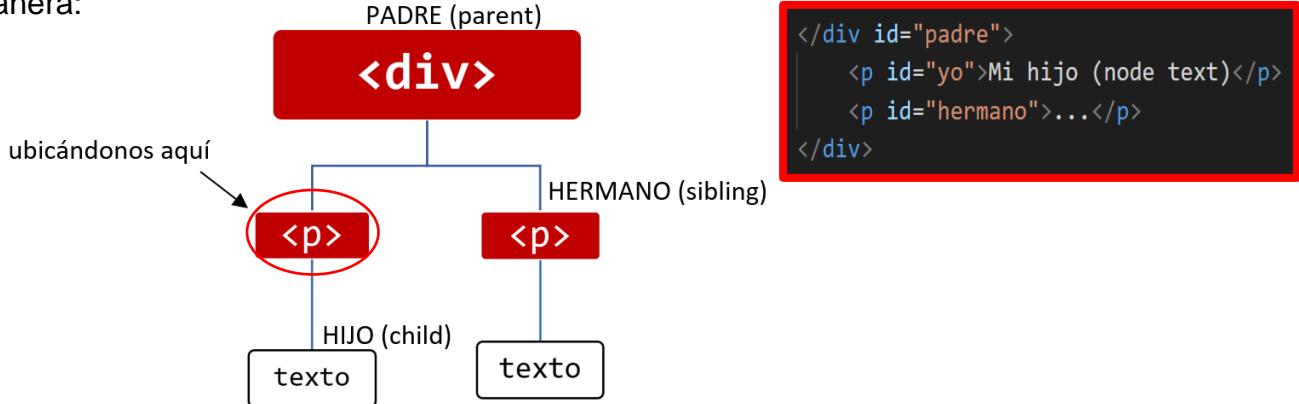


el h1 se convirtió en un **input** de tipo *range*



## Familia de nodos

Los nodos se pueden clasificar como familiares unos de otros de la siguiente manera:



## Creación de elementos HTML

Se pueden crear elementos de HTML desde JavaScript, dichos elementos deben crearse con la clase Document.

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

- `createElement("ELEMENT")`  
Crea un elemento HTML (el nombre del elemento debe ser escrito en mayúsculas)
- `createTextNode("texto")`  
Crea un nodo de tipo texto con un texto dentro

```
// Selección del contenedor // [HTMLDivElement]
const divNode = document.querySelector(".contenedor");

// Creación de un elemento <p> // [HTMLParagraphElement]
const paragraphNode = document.createElement("P");

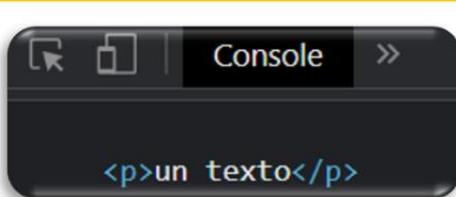
// Creación de un text node // [object text]
const textNode = document.createTextNode("un texto");
```

Para agregar un nodo como hijo a otro nodo, se puede utilizar el siguiente método:

- `appendChild (childNode)`

```
// Creación de elementos
const paragraphNode = document.createElement("P");
const textNode = document.createTextNode("texto");

// Agrega el text node al nodo paragraph <p>
paragraphNode.appendChild(textNode)
```



Por lo tanto, en el ejemplo mostrado, donde se quería añadir un párrafo `<p>` a un contenedor `<div>`, se puede hacer de la siguiente manera:

```
// Selección del contenedor
const divNode = document.querySelector(".contenedor");// [object HTMLDivElement]

// Creación de elementos
const paragraphNode = document.createElement("P");// [HTMLParagraphElement]
const textNode = document.createTextNode("un texto");// [object text]

paragraphNode.appendChild(textNode);// Agrega el text node al nodo paragraph <p>
divNode.appendChild(paragraphNode);// Agrega el nodo paragraph <p> al div del HTML
```

Esto es útil cuando se desea añadir unos pocos elementos ya que, para crear un elemento HTML, el DOM debe borrarse y volverse a escribir en cada proceso de creación. Por dicho motivo, es muy ineficiente cuando se trata de agregar muchos elementos.

Por lo tanto, para agregar varios elementos HTML a la vez, es necesario utilizar el siguiente método:

- `createDocumentFragment()`

```
// Selección del contenedor donde se añadirán elementos
const divNode = document.querySelector(".contenedor");

// Crear el fragment donde se guardarán los elementos
const fragment = document.createDocumentFragment();

for(let i = 0; i < MAX_ELEMENTOS; i++) {

    // Crear párrafos (<p>) incluyéndole un texto con innerHTML
    const paragraphNode = document.createElement("P");
    paragraphNode.innerHTML = `un texto ${i}`;

    // Agregar elemento creado al fragment
    fragment.appendChild(paragraphNode);
}

// Agregar el fragment a nuestro contenedor <div> original
divNode.appendChild(fragment);
```

un texto (0)  
 un texto (1)  
 un texto (2)  
 un texto (3)  
 un texto (4)  
 un texto (5)  
 un texto (6)  
 un texto (7)  
 un texto (8)  
 un texto (9)  
 un texto (10)  
 un texto (11)  
 un texto (12)  
 un texto (13)  
 un texto (14)  
 un texto (15)  
 un texto (16)  
 un texto (17)  
 un texto (18)

## ➤ Child (nodos hijos)

- `appendChild(childNode)` crea un nodo hijo
- `removeChild(childNode)` elimina un nodo hijo
- `replaceChild(new_childNode, old_childNode)`
- `hasChildNodes()` devuelve **true** si tiene nodos hijo

Un h2 común

Un h4 común

Un simple párrafo

<body>

```
<div class="contenedor">
<h2>Un h2 común</h2>
<h4>Un h4 común</h4>
<p>Un simple párrafo</p>
</div>
```

```
<script src="codigo.js"></script>
</body>
```

- `firstChild`

devuelve el primer nodo hijo que encuentre

- `lastChild`

devuelve el último nodo hijo que encuentre

- `firstElementChild`

devuelve el primer element node hijo que encuentre

- `lastElementChild`

devuelve el último element node hijo que encuentre

- `childNodes`

Devuelve un *NodeList* con todos los hijos, a los cuales se puede acceder mediante [ ]

```
// Selección del contenedor <div>
const contenedor = document.querySelector(".contenedor");

const primerHijo = contenedor.firstElementChild; // [object HTMLHeadingElement]
                                                 <h2>
```

→ Recordar que se puede recorrer un NodeList con un **for each**

- `children`

devuelve un *HTMLcollection* con todos los elementos hijos

Recordar que se puede recorrer un HTMLCollection con un **for of**

Propiedades / atributos

Es importante saber que, al haber un “espacio” entre un elemento y otro (salto de línea e indentado), dicho espacio se considera como un nodo de tipo texto. Por lo tanto, el primer/último child que se encuentre en estos casos, será un nodo de tipo texto.

➤ Parent (nodos padres)

- `parentElement` selecciona el padre de tipo elemento (**etiqueta HTML**) de un nodo
- `parentNode` selecciona el nodo padre (incluso si no es un elemento)

```
<body> <!--Cuerpo de la página web -->
  <ui>
    |   <li>item-1</li>
    |   <li>item-2</li>
    |   <li>item-3</li>
  </ui>
```

```
const element = document.querySelector("UI");
element.parentElement;// -> [object HTMLElement]
```

➤ Sibling (nodos hermanos)

- `nextSibling` devuelve el nodo hermano siguiente
- `previousSibling` devuelve el nodo hermano anterior
- `nextElementSibling` devuelve el elemento HTML hermano siguiente
- `previousElementSibling` devuelve el elemento HTML hermano anterior

```
<div class="contenedor">
  <h2>Soy el hermano anterior</h2>
  <h4>Elemento seleccionado</h4>
  <p>Soy el hermano siguiente</p>
</div>
```

```
const element = document.querySelector("H4");
element.nextElementSibling;// -> [object HTMLParagraphElement]
```

➤ Nodos extra

- `closest(selector)` selecciona el elemento de tipo seleccionado ascendente más cercano

## Window

El objeto Window hace referencia a la ventana del navegador. Es el objeto más importante en la jerarquía de JavaScript y todos dependen de él, ya que contiene el objeto *document*, el historial de navegación, entre otros.

El objeto Window hereda las propiedades de *EventTarget*

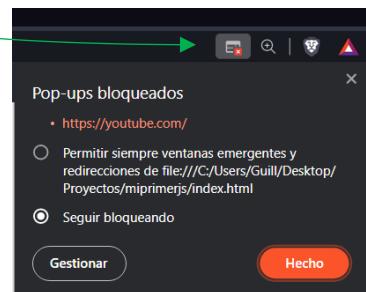
Permite manipular la ventana del navegador, las resoluciones, el scroll realizado en una página, redireccionar a otras páginas web, cerrar la página web actual, permite conocer el host-name, protocolo utilizado y otros datos de la página web.

Algunas de las propiedades y métodos de este objeto son:

- + **open("url")** abre una ventana nueva, la cual puede guardarse en una variable. `window.open("https://youtube.com");`  
Es posible que requiera que desactive el bloqueo de "pop-up" o ventanas emergentes de la página.

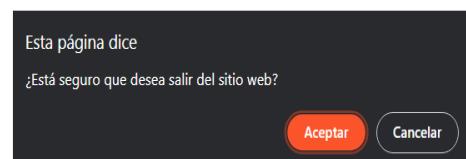
El equivalente en HTML es

```
<meta http-equiv="refresh" content="0;URL=https://youtube.com">
```



- + **close(window)** cierra una ventana abierta  
`window.close(ventana);`
- ❖ **closed** devuelve *true* si la ventana está cerrada
- + **stop()** detiene el cargado de la página
- + **alert("text")** muestra un mensaje emergente
- + **print()** Abre el cuadro de diálogo Imprimir
- + **prompt()** Abre un cuadro de diálogo con un mensaje que solicita un texto (string) al usuario.
- + **confirm()** Abre un cuadro de diálogo con un mensaje y dos botones.

```
confirmacion = window.confirm("¿Está seguro que desea salir del sitio web?");
// Devolverá TRUE si ACEPTA
// Devolverá FALSE si CANCELA
```

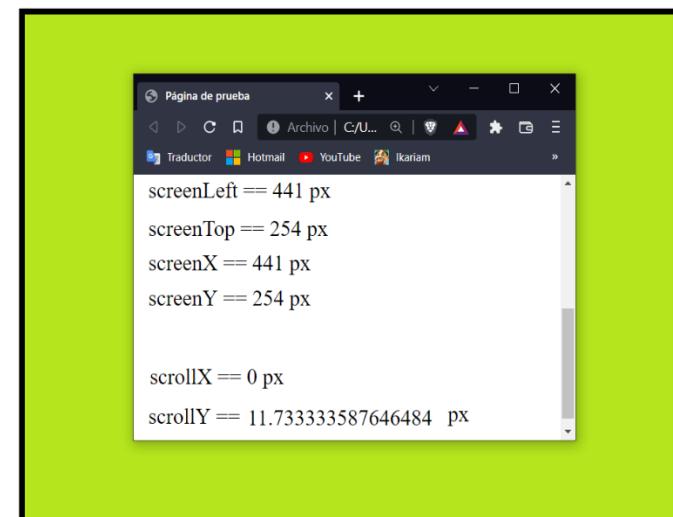


- ❖ **screen**      objeto que representa la pantalla
  - availHeight
  - availWidth

} tamaño de la pantalla
  
- ❖ **screenLeft**      distancia horizontal entre el borde izquierdo del navegador y el borde izquierdo de la pantalla.
  
- ❖ **screenTop**      distancia vertical entre el borde superior del navegador y el borde superior de la pantalla.
  
- ❖ **scrollX**      píxeles que el documento se desplaza actualmente horizontalmente
  
- ❖ **scrollY**      píxeles que el documento se desplaza actualmente verticalmente

```
let screenLeft = window.screenLeft;
let screenTop = window.screenTop;
let screenX = window.screenX;
let screenY = window.screenY;

let scrollX = window.scrollX;
let scrollY = window.scrollY;
```



- + **scroll(x, y)**      scrollea en píxeles.

- + **resizeBy(x, y)**      cambia el tamaño de la ventana actual
- + **resizeTo(x, y)**      redimensiona dinámicamente la ventana
- + **moveBy(x, y)**      mueve la ventana relativamente
- + **moveTo(x, y)**      mueve la ventana absolutamente (px)

Para funcionar, se requiere que la página web esté alojada en un servidor

```
function scrollSlowlyTo(anElement) {
  const scrolled = window.scrollY || window.pageYOffset;
  const posTop = anElement.getBoundingClientRect().y + scrolled;
  window.scrollTo({ top: posTop, behavior: "smooth" });
}
```

## Objetos barprop

- locationbar
- menubar
- personalbar
- scrollbars
- statusbar
- toolbar

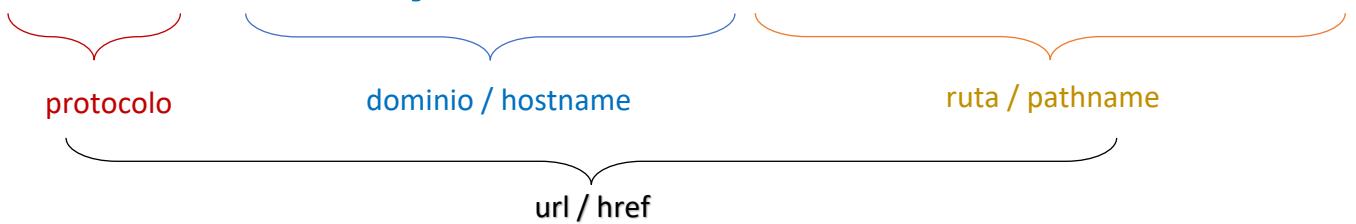
```
let esVisible = window.locationbar.visible;
```

```
locationbar == true
```

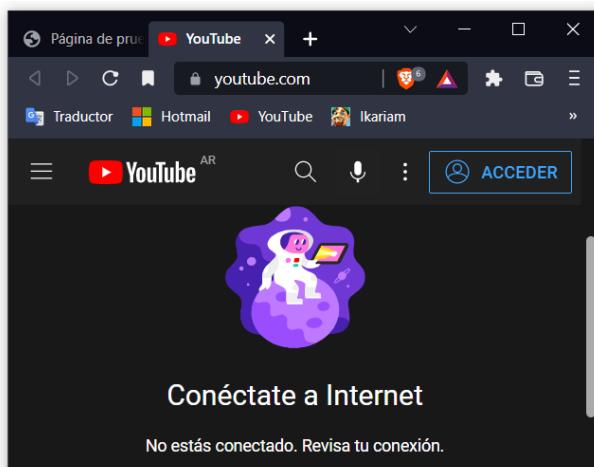
## Objeto Location

Brinda información sobre la locación de la página web.

**HTTPS://www.youtube.com/channel/OD9ASDQCY**



- ❖ **href** devuelve la URL de la página actual o su localización
- ❖ **hostname** devuelve el nombre de dominio del servidor web, incluyendo el subdominio `www`
- ❖ **pathname** devuelve la ruta y el nombre del archivo de la página actual desde el dominio (desde la primera / simple de la URL)
- ❖ **protocol** devuelve el protocolo web utilizado (`HTTP` o `HTTPS`)



```

let url = window.location.href;
let dominio = window.location.hostname;
let path = window.location.pathname;
let protocolo = window.location.protocol;
  
```

```

href === "https://www.youtube.com/"
hostname === "www.youtube.com"
pathname === "/"
protocol === "https:"
  
```

⊕ `assign("url")` carga un nuevo documento.

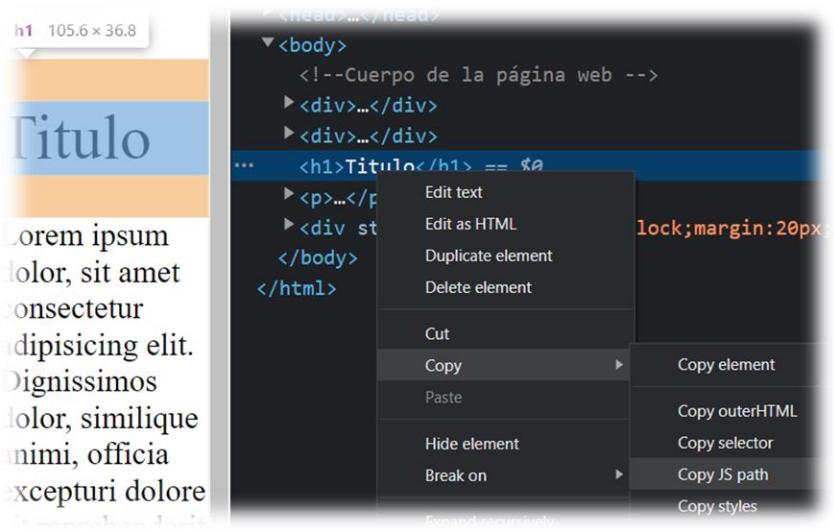
```
window.location.assign("https://youtube.com");
```

## Herramientas de Desarrollo del Navegador

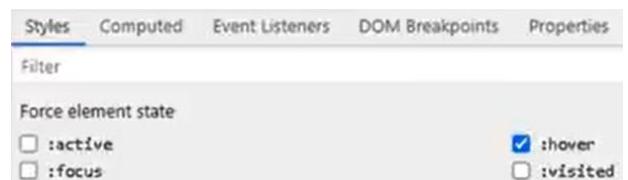
Donde se encuentra la consola del sistema, se pueden utilizar varias herramientas disponibles allí.

Por ejemplo, seleccionar un elemento de la página y copiar el método selector de dicho elemento para JavaScript, para ello sólo se debe seleccionar “Copy JS path”.

También es posible copiar el estilo de CSS del elemento, seleccionando “Copy styles”.



Otra acción que permite, es la de forzar estados como la de :hover



## Eventos

Los eventos son cualquier cambio que ocurre o sucede en la página.

Se considera “punto cero” al momento en que nada de la página está cargado. A partir de ese punto, comienzan a efectuarse muchos cambios (eventos).

### Event handler

Una forma antigua para asociar un evento al código, es utilizando el “event handler”, el cual se realiza con el prefijo **on**. El ejemplo más sencillo es el de aplicar una función a un botón, el cual se puede hacer de la siguiente manera:

```
<button onclick="alert('saludar')>Apretame</button>
```

O bien, utilizando un archivo JavaScript:

```
<button class="button">Apretame</button>
```

```
const button = document.querySelector(".button");

button.onclick = ()=>{
    alert("parlo hola")
}
```

### Event listener

Permite añadir una escucha de eventos a un elemento. Además, permite asociar una función tradicional sin parámetros, o bien definir una función flecha implícita dentro del listener.

```
<button id="btn">Botón</button>
```

```
const button = document.getElementById("btn");

// Función tradicional sin parámetros
button.addEventListener("click", saludar);

function saludar() {
    alert("Hola");
    button.removeEventListener("click", saludar);
}// Se puede eliminar el evento luego de ejecutarlo
```

```
const button = document.getElementById("btn");

// Función flecha implícita
button.addEventListener("click", () => {
    alert("Hola");
});
```

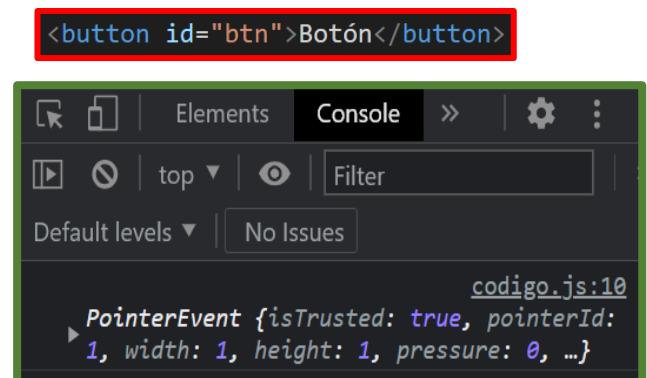
Es importante saber que, dentro de la función flecha implícita, se está trabajando dentro del entorno del objeto `window`. Es decir, no se pueden acceder a las variables declaradas por fuera.

## Objeto Event

Recordando los event listener y su aplicación con funciones flecha, dichas funciones permiten el uso de un solo parámetro, el cual será el objeto event, comúnmente representado con la letra `e`.

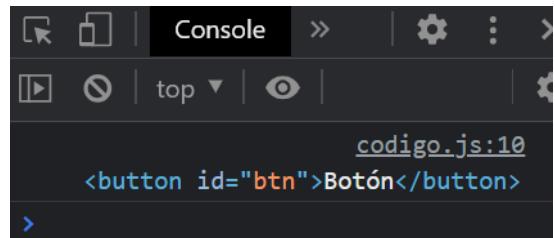
```
const button = document.getElementById("btn");

// Función flecha implícita
button.addEventListener("click", e => {
  console.log(e);
});
```



Para acceder al elemento que desencadenó el evento, se utiliza la propiedad `target` del event object.

```
// Función flecha implícita
button.addEventListener("click", e => {
  console.log(e.target);
});
```



Es posible detectar que la página dejó de estar visible por el usuario con el siguiente código

```
document.addEventListener("visibilitychange", function() {
  if (document.visibilityState === "hidden") {
    console.log("La página se ha minimizado");
  } else if (document.visibilityState === "visible") {
    console.log("La página está nuevamente visible");
  }
});
```

## Event flow

Orden en que se ejecutan los eventos.

- **Event Bubbling**

Es la ejecución por defecto, en la cual se ejecutan primero los elementos más específicos (más hijos), y luego los menos específicos (más contenedores)

Es decir, se ejecuta desde el más hijo al más padre (contenedor)

```
<div class="contenedor">
|   <button id="btn">Botón</button>
</div>
```

```
const button = document.getElementById("btn");
const contenedor = document.querySelector(".contenedor");

contenedor.addEventListener("click", e => {
    alert("Yo me ejecuto segundo porque soy contenedor");
});

button.addEventListener("click", e => {
    alert("Yo me ejecuto primero porque soy hijo");
});
```

- **Event Capturing**

Se ejecutará primero el que tenga el parámetro true. Si hay más contenedores padres que contengan el parámetro true, serán ejecutados en orden de aparición en el código.

```
const button = document.getElementById("btn");
const contenedor = document.querySelector(".contenedor");

contenedor.addEventListener("click", e => {
    alert("Yo me ejecuto primero porque tengo TRUE");
}, true);

button.addEventListener("click", e => {
    alert(`Yo me ejecuto justo después de los que
          tienen TRUE porque soy el más hijo`);
});
```

Para detener una propagación de eventos, se puede utilizar el método `stopPropagation()`

```
const button = document.getElementById("btn");
const contenedor = document.querySelector(".contenedor");

contenedor.addEventListener("click", e => {
    alert(`Yo me ejecuto primero porque tengo TRUE,
          y detengo la propagación de eventos porque
          tengo el método que los detiene`);

    e.stopPropagation();
}, true);

button.addEventListener("click", e => {
    alert(`Si los eventos empiezan en contenedor,
          yo no me ejecuto.`);
});
```

## Eventos del mouse

- **click** se ejecuta al hacer click sobre un elemento (y soltarlo)
- **dblclick** se ejecuta si se hacen 2 click en menos de 0,5 segundos
- **mouseover** cuando el puntero entra en un elemento (o uno de sus hijos)
- **mouseout** cuando el puntero sale de un elemento (o de sus secundarios)
- **contextmenu** se ejecuta al hacer click en el botón derecho en un elemento
- **mouseleave** cuando el puntero se mueve fuera de un elemento
- **mousedown** cuando se aprieta un botón del mouse sobre un elemento
- **mouseup** cuando un usuario suelta un botón del mouse sobre un elemento
- **mousemove** cuando el puntero se mueve mientras está sobre un elemento
- **mouseenter** cuando el puntero se mueve sobre un elemento  
Sólo aplicable en internet explorer

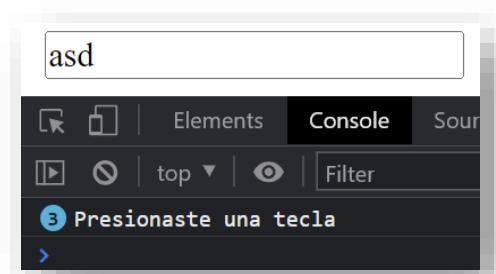
## Eventos de teclado

- **keydown** cuando una tecla se presiona
- **keypress** cuando una tecla se presiona y se suelta en un mismo elemento
- **keyup** cuando una tecla se deja de presionar

```
<input id="input" type="text" name="nombre">
```

```
const input = document.getElementById("input");

input.addEventListener("keydown", e => {
  const tecla = e.key;
  console.log(`Presionaste la tecla ${tecla}`);
});
```



## Eventos de la interfaz

- **abort** cuando un elemento madre elimina a su hijo.
- **error** cuando sucede un error durante la carga de un archivo multimedia
 

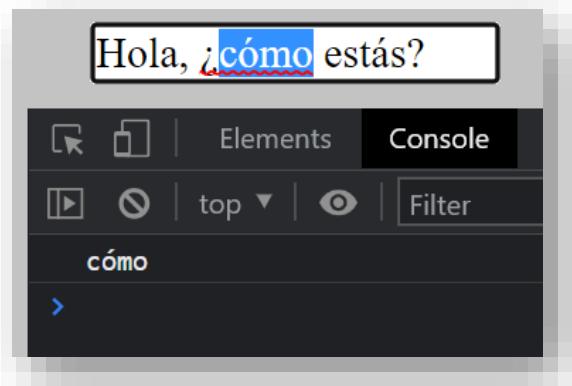
```
const image = document.getElementById("img-test");
image.addEventListener("error", e => {
  console.log("No se pudo cargar la imagen");
});
```
- **load** cuando un objeto se ha cargado (incluso el window)
 

```
window.addEventListener("load", e => {
  console.log("El sitio está cargado");
})
```
- **beforeunload** antes de que la página sea cerrada
 

Se aplica sobre el  
objeto window
- **unload** luego de que la página sea cerrada
- **resize** se ejecuta cuando se cambia el tamaño de la página
- **scroll** se ejecuta cuando se scrollea la página (o un elemento)
- **select** se ejecuta cuando se selecciona algún texto de un **<input>** o un **<textarea>**. Este evento puede guardar información sobre el texto seleccionado dentro de la propiedad **target**.

```
<input id="input" type="text" name="nombre">
```

```
Const input = document.getElementById("input");
input.addEventListener("select", e => {
  const textoCompleto = input.value;
  const start = e.target.selectionStart;
  const end = e.target.selectionEnd;
  const textoSeleccionado = textoCompleto.substring(start, end);
  console.log(textoSeleccionado);
});
```



# Formulario

```
<section class="form-container">
  <div id="form">
    <fieldset>
      <legend>Form title</legend>
      <div class="form-container__inputcontainer">
        <div class="form-container__inputcontainer-inputfield grid-simple">
          <label>Name</label>
          <input class="input" id="input-name" type="text" placeholder="A name" name="name" required="">
        </div>
        <div class="form-container__inputcontainer-inputfield grid-simple">
          <label>Amount</label>
          <input class="input" id="input-amount" type="number" placeholder="An amount" name="amount" required="" min=0>
        </div>
        <div class="form-container__inputcontainer-inputfield grid-double">
          <label>Description</label>
          <textarea class="input" id="input-text" type="text" placeholder="A text" name="textarea" required=""></textarea>
        </div>
      </div>
      <div class="form-container__warningcontainer hidden">
        <p class="warning">* You must complete all fields</p>
      </div>
      <div class="form-container__submitcontainer">
        <button id="submit">Send</button>
      </div>
    </fieldset>
  </div>
</section>
```

**FORM TITLE**

Name	Amount
A name	An amount

**Description**

A text

Send

```
:root {
  --form-background-color: rgb(168, 67, 39);
  --warning-color: rgb(167, 183, 252);
  --form-width: 85vw;
  --form-legend-size: 5vw;
  --form-input-size: 2.5vw;
}

.form-container #form {
  background-color: var(--form-background-color);
  width: var(--form-width);
  margin: 0 auto; /* centrado */
  padding: 1vw;
  border-radius: 1rem;
}

.form-container a:link , .form-container a:visited {
  text-decoration: none;
}

.form-container textarea { resize: none; }

.form-container fieldset {
  display: flex;
  flex-direction: column;
  border: none;
}

.form-container legend {
  text-transform: uppercase;
  font-weight: bold; /* negrita */
  text-align: center;
  font-size: var(--form-legend-size);
  margin-bottom: 4vh;
}

.form-container__inputcontainer {
  display: grid;
  align-items: center;
  place-items: center;
  width: 100%;
  grid-template-columns: repeat(2, calc(var(--form-width) * .42));
  grid-gap: 3vh 3vw;
  font-size: var(--form-input-size);
}

@media (min-width: 768px) { /* PC */
  .form-container__inputcontainer {
    grid-template-columns: repeat(2, calc(var(--form-width) * .45));
    font-size: calc(var(--form-input-size) * .8 );
  }
}
```

```
.form-container__inputcontainer-inputfield {
  display: flex;
  flex-direction: column;
  width: 100%;
}

.form-container__inputcontainer-inputfield label {
  font-family: "Comic Sans MS", "Helvetica Neue", Helvetica, Arial;
  font-size: calc(var(--form-input-size) * 1.4);
  font-weight: bold; /* negrita */
  color: white;
  margin: 5px;
}

.form-container__inputcontainer-inputfield .input {
  height: calc(var(--form-input-size) * 2);
  border-radius: 8px;
  padding: 6px;
}

.form-container__inputcontainer-inputfield textarea {
  height: calc(var(--form-input-size) * 6.5) !important;
}

.grid-double { grid-column: 1 / span 2; }
.grid-simple { grid-column: 1 / span 0; }
.hidden { display: none !important; }
.warning {
  font-weight: bold; /* negrita */
  color: var(--warning-color);
  margin-top: .7rem;
}

.form-container__submitcontainer {
  display: grid;
  place-items: center;
  margin-top: 3vh;
  font-size: 24px;
}

.form-container__submitcontainer button {
  padding: 4px 12px;
  font-size: 3vw;
  background-color: bisque;
  border-radius: 10%;
  transition: transform 0.15s;
}

.form-container__submitcontainer button:hover {
  transform: scale(1.1);
}
```

Los formularios se pueden trabajar de dos maneras con JavaScript.

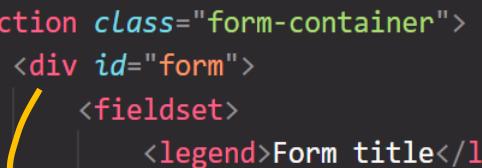
1. Hacer todo desde JavaScript mediante la detección del botón del formulario y ejecutar un bloque de código.

```
Window.onload = () => {
    const bSubmit = document.getElementById("submit");
    const inputName = document.getElementById("input-name");
    const inputAmount = document.getElementById("input-amount");
    const inputTextArea = document.getElementById("input-text");
    const warnDiv = document.querySelector(".form-container__warningcontainer");

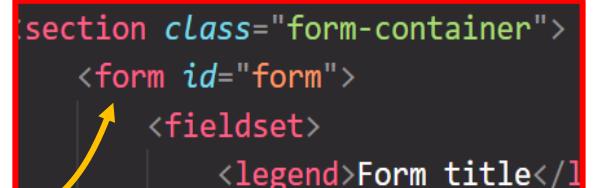
    bSubmit.addEventListener("click", () => {
        if (inputName.value.length > 0
            && inputAmount.value.length > 0
            && inputTextArea.value.length > 0) {
            warnDiv.classList.toggle("hide", true); // Oculta el aviso
            // CODE

        } else warnDiv.classList.remove("hide"); // Muestra el aviso
    });
}
```

2. Trabajar con GET y POST, para lo cual es necesario utilizar la etiqueta **form**

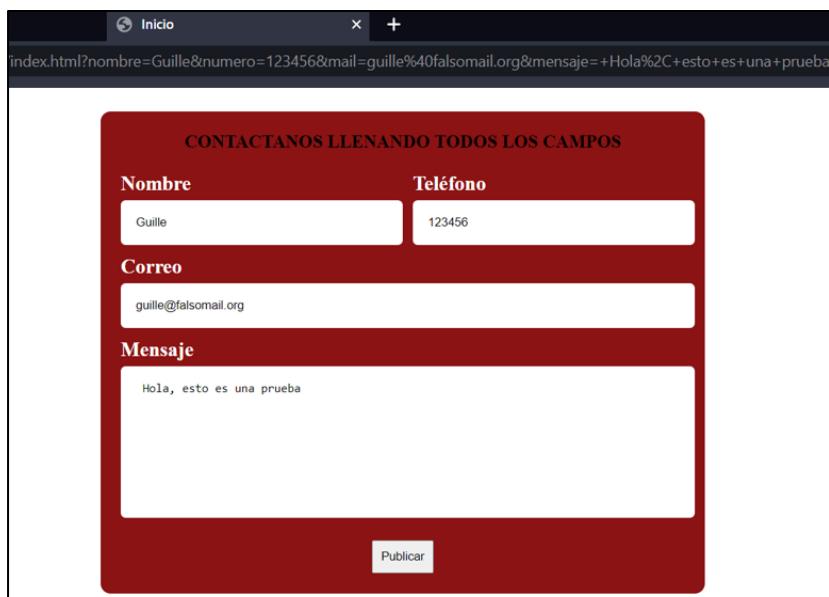


```
<section class="form-container">
    <div id="form">
        <fieldset>
            <legend>Form title</legend>
```



```
<section class="form-container">
    <form id="form">
        <fieldset>
            <legend>Form title</legend>
```

De esta forma, al tocar el botón de submit se enviará como parámetros a la misma página el contenido de los **input**.



Se puede establecer a qué página se redirigirá tras presionar el botón:

```
<section class="form-container">
  <form id="form" method="get" action="anotherpage.html">
    <fieldset>
      <legend>Form title</legend>
```

Con el método GET, se pueden obtener los parámetros recibidos desde la URL

```
const params = new Proxy(new URLSearchParams(window.location.search), {
  get: (searchParams, prop) => searchParams.get(prop)
});

const paramName = params.name;
const paramAmount = params.amount;
const paramText = params.textarea;
```

Por otro lado, se puede obtener el momento en que un **input** cambia con el evento "change".

```
anInput.addEventListener("change", () => console.log("Cambio"));
```

## Temporizadores (timers)

Permite la ejecución de una función en un determinado tiempo o cada cierto tiempo establecido.

Estos métodos **consumen demasiados recursos**, por lo que no es recomendable utilizarlo indiscriminadamente, sino para lo que es realmente necesario y que no se encuentre otra alternativa.

- `setTimeout(function, tiempo_en_ms)` ejecuta una función en un tiempo, una sola vez
- `setInterval(function, tiempo_en_ms)` ejecuta una función cada cierto tiempo
- `clearTimeout(timer)` detiene la ejecución del timeout
- `clearInterval(interval)` detiene la ejecución del Interval

```
// Se ejecuta una sola vez en 2 seg
const timer = setTimeout(() => {
  console.log("Hola");
}, 2000);

// Eliminar el timer
clearTimeout(timer);
```

```
// Se ejecuta cada 2 segundos
const interval = setInterval(() => {
  document.write("Hola");
}, 2000);

// Eliminar el interval
clearInterval(interval);
```

## Código obsoleto

Mantener código obsoleto en nuestro programa puede provocar problemas de rendimiento por uso excesivo de recursos e, incluso, bajar el nivel de posicionamiento ante los motores de búsqueda (SEO) como Google.

Un código obsoleto (*deprecated*) se define como no útil para las circunstancias actuales debido a que puede ser ineficiente ante las nuevas alternativas existentes, presenten bugs o fallos, sean inseguros, o ya no existan.

Estadísticamente se sabe que 1 de cada 3 páginas web utilizan librerías de JavaScript obsoletas.

Se puede verificar cuáles son los métodos de JavaScript obsoletos en los sitios basados en estándares oficiales. En estas páginas se suelen usar íconos que representan si un método está obsoleto o no, o bien lo indican con la palabra “*deprecated*”.

Una página recomendada para esto es: <https://developer.mozilla.org/>



Experimental

puede sufrir cambios en el futuro.



No estandarizado

se debe verificar que funcione.



Deprecated

el método ya no funciona.



No recomendado

probablemente no funcione o deje de funcionar pronto.

Es importante que se verifique el navegador que esté utilizando el usuario (con el objeto *navigator*) para poder implementar los métodos más adecuados para ellos.

## Callbacks

Se denomina así a las funciones que llaman a otras funciones, las cuales son pasadas como parámetro a otra función.

```
let myName = "Nombre"

function fCallback(callback) {
    callback(myName);
}

function decirNombre(nombre) {
    console.log(nombre);
}

fCallback(decirNombre);
```

Llama a esta función, con la función *decirNombre* como parámetro. Por lo tanto, es como si `callback` fuera `decirNombre`, es decir, se comporta como tal

Una forma abreviada de escribir este proceso es escribiendo la función implícitamente, de la siguiente manera:

```
let myName = "Nombre"

function fCallback(callback) {
    callback(myName);
}

fCallback(function decirNombre(nombre) {
    console.log(nombre);
});
```

Se puede abreviar aún más si se utiliza una función flecha:

```
let myName = "Nombre"

function fCallback(callback) {
    callback(myName);
}

fCallback(nombre => {
    console.log(nombre);
});
```

Teniendo un array de objetos, se puede obtener una de sus propiedades de la siguiente manera:

```
class Persona {
  constructor(nombre, instagram) {
    this.nombre = nombre;
    this.instagram = instagram;
  }
};
```

```
let personas = [
  new Persona("Guillermo", "@guille.hern"),
  new Persona("Sol Pereira", "@solcha"),
  new Persona("Pablo Benitez", "@kantro")
];
```

Con esta base, podemos crear un callback que permita obtener el nombre de una persona del array personas.

```
const obtenerNombre = (id, cb) => {
  if(personas[id] === undefined) cb("No se ha encontrado a la persona");
  else cb(null, personas[id].nombre);
}

obtenerNombre(2, (err, nombre)=>{
  if(err) console.log(err);
  else console.log(nombre);
});
```

Con este código, se obtendrá el nombre de la persona con el index 2 (Pablo Benitez)

En caso de ingresar un index inválido, devolverá el mensaje “no se ha encontrado...”.

Este método tiene como desventaja el hecho de que debe comprobarse cada atributo de la clase u objeto que se trabaja en un nuevo callback para verificar si existe o no. Esto puede ser problemático cuando existen muchos atributos y algunos de ellos sean opcionales de colocar.

## Promise

Al trabajar con “promesas”, se puede evitar la desventaja que teníamos con los callback al trabajar con objetos de múltiples propiedades/atributos de los cuales no sabemos si están definidos o no.

Se trata de un objeto, el cual tiene como parámetro una función con 2 callbacks (resolve y reject).

```
const promesa = new Promise((resolve, reject) => {
  |  (condicion) ? resolve("Se cumple") : reject("No se cumplió");
});
```

Para acceder al contenido de `resolve` (el cual es lanzado cuando la condición dentro de la promesa se cumple, es decir, es `true`), se utiliza el método `then()`, el cual recibe como parámetro un `callback`. Además, agregando el `.catch()` (como manejo de errores) se puede acceder al contenido de `reject`.

The diagram illustrates the execution flow of a Promise-based script. On the left, a yellow-bordered code block contains:

```
promesa.then((resultado)=>{// resolve
  |  console.log(resultado);
})
.catch((err) => {// reject
  |  console.log(err);
})
```

Two arrows point from this code block to two separate browser developer tool consoles on the right. The top console, associated with the green arrow, shows the output "Se cumple". The bottom console, associated with the red arrow, shows the output "No se cumplió".

Volviendo al ejemplo presentado en los callback:

```
class Persona {
    constructor(nombre, instagram) {
        this.nombre = nombre;
        this.instagram = instagram;
    }
};
```

```
let personas = [
    new Persona("Guillermo", "@guille.hern"),
    new Persona("Sol Pereira", "@solcha"),
    new Persona("Pablo Benitez", "@kanthro")
];
```

Se puede trabajar de la siguiente manera:

### 1. Definir las funciones promesa

```
const obtenerPersona = id => {
    return new Promise((resolve, reject) => {
        if(personas[id] === undefined) reject("No se ha encontrado la persona");
        else resolve(personas[id]);
    });
}

const obtenerInstagram = id => {
    return new Promise((resolve, reject) => {
        if(personas[id].instagram === undefined) reject("No hay instagram");
        else resolve(personas[id].instagram);
    });
}
```

### 2. Ejecutar las funciones

```
let id = 2;

obtenerPersona(id).then((persona)=>{
    console.log(persona.nombre);
    return obtenerInstagram(id);
}).then((instagram)=>{
    console.log(instagram);
}).catch((reject)=>{
    console.warn(reject);
});
```

En este caso, si alguna persona no tuviera Instagram definido, se ejecutaría el *catch* con el mensaje establecido en caso de *reject*.

## Funciones asíncronas

Trabajan en tiempo real, es decir, esperan a obtener la información para continuar. Esto sucede, por ejemplo, cuando se intenta obtener información de un servidor o de una función que puede demorar un tiempo en terminar de ejecutarse.

Las funciones asíncronas trabajan con **promise**.

Se utiliza el operador **async** para establecer a una función como asíncrona.

```
const showResult = async() => {  
}
```

```
async function showResult() {  
}
```

El operador **await** sirve para indicarle a una función asíncrona que debe esperar que una función finalice para continuar.

Por ejemplo, una función que demora 3 segundos en ejecutarse:

Se puede utilizar dicho operador de la siguiente forma:

```
const objeto = {  
    propiedad1 : "valor1",  
    propiedad2 : "valor2",  
    propiedad3 : "valor3"  
};  
  
const obtenerInformacion = ()=>{  
    return new Promise((resolve,reject)=>{  
        setTimeout(()=> {resolve(objeto)},3000)  
    })  
}
```

```
const mostrarResultado = async() => {  
    const resultado = await obtenerInformacion();  
    console.log(resultado);  
}
```

De esta forma, se espera a que la función *obtenerInformación* finalice para continuar con el resto de la función.

## Promise como función asíncrona vs operador await en función asíncrona

Las “promesas” funcionan como funciones asíncronas, por lo tanto, pueden utilizarse como tales.

```
// Simulación de lo que demora un servidor en devolver un dato
const devolverTextoEnTiempoAleatorio = (texto) => {
    return new Promise((resolve, reject) => {
        setTimeout(() => { resolve(texto); }, Math.random()*500);
    });
}
```

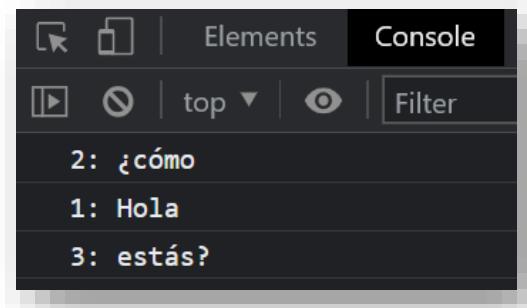
Si se quieren obtener distintas informaciones de un servidor, o una función que demora en terminar de ejecutarse un tiempo desconocido, se pueden utilizar funciones asíncronas o promesas, pero con una diferencia entre cada una:

- **Promise:** no se obtienen los distintos datos en el orden en que fueron solicitados.

```
// Primer llamado
devolverTextoEnTiempoAleatorio("1: Hola")
.then(resultado => console.log(resultado));

// Segundo llamado
devolverTextoEnTiempoAleatorio("2: ¿cómo")
.then(resultado => console.log(resultado));

// Tercer llamado
devolverTextoEnTiempoAleatorio("3: estás?")
.then(resultado => console.log(resultado));
```

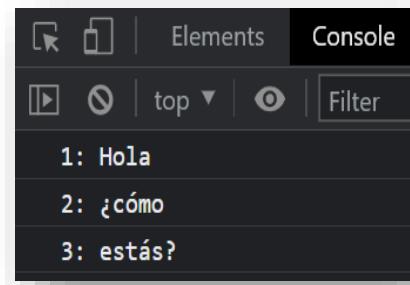


- **Función asíncrona con await:** las funciones se mostrarán siempre en orden ya que el *await* esperará a que la función termine para continuar con el resto del código.

```
// Creando la función asíncrona
const mostrarData = async() => {
    let data1 = await devolverTextoEnTiempoAleatorio("1: Hola");
    let data2 = await devolverTextoEnTiempoAleatorio("2: ¿cómo");
    let data3 = await devolverTextoEnTiempoAleatorio("3: estás?");

    console.log(data1);
    console.log(data2);
    console.log(data3);
}

// Llamado a la función asíncrona
mostrarData();
```



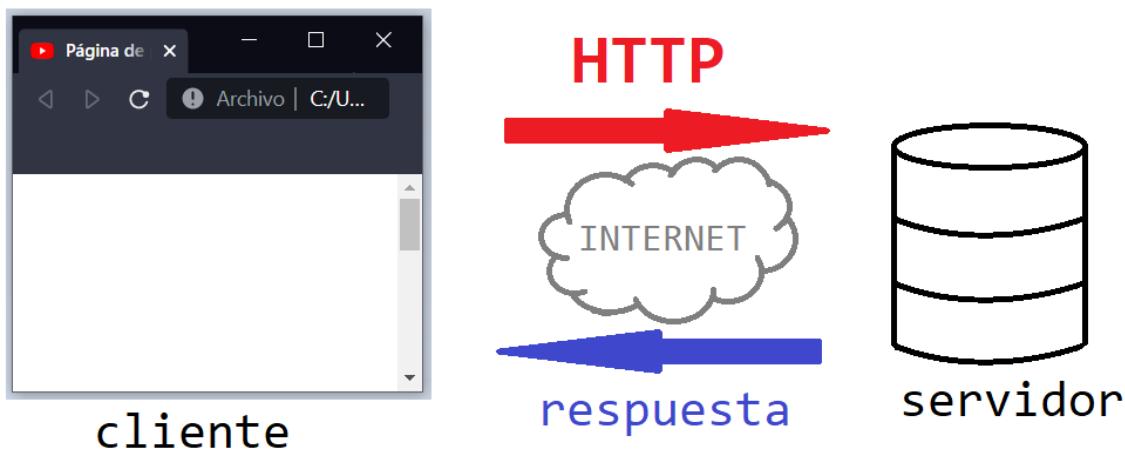
Ejemplo

## Peticiones HTTP

El protocolo HTTP es una petición que realiza el usuario / cliente hacia un servidor, y dicho servidor le devuelve una data (información).

Esto es unidireccional, es decir, sólo se puede obtener información y no guardarla dentro de un servidor (no mediante HTTP).

- Cliente: todo a lo que el usuario puede acceder desde su interfaz (navegador).
- Servidor: donde se procesa toda la información.

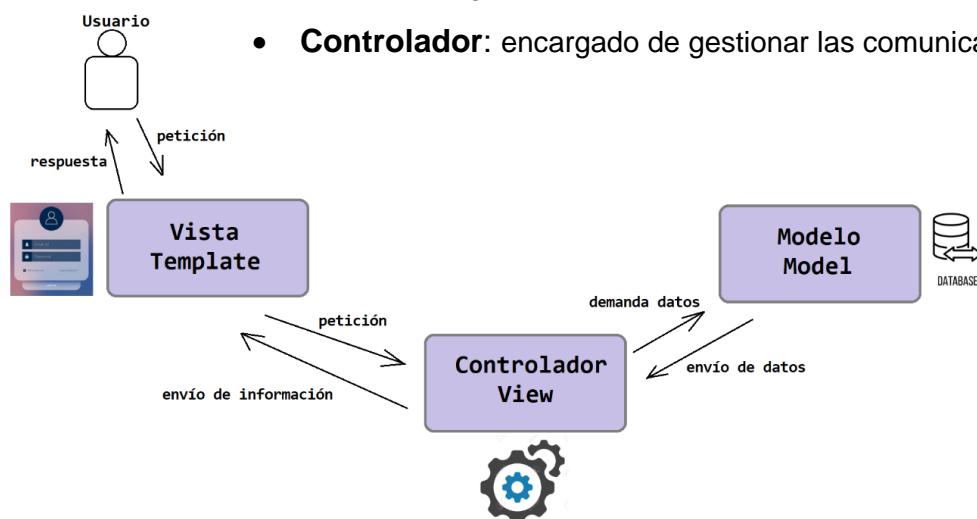


Cada vez que el servidor envía una respuesta al cliente, se actualiza la página web activa para poder visualizarse.

## Modelo Vista Controlador

Patrón que consiste en dividir cualquier aplicación en tres grandes módulos.

- **Modelo:** encargado de gestionar los datos, en general con una Base de Datos.
- **Vista:** encargado de mostrar la información al usuario, la interfaz gráfica.
- **Controlador:** encargado de gestionar las comunicaciones entre la vista y el modelo.



## Datos Estructurados (JSON)

Son similares a los array asociativos (arreglos con los nombres de posición personalizado). La diferencia que se encuentra en un JSON es que las posiciones se guardan con comillas y no con nombre de variable.

```
let JSON = // object
    "nombre" : "Guille",
    "apellido" : "Hernandez"
}
console.log(JSON.nombre);
```

```
let arrayAsociativo = {
    nombre : "Guillermo",
    apellido : "Hernandez"
}
console.log(arrayAsociativo.nombre);
```

En ambos casos, se puede acceder al dato por medio de un punto luego del nombre de la variable (objeto) donde se encuentra almacenado.

### Serialización

Para poder enviar y recibir una estructura de datos en formato JSON, es necesario que se encuentre serializada.

Se considera que un JSON está serializado cuando es una cadena de texto (string) sin espacios intermedios.

```
const JSONserializado = '{"nombre" : "Guillermo", "apellido" : "Hernandez"};
```

El proceso de deserialización consta, simplemente, de quitar las comillas que convierten toda la variable en una cadena de texto.

```
const JSONdeserializado = {"nombre" : "Guillermo", "apellido" : "Hernandez"};
```

### Métodos de serialización y deserialización

- Serializar: **JSON.stringify** convierte un objeto JSON en un string

```
const JSONdesherializado = {"nombre" : "GuillePC", "apellido" : "Intel Core I7"};
const JSONserializado = JSON.stringify(JSONdesherializado);
```

- Deserializar: **JSON.parse** convierte un string en un objeto JSON

```
const JSONserializado = '{"nombre" : "Guillermo", "apellido" : "Hernandez"}';
const JSONdesherializado = JSON.parse(JSONserializado);
```

## Iteración de un objeto JSON

Un objeto JSON no se puede recorrer de forma normal

```
for (let index = 0, index < jsonObject.length; index++) {
    const value = jsonObject[index]; // do nothing
    print(value);
}
// jsonObject.length is undefined
```

```
for (const index of jsonObject) {
    // ERROR: jsonObject is not iterable
}
```

```
for each
jsonObject.forEach(element => {
    print(element);
}); // ERROR. Is not a function
```

Será necesario usar un **for in**

**for in**

```
for (const key in jsonObject) {
    const value = jsonObject[key];
    print(value);
}
```

O bien, ayudarse del objeto Object

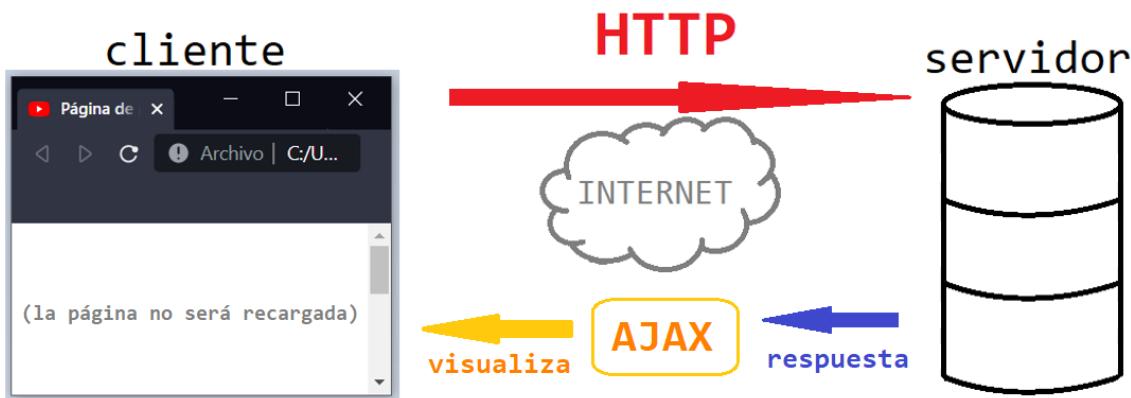
```
const keys = Object.keys(jsonObject);
keys.forEach(key => {
    const value = jsonObject[key];
    print(value);
})
```

```
const values = Object.values(jsonObject);
values.forEach(value => { // no key
    print(value);
})
```

```
Object.entries(jsonObject).forEach(([key, value]) => {
    print(key + ": " + value);
});
```

## Asynchronous JavaScript And XML (AJAX)

Permite obtener información de un servidor tras una petición HTTP sin necesidad de volver a cargar la página web para poder ser visualizado, ya que hace que las peticiones se ejecuten en segundo plano.



Para trabajar con AJAX es necesario tener un servidor, esto es debido a protocolos de seguridad establecidos.

Un buen servidor que se puede utilizar es XAMPP.

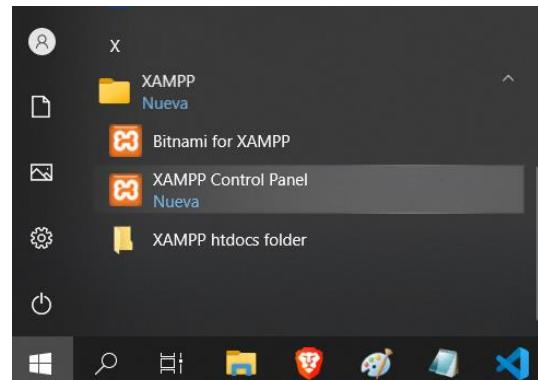
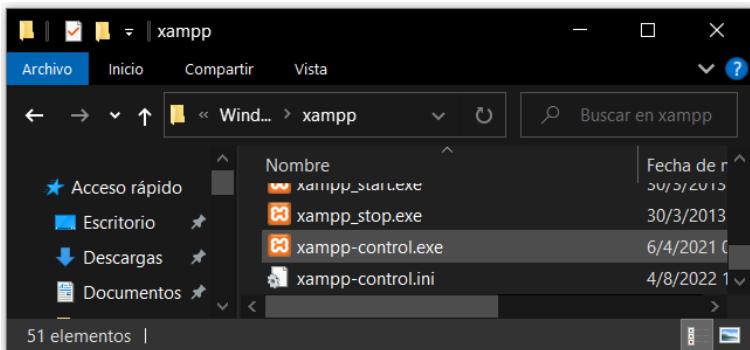
**Es necesario saber que AJAX no es soportado por todos los navegadores.**

## Servidor XAMPP

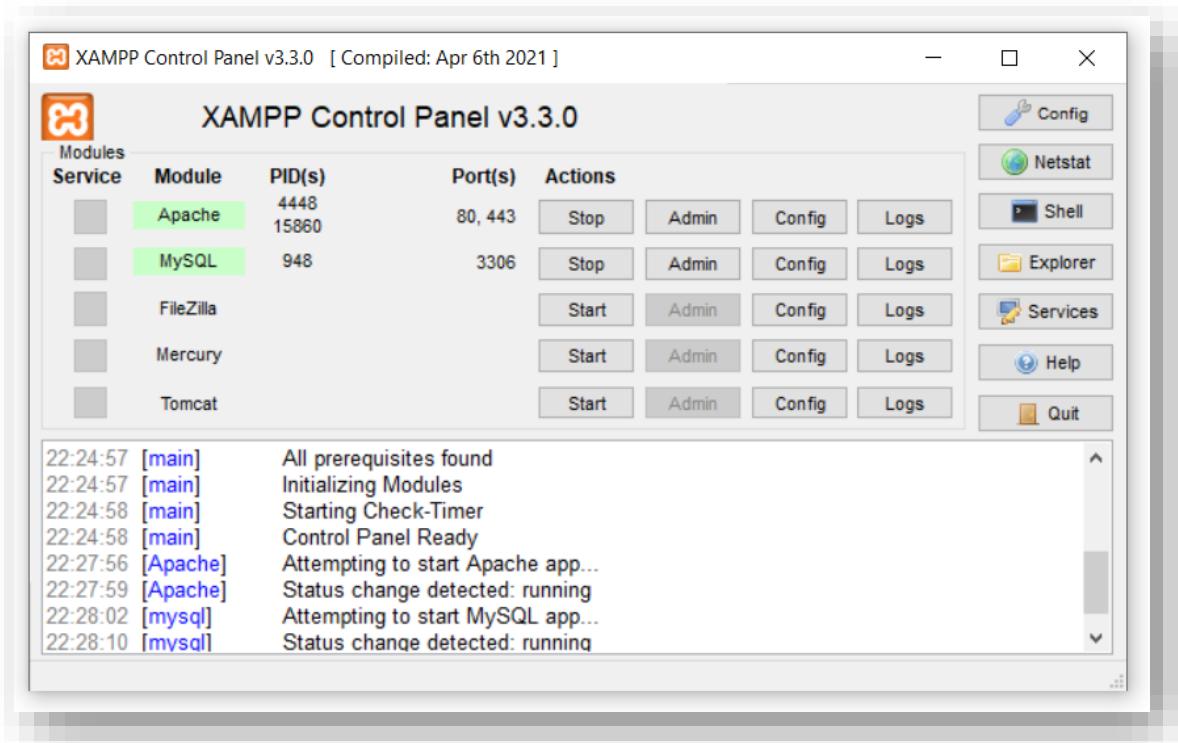
El servidor XAMPP permite alojar un servidor tanto en Windows como en Linux.

Una página segura de donde descargar es: <https://www.apachefriends.org/es/download.html>

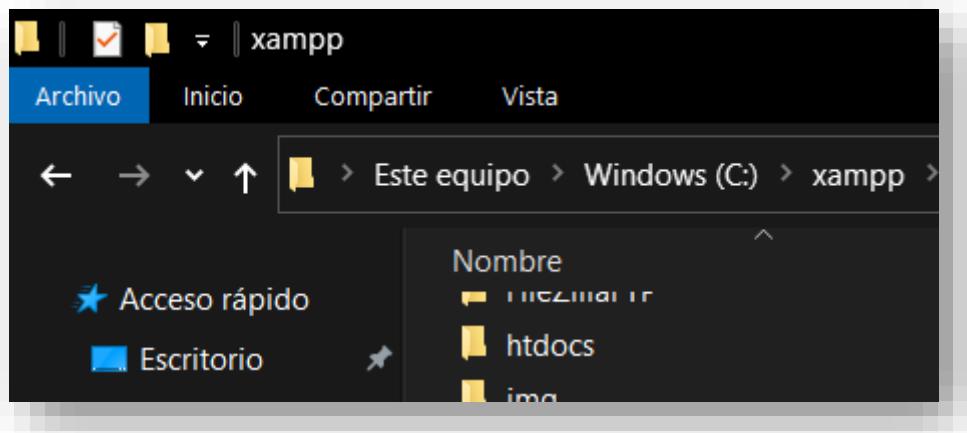
Una vez instalado, se debe ejecutar el Panel de Control de XAMPP. Se puede encontrar en el menú inicio o en el directorio donde se instaló el programa.



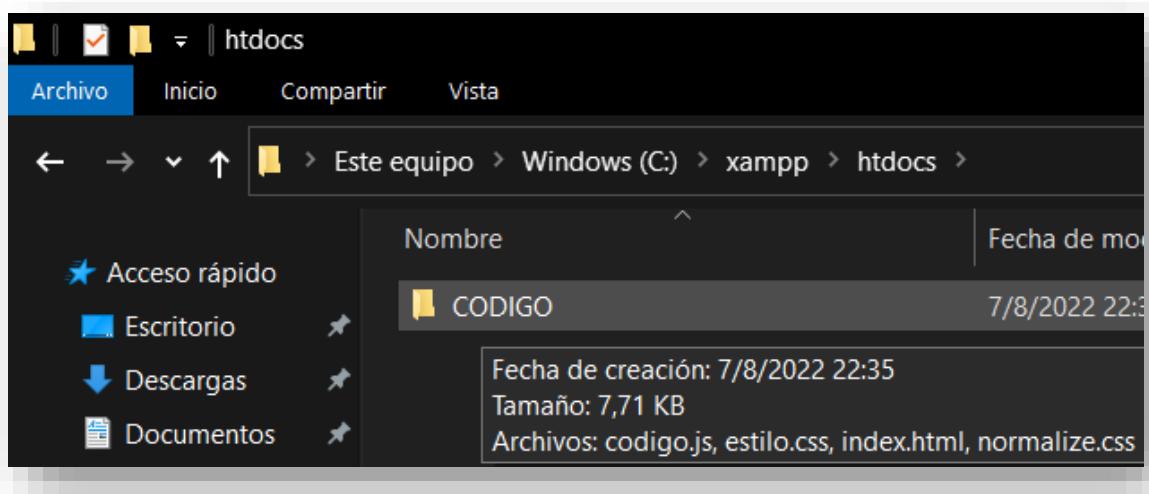
Luego, inicializar (start) tanto **Apache** como **MySQL** (los dos primeros).



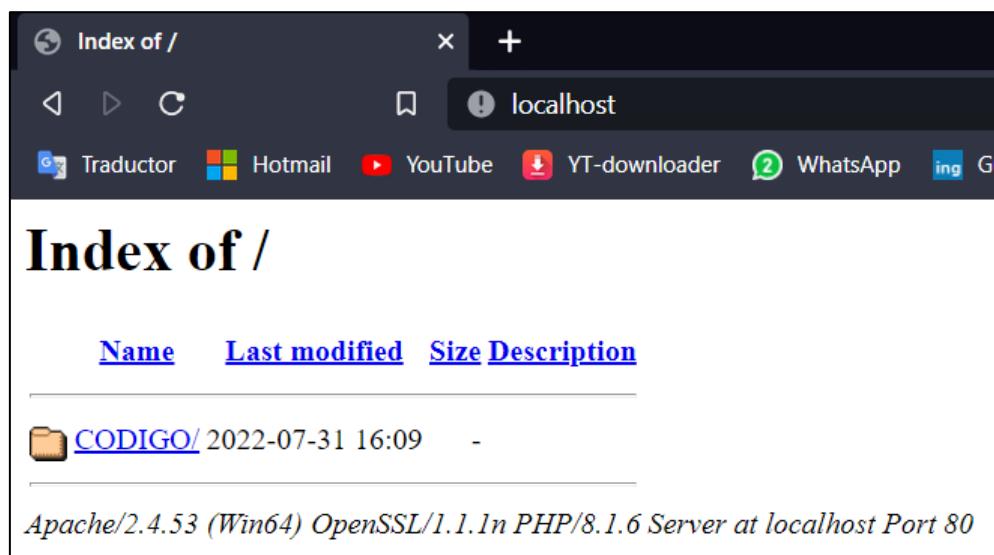
Luego, ingresar al directorio donde se instaló XAMPP e ingresar a la carpeta **htdocs**:



Se debe eliminar todo el contenido de la carpeta, y crear una nueva carpeta con el nombre donde se almacenará nuestro código.



Se puede acceder al servidor creado mediante la dirección **localhost**.



**Index of /**

Name	Last modified	Size	Description
<a href="#">CODIGO/</a>	2022-07-31 16:09	-	

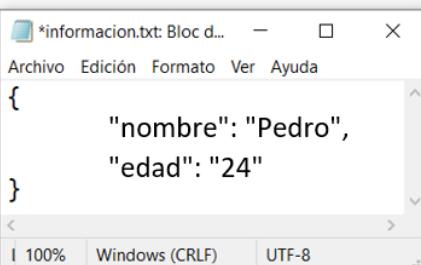
Apache/2.4.53 (Win64) OpenSSL/1.1.1n PHP/8.1.6 Server at localhost Port 80

## Objeto XMLHttpRequest

```
const request = new XMLHttpRequest();
```

Permite realizar peticiones (request).

A modo instructivo, se trabajará con un archivo de texto local que contendrá información en formato JSON serializado.



```
{ "nombre": "Pedro", "edad": "24"}
```

Las iniciales deben estar en minúscula



- Petición de tipo GET

```
const URL = "informacion.txt"; // archivo local
const request = new XMLHttpRequest();

request.open("GET", URL); // inicializa la petición
request.send(); // envía la petición
```

Se podrá visualizar el contenido en la propiedad **response** y **responseText**, sin embargo, no se lo puede trabajar a menos que el código de respuesta sea **3 o 4**, y cuando el **status** sea **200**.

```
readyState: 4
status: 200
response: "{\r\n\t\"Nombre\": \"Pedro\", \r\n\t\"Edad\""
responseText: "{\r\n\t\"Nombre\": \"Pedro\", \r\n\t\"Edad\""
responseURL: "http://localhost/CODIGO/informacion.txt"
```

La propiedad **readyState** tiene cuatro estados que se ejecutan en orden:

1. La solicitud se creó correctamente.
2. La solicitud se envió correctamente.
3. La petición se está procesando.
4. La petición fue terminada (no necesariamente con éxito).

La propiedad **status** indica el estado de la petición, en un código numérico:

404 (Not found)	La URL del request no pudo encontrarse.
200 (Success)	La petición fue realizada con éxito.

en desuso

```
peticion.addEventListener("readystatechange",()=>{
  if (peticion.readyState == 4 && petiction.status == 200
    console.log(peticion.response)
  }
})
```

Conocido todo lo anterior sobre las peticiones, se puede utilizar el evento “load” (el cual se ejecuta únicamente cuando la petición fue completada) para realizar una petición de forma muy básica de tipo *GET*.

```
const URL = "informacion.txt"; // archivo local
const request = new XMLHttpRequest();

request.addEventListener("load", ()=>{
    let respuesta;
    if(request.status == 200) respuesta = request.response;
    else respuesta = "No se ha encontrado el recurso.";
    console.log(respuesta);
});

request.open("GET", URL); // inicializa la petición
request.send(); // envía la petición
```

En la actualidad se utiliza **FETCH**

De esta manera, la respuesta será un JSON serializado. Para poder utilizarlo efectivamente, se lo debe deserializar.

`JSON.parse(respuesta)`

**No olvidar que cuando se envía o se recibe información, siempre es en formato de texto**

## Objeto ActiveXObject

Sabiendo que AJAX no es soportado por todos los navegadores, es necesario verificar que pueda utilizarse. En caso contrario, se utilizará ActiveXObject.

```
let request;
if(window.XMLHttpRequest) {
    request = new XMLHttpRequest();
} else { // Por ej: Internet Explorer
    request = new ActiveXObject("Microsoft.XMLHTTP");
}
```

- Petición de tipo *POST*

A diferencia de las peticiones GET (las cuales se envían por URL), las peticiones POST se envían a través de un método propio y se necesita enviar información para ejecutar dicha petición (dentro del método send y en formato JSON serializado).

```
Let request;
if(window.XMLHttpRequest) {
    request = new XMLHttpRequest();
} else {// Internet Explorer
    request = new ActiveXObject("Microsoft.XMLHTTP");
}

// Evento
request.addEventListener("load", ()=>{
    let respuesta;
    if(request.status == 200 || request.status == 201)
        respuesta = request.response;
    else respuesta = "No se ha encontrado el recurso.";
    console.log(JSON.parse(respuesta));
});

// Inicializar la petición a una URL
request.open("POST", "https://reqres.in/api/users");

// Especificar el contenido que se solicita por encabezado
request.setRequestHeader("Content-type", "application/json;charset=UTF8");

// Enviar la petición con los datos requeridos para ejecutarse
request.send(JSON.stringify({
    "nombre": "morfeo",
    "trabajo": "líder"
}));
```

Aspecto	Con GET	Con POST
Los datos son visibles en la url	Sí	No
Los datos pueden permanecer en el historial del navegador	Sí	No
Una url puede ser guardada conteniendo parámetros de un envío de datos	Sí	No
Existen restricciones en la longitud de los datos enviados	Sí (no se puede superar la longitud máxima de una url) 2083 caracteres	No
Se considera preferible para envío de datos sensibles (datos privados como contraseñas, números de tarjeta bancaria, etc.)	No (los datos además de ser visibles pueden quedar almacenados en logs)	Sí (sin que esto signifique que por usar post haya seguridad asegurada)
Codificación en formularios	application/x-www-form-urlencoded	application/x-www-form-urlencoded ó multipart/form-data. Se usa multipart para envío de datos binarios, por ejemplo ficheros.
Restricciones de tipos de datos	Sí (sólo admite caracteres ASC-II)	No (admite tanto texto como datos binarios p.ej. archivos)
Se considera preferible para disparar acciones	No (podría ser accedido por un robot que dispararía la acción)	Sí (sin que esto garantice que no pueda acceder un robot)
Riesgo de cacheado de datos recuperados en los navegadores	Sí	No
Posibles ataques e intentos de hackeo	Sí (con más facilidad)	Sí (con menos facilidad)

Activar W  
Vé a Configur

## Objeto FETCH

Forma de trabajar con AJAX y promesas (promise). Un objeto de tipo Fetch siempre devolverá una **promesa encapsulada**, es decir, no se pueden acceder a sus datos sin utilizar los métodos específicos del objeto Fetch.

Tiene el método **GET** por defecto.

```
// let request;
// if(window.XMLHttpRequest) request = new XMLHttpRequest();
// else request = new ActiveXObject("Microsoft.XMLHTTP");
// request.open("GET", URL);

const request = fetch("https://reqres.in/unknown/2");
```

Como la promesa que se obtiene ya se comporta como asíncrona, no es necesario añadir un evento que se active cuando el proceso de petición haya finalizado. Simplemente se trabaja con **then** al igual que una promesa normal.

Se puede utilizar **catch** para cuando la petición no puede finalizar con éxito.

```
const request = fetch("https://reqres.in/unknown/2");

request.then(res => { // resolve
  console.log(res);
```



```
* Response {type: "cors", url: "https://reqres.in/unknown/2"}
  body: {...}
  bodyUsed: false
  headers: Headers {}
  ok: true
  redirected: false
  status: 200
  statusText: ""
  type: "cors"
  url: "https://reqres.in/unknown/2"
  > __proto__: Response
```

Sin embargo, el contenido se encuentra encapsulado y sólo se podrá acceder mediante los siguientes métodos:

- **text()** Se obtiene una nueva promesa, a la cual se puede acceder mediante un nuevo método **then**. De esta manera, se obtiene la información en formato texto (json serializado).

```
const request = fetch(URL);
request
  .then(respuesta => respuesta.text())
  .then(texto => console.log(texto));
```

- **json()** Al igual que text, pero se obtiene el objeto json deserializado.

```
const request = fetch(URL);
request
  .then(respuesta => respuesta.json())
  .then(jsonObject => console.log(jsonObject));
```

- `blob()` Es útil para trabajar, por ejemplo, con imágenes.

```
<body>
  <img class="imagen">
  <script src="codigo.js"></script>
</body>
```

▶ Blob {size: 87425, type: "image/png"}

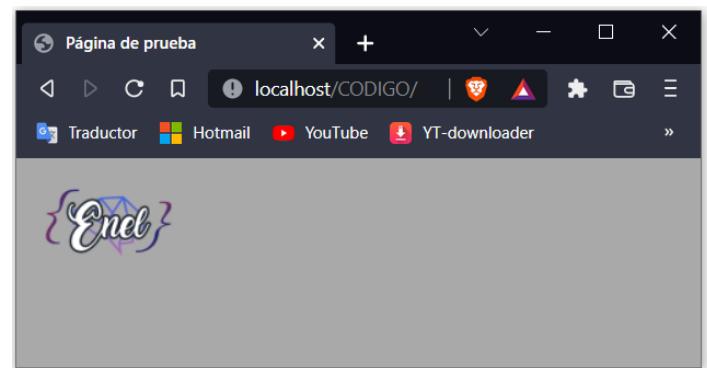
```
const imagen = document.querySelector(".imagen");

const url = "Imagenlocal.png";

const request = fetch(url);

request
  .then(respuesta => respuesta.blob())
  .then(blob => imagen.src = URL.createObjectURL(blob));
```

Muestra la imagen



El método `URL.createObjectURL` crea una URL temporal donde se almacena lo que le envíemos como parámetro, y existe sólo el tiempo que dura la petición.

- `formData()`
- `arrayBuffer()`

Para realizar una petición de tipo *POST* es necesario utilizar un segundo parámetro al utilizar el objeto Fetch, el cual es un objeto:

```
const request = fetch(URL, {
    // Especificar método que se utilizará
    method: "POST",

    // Especificar contenido del send
    body: JSON.stringify({
        "nombre": "Guillermo",
        "edad": "27"
    }),

    // Especificar el sendRequestHeader
    headers: {
        "Content-type": "application/json"
    }
});

request
    .then(respuesta => respuesta.json())
    .then(jsonObject => console.log(jsonObject));
```

Recordar que se puede utilizar **.catch** para cuando la petición no puede finalizar con éxito.

Se puede utilizar la propiedad **ok** para verificar si la petición fue ejecutada exitosamente.

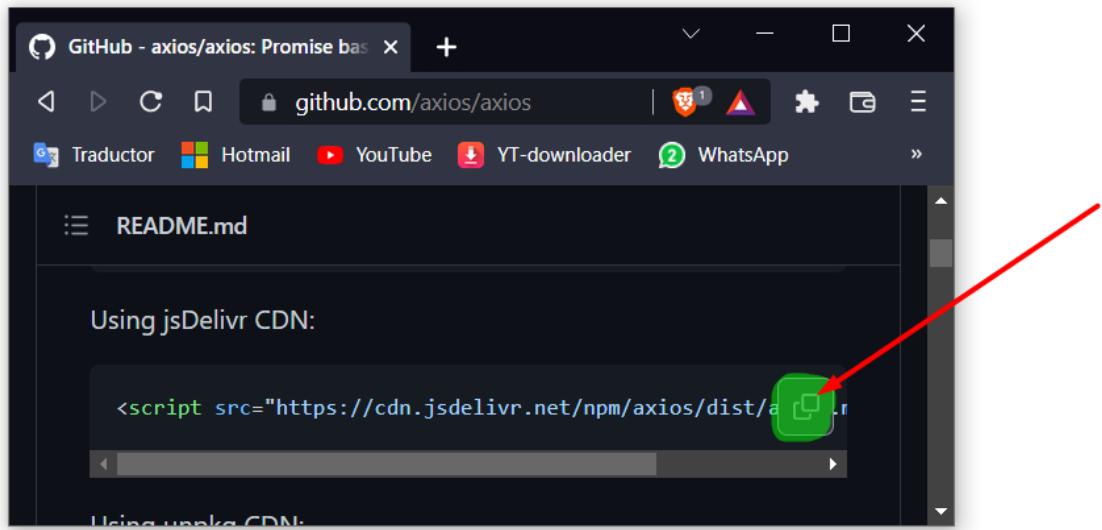
## Axios (no nativo de JavaScript)

Reemplazo más moderno de Fetch que se utiliza cuando deben realizarse múltiples peticiones, ya que es una versión optimizada de XMLHttpRequest.

Se encuentra basado en promesas (promise) y, por defecto, trabaja con las peticiones de tipo *GET*.

Para utilizar Axios, es necesario descargarlo desde <https://github.com/axios/axios>

Luego, copiar el código brindado en el recuadro de **jsDelivr CDN** y pegarlo antes de nuestro código.



```
<div>
  ...
</div>
<script src="https://cdn.jsdelivr.net/npm/axios/dist/axios.min.js">
<script src="codigo.js"></script>
```

A modo instructivo, se trabajará con un archivo de texto local que contendrá información en formato JSON serializado.

Para realizar una petición (request) con axios, se procede de la siguiente manera:

```
const url = "informacion.txt";

const request = axios(url);

request
  .then(respuesta => console.log(respuesta.data));
```

```
*informacion.txt: Bloc d... Archivo Edición Formato Ver Ayuda
{
  "nombre": "Pedro",
  "edad": "24"
}
I 100% Windows (CRLF) UTF-8
```

```
const request = fetch(URL);      ahorramos este
request                         paso de fetch
  .then(respuesta => respuesta.json())
  .then(jsonObject => console.log(jsonObject));
```

Es necesario trabajar con la propiedad ***data*** para poder acceder a la información, y no es necesario deserializar el json.

Si se desea trabajar con peticiones de tipo POST, se configura automáticamente al añadir dicho método de la siguiente manera:

```
const url = "https://reqres.in/api/users";

// Especificar contenido del send en otro parámetro
// NO ES NECESARIO SERIALIZAR EL OBJETO JSON
const request = axios.post(url, {
  "nombre": "Guillermo",
  "edad": "27"
});

request
  .then(respuesta => console.log(respuesta.data));
```

## Peticiones con Async y Await

Se pueden utilizar funciones asíncronas para realizar peticiones.

De esta manera, se podrán trabajar las peticiones en tiempo real.

```
<body>
    <button id="btn-nombre">Obtener nombre</button>
    <button id="btn-edad">Obtener edad</button>
    <div class="nombre"></div>
    <div class="edad"></div>
    <script src="https://cdn.jsdelivr.net/npm/axios/dist/axios.min.js"></script>
    <script src="codigo.js"></script>
</body>
```

```
#btn-nombre, #btn-edad {
    font-size: 40px;
    margin: 20px;
}

.nombre, .edad {
    color: #000;
    font-size: 25px;
    margin: 5px;
}
```

```
// Obtener los elementos HTML
const divNombre = document.querySelector('.nombre');
const divEdad = document.querySelector('.edad');
const btnGetNombre = document.getElementById('btn-nombre');
const btnGetEdad = document.getElementById('btn-edad');

const url = "informacion.txt"; // fichero local

const getName = async()=> {
    try {
        let request = await fetch(url); // devuelve un promise
        let respuesta = await request.json(); // devuelve el json deserializado
        let nombreObtenido = respuesta.nombre;
        divNombre.textContent = nombreObtenido;
    } catch { console.log("Error al acceder a la información."); }
}

const getAge = async()=> {
    try {
        let request = await fetch(url); // devuelve un promise
        let respuesta = await request.json(); // devuelve el json deserializado
        divEdad.textContent = `${respuesta.edad} años`;
    } catch { console.log("Error al acceder a la información."); }
}

// Añadir funcionalidad a los botones
btnGetNombre.addEventListener("click", getName);
btnGetEdad.addEventListener("click", getAge);
```

Si se desea trabajar con Axios, simplemente se reemplazan las líneas del fetch por la única línea que utiliza el método axios, además de agregar la propiedad **data**.

```
let request = await fetch(url);
let respuesta = await request.json();
```



```
let respuesta = await axios(url);
```

```
respuesta.nombre;
```



```
respuesta.data.nombre;
```

## Librerías

Una librería es un conjunto de funciones (código externo) que pueden implementarse dentro de nuestro propio código.

Cuando importamos una librería, se puede implementar como contenido de un archivo externo en formato js, y debe implementarse en diferentes partes de nuestro HTML según cómo se utilice y la prioridad que necesite.

- **Máxima prioridad:** se aplica sobre el `<head>` de nuestro HTML, así puede ejecutarse por encima de cualquier otro código de nuestra página, o incluso para comenzar a funcionar antes de cargar el cuerpo de la página.

```
<head> <!--Cabeza de la página web -->
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <script src="https://kit.fontawesome.com/62ea397d3a.js" crossorigin="anonymous"></script>
```

- **Prioridad normal:** se aplica por encima de nuestro propio código.

```
<body>
    <div>
        ...
    </div>
    <script src="https://cdn.jsdelivr.net/npm/axios/dist/axios.min.js"></script>
    <script src="codigo.js"></script>
</body>
```

- **Código extremadamente corto:** se puede aplicar directamente como una etiqueta html (por ejemplo, para ejecutar sólo un alert).

```
<script type="text/javascript">
    alert("Tu nombre es " + prompt("su nombre"));
</script>
```

## Prototipos

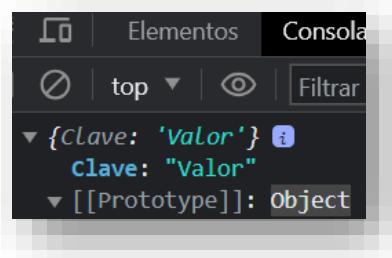
Tipo de lenguaje donde los objetos se crean a partir de la heredación de otros objetos llamados prototipos en vez de instanciación de un objeto desde cero.

En JavaScript, todo es un objeto el cual se crea a partir de un prototipo.

### Cadena de prototipos

Existe un prototipo padre, del cual se heredan todos los objetos que pueden crearse, llamado **Object**.

```
const unObjeto = {
  "Clave": "Valor"
}
```



Además, todos los objetos heredan un **tipo de dato**, el cual también es un prototipo.

Por ejemplo, si creamos una cadena de texto, éste es un objeto creado a partir de los prototipos String y Object (en ese orden).

```
const unaCadenaDeCaracteres = "Hello"; → dunder proto
console.log(unaCadenaDeCaracteres.__proto__); // prototipo String
console.log(unaCadenaDeCaracteres.__proto__.__proto__); // prototipo Object
console.log(unaCadenaDeCaracteres.__proto__.__proto__.__proto__); // null
```

Observación: **null** y **undefined** no tienen prototipo.

Al crear una función, no se está creando un objeto sino que se está creando un prototipo de tipo **function**, con herencia del prototipo Object.

```
const unaFuncion = function() { }
console.log(unaFuncion.prototype);
```

Así mismo, los prototipos pueden trabajarse como objetos, y sólo los prototipos creados por nosotros pueden ser modificados. Tienen métodos y atributos propios, así como los heredados por sus prototipos padre.

Todos los métodos que se encuentren en un objeto, como un Array, se encuentran definidos dentro del prototipo Array y no en el objeto en sí.

```
const unArray = [];
console.log(unArray);
```

*el prototipo es quien contiene todas las funciones del array*

Al crear una clase, no se crea un prototipo sino un objeto que hereda del prototipo Object. Y todos los atributos y métodos se guardan dentro del prototipo Object.

Esto permite que el método pueda ser modificado tanto desde fuera como desde dentro del prototipo.

Existe un orden de ejecución, primero se ejecuta lo definido afuera, y luego lo de dentro.

```
class UnaClase {
  constructor() {}
  print = () => console.log("Original")
```

```
const unObjeto = new UnaClase();
unObjeto.print();
```

Original

```
const unObjeto = new UnaClase();
unObjeto.print = () => console.log("Modificado desde fuera");
unObjeto.print();
```

Modificado desde fuera

```
const unObjeto = new UnaClase();
unObjeto.__proto__.print = () => console.log("Modificado desde dentro");
unObjeto.__proto__.print();
```

Modificado desde dentro

## Manipulación de prototipos

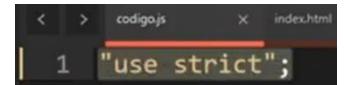
Manipulando los prototipos de un objeto, se pueden agregar nuevas funciones a otro objeto.

```
class UnaClase {  
    constructor() {}  
    print = () => console.log("Original")  
}  
  
const unObjeto = new UnaClase();  
  
const unArray = [];  
  
// Se establece que unObjeto sea el prototipo de unArray  
unArray.__proto__ = unObjeto;  
  
// Ahora el array tiene la función print definida  
unArray.print();
```

## Modo estricto

Convierte errores de JavaScript en excepciones, previniendo malas sintaxis.

Para activarlo, se escribe “use strict” al inicio del archivo JS para que el intérprete de JavaScript agregue todas las reglas del modo.



Es una buena práctica que este modo esté activo siempre.

### Activación automática

El modo estricto se activa automáticamente dentro de módulos:

```
<script type="module">
```

Las funciones flecha están en modo estricto automáticamente.

### Prohibición de sobreescritura del valor de un objeto

```
①: script.js U X
pages > test > scripts > ①: script.js > ...
1   "use strict";
2
3   const unObjeto = {} //           clave           valor     prohíbe sobreescritura
4   Object.defineProperty(unObjeto, 'nombre', {value: "Guille", writable:false});
5
6   console.log(unObjeto.nombre) // Guille
7
8   unObjeto.nombre = "Nicolas"; // ERROR
                                     ✘ ▶ Uncaught TypeError: Cannot assign to read only property 'nombre' of object '#<Object>'
                                         at script.js:8:17
```

Así mismo se pueden prohibir la creacion de nuevas claves en el objeto.

```
Object.preventExtensions(unObjeto);
```

Además, se pueden eliminar claves / propiedades del objeto con la palabra **delete**.

```
const unObjeto = {
  nombre: "Guille"
};

delete unObjeto.nombre;

console.log(unObjeto.nombre) // undefined
```

En el modo estricto, no se puede usar **delete** para eliminar variables ni funciones.

## Funciones

No se pueden utilizar funciones flecha dentro de objetos como métodos ya que la palabra reservada **this** en una función flecha hace referencia al objeto **window**.

Sin embargo, una función clásica sí puede usarse como método

```
const unObjeto = {
    nombre: "Guille",
    saludar: function(){console.log(`Hola ${this.nombre}`)}
};

unObjeto.saludar() // Hola Guille
```

Además, una función clásica puede usarse como constructor (no dentro de una class) de la siguiente manera:

```
function constructorPersona(nombre, apellido) {
    this.nombre = nombre;
    this.apellido = apellido;
}

const persona = new constructorPersona("Guille", "Hern");

console.log(persona.apellido); // Hern
```

\* No se puede usar una función flecha por el mismo motivo: **this** hace referencia a **window**.

## This contextual

La palabra reservada **this** hace referencia, por defecto y fuera de cualquier función, al objeto **window**. `console.log(this); // window`

Sin embargo, si se encuentra dentro de una función, hace referencia a la función donde se encuentre (con excepción de las funciones flecha).

Incluso permite acceder a atributos declarados dentro de objetos los cuales contengan una función.

```
function saludar() {
    // no existe el atributo nombre
    console.log(`Hola ${this.nombre}`);
}

const obj = {
    // objeto con el atributo nombre
    nombre : "Guille",
    saludar : saludar
}

obj.saludar(); // Hola Guille
```

## Recursividad

Cuando una función se llama a sí misma, como los callbacks, closures, .

Un procedimiento recursivo está dividido en tres fases principales.

### ⊕ Caso base

Cuando termina la recursión.

### ⊕ Proceso

Valor agregado o acción de la función.

### ⊕ Llamada recursiva

La función/procedimiento se llama a sí mismo

```
function imprimirNumeros(n) {
    // Caso base
    if (n <= 0)
        return;

    // Proceso
    console.log(n);

    // Llamado recursivo
    imprimirNumeros(n - 1);
}
```

## Setup de rutina recursiva

Se utiliza una función/procedimiento intermedio con un **parámetro extra** que permita iterar de forma más eficiente.

```
#define MAX_ELEMENTOS 24
#define NO_EXISTE -1

int busqueda_lineal_rec(int vector[MAX_ELEMENTOS], int tope, int numero_buscado, int i){  

    /* Caso base: ENCONTRADO */
    if(vector[i] == numero_buscado) return i;  

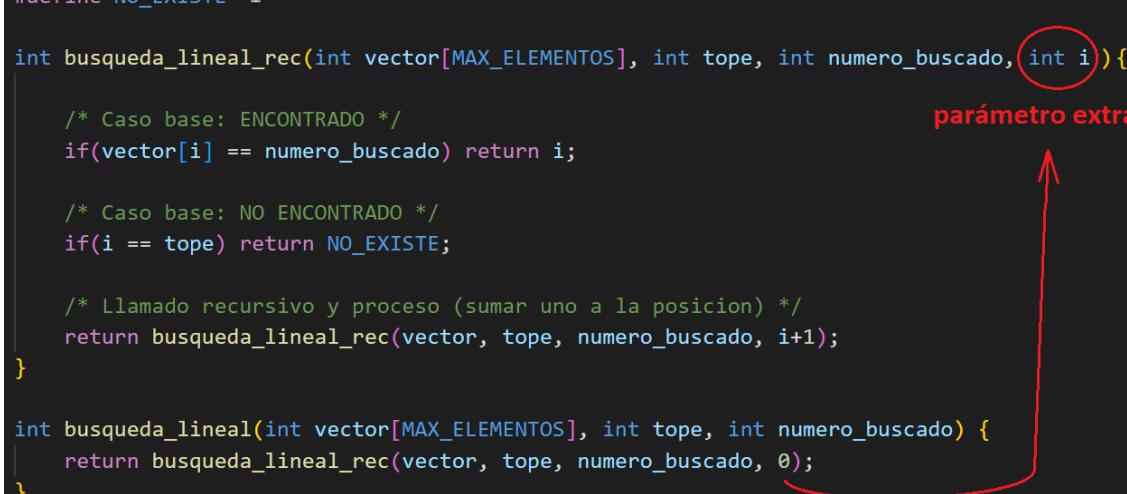
  

    /* Caso base: NO ENCONTRADO */
    if(i == tope) return NO_EXISTE;  

    /* Llamado recursivo y proceso (sumar uno a la posición) */
    return busqueda_lineal_rec(vector, tope, numero_buscado, i+1);
}

int busqueda_lineal(int vector[MAX_ELEMENTOS], int tope, int numero_buscado) {
    return busqueda_lineal_rec(vector, tope, numero_buscado, 0);
}
```



parámetro extra

Un ejemplo práctico para el uso de la recursividad es la validación de un dato.

```
const validarEdad = (error) => {
  let edad;
  try {
    if (error) edad = prompt("Introduce un número para tu edad");
    else edad = prompt("Introduce tu edad");
    edad = parseInt(edad); // convertir a entero
    if (isNaN(edad)) throw "Introduce un número válido para tu edad";
    return edad;
  } catch(err) {
    return validarEdad(err);
  }
}

const miEdad = validarEdad();
```

## Closures

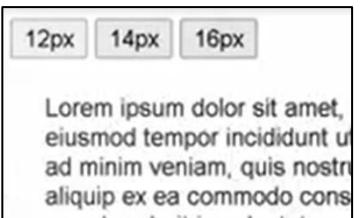
Un closure permite trabajar con una función dentro de otra función.

Por ejemplo

```
const b12px = document.getElementById("b12px");
const b14px = document.getElementById("b14px");
const b16px = document.getElementById("b16px");

const changeSize = size => {
    document.querySelector(".text").style.fontSize = `${size}px`;
}

b12px.addEventListener("click", () => changeSize(12));
b14px.addEventListener("click", () => changeSize(14));
b16px.addEventListener("click", () => changeSize(16));
```



Se puede reemplazar por:

```
const b12px = document.getElementById("b12px");
const b14px = document.getElementById("b14px");
const b16px = document.getElementById("b16px");

const changeSize = size => {
    return () => {
        document.querySelector(".text").style.fontSize = `${size}px`;
    }
}

px12 = changeSize(12);
px14 = changeSize(14);
px16 = changeSize(16);

b12px.addEventListener("click", px12);
b14px.addEventListener("click", px14);
b16px.addEventListener("click", px16);
```

## Parámetros rest

En general, las funciones pueden recibir un número específico de parámetros (los definidos en su firma), sin embargo se pueden definir parámetros rest (antes conocido como objeto Arguments) que pueden obtener parámetros no previstos pasados a una función.

Obtiene los parámetros “extra” dentro de un Array.

```
function parametrosRest (...num) {
  console.log(num);
}

parametrosRest(1, 6, 3, 2);
```

El parámetro rest siempre debe estar al final de la firma

```
function parametrosRest (unParametro, ...num) {
  // code
}
```

## Operador Spread

Convierte un array en valores por separado, es decir, los desestructura.

```
let valor1 = "valor 1";
let valor2 = "valor 2";
let valor3 = "valor 3";

let arr = ["valor 1","valor 2","valor 3"];
console.log(valor1,valor2,valor3)
console.log(...arr)
```

```
let arr = ["manzana","pera","banana"];
let arr2 = ["kiwi","naranja"];

arr.push(...arr2);

console.log(arr);
```

## Application Programming Interfaces

Permite añadir funcionalidades a nuestro programa por medio de las **APIs**, es decir, conectar dos o más aplicaciones entre sí e intercambiar información, así como conectar con otros lenguajes de programación.

Con las APIs, se pueden añadir funcionalidades como la posibilidad de “arrastrar y soltar” elementos, reconocimiento de voz, entre otros.

Existen distintos tipos de API

- API rest Se envía información a otro sitio web, el cual responde
- API del lenguaje Funciones nativas del lenguaje (como **prompt**).

### Objeto Date

Es una clase que permite trabajar las fechas.

```
const fecha = new Date();
```

- `getDate()` Obtiene la fecha del mes actual.
- `getDays()` Obtiene el día →
- `getMonth()` Obtiene el mes actual, comenzando desde 0.
- `getYear()` Obtiene el año actual (**restado 1900**).
- `getHours()` Obtiene la hora actual.
- `getMinutes()` Obtiene el minuto actual.
- `getSeconds()` Obtiene el segundo actual.

```
// Domingo -> 0
// Lunes -> 1
// Martes -> 2
// Miércoles -> 3
// Jueves -> 4
// Viernes -> 5
// Sábado -> 6
```

Un ejemplo de un reloj:

```
const addZeros = n => {
  if (n.toString().length < 2) return "0".concat(n);
  return n;
}

const actualizarHora = ()=>{
  const time = new Date();
  let hora = addZeros(time.getHours());
  let min = addZeros(time.getMinutes());
  let seg = addZeros(time.getSeconds());
  document.querySelector(".hora").textContent = hora;
  document.querySelector(".min").textContent = min;
  document.querySelector(".seg").textContent = seg;
}

actualizarHora() : 
setInterval(actualizarHora,1000)
```

22 59 23

\* En la actualidad se usan librerías para actualizar la hora ya que **setInterval** consume muchos recursos del navegador

## LocalStore y SessionStore

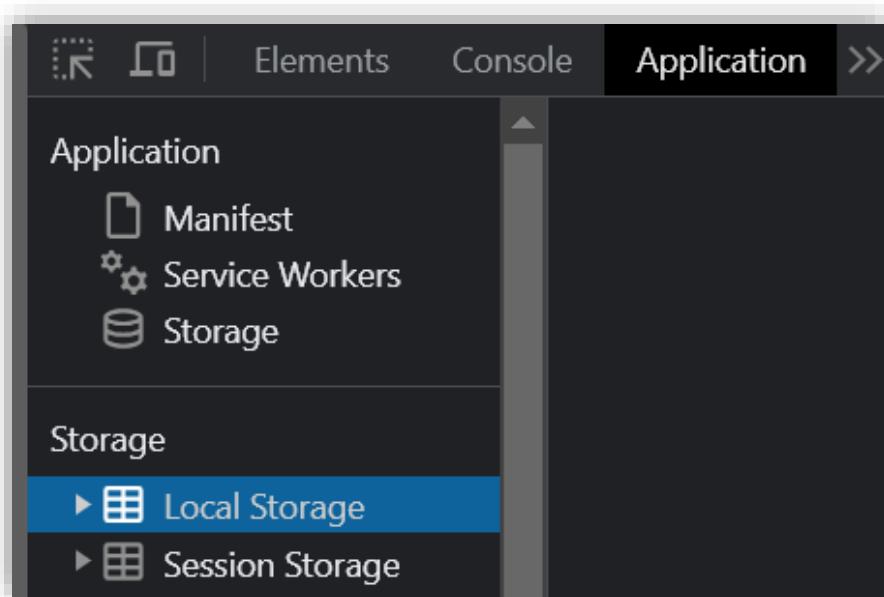
Permite almacenar información en el navegador.

- LocalStore: la información permanece incluso si el navegador fue cerrado.
- SessionStore: la información no permanece al cerrarse la página.

Los métodos para trabajar con ambos son:

- `setItem(object-key, object-value)`
- `getItem(object-key)`
- `removeItem(object-key)`
- `clear()`

Se pueden visualizar los datos guardados en la pestaña de Application en el navegador.



## Drag and Drop

API que permite detectar el arrastre de elementos de la página.

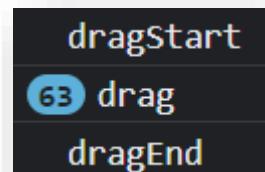
Teniendo un elemento arrastrable, se puede detectar los momentos del arrastre con los eventos siguientes:

- dragstart      detecta cuándo comienza a arrastrarse
- drag            se ejecuta repetidamente mientras se está arrastrando
- dragend        se ejecuta al soltar el elemento

```
<a id="element">
  
</a>
```



```
const element = document.getElementById("element");
element.addEventListener("dragstart", () => console.log("dragStart"));
element.addEventListener("drag", () => console.log("drag"));
element.addEventListener("dragend", () => console.log("dragEnd"));
```



Además, existen eventos en otros elementos que permiten la detección del elemento que está siendo arrastrado sobre ellos.

- dragenter      detecta cuando el elemento entró en su espacio
- dragover        se ejecuta repetidamente mientras el elemento esté dentro
- drop            detecta cuando el elemento fue soltado dentro de su espacio
- dragleave       detecta cuando el elemento salió de su espacio

```
#section {
  background: #ccc;
  padding: 100px 200px;
  margin: 50px;
  outline: 6px dashed #aaa;
}
```



```
const section = document.getElementById("section");
section.addEventListener("dragenter", () => console.log("Entraste al section"));
section.addEventListener("dragover", event => {
  event.preventDefault(); // Permite que se detecte el drop
  console.log("drag dentro del section"));

});
section.addEventListener("drop", () => console.log("Soltaste el elemento dentro"));
section.addEventListener("dragleave", () => console.log("Saliste del section"));
```

Con el Drag & Drop también se pueden transferir datos mediante los métodos del objeto **dataTransfer**.

- **getData()**
- **setData()**

```
const element = document.getElementById("element");
const section = document.getElementById("section");

element.addEventListener("dragstart", e => {
    // Setear información en el elemento un valor
    e.dataTransfer.setData("una-clave", e.target.id);
    console.log(`#${e.target.id} comenzó a arrastrarse`)
});

section.addEventListener("dragover", e => e.preventDefault()); // Detectar drop
section.addEventListener("drop", e => {
    const elementoEntrante = e.dataTransfer.getData("una-clave");
    console.log(`${elementoEntrante} entró al section`);
});
```



VER 02:47

## Geolocalization

Se puede obtener la geolocalización mediante el objeto **navigator**, el cual hace referencia al navegador (con el cual también se puede acceder a la cámara, etc).

```
const geolocation = navigator.geolocation;
```

- **getCurrentPosition(pos, err, options)** obtiene la posición actual (requiere permisos)
- **watchPosition()** detecta cambios en la posición
- **clearWatch()** detiene la detección de cambios en la posición

```
// Datos de posición
const posicion = (pos) => {
    console.log(pos); // Muestra todos Los datos de posición
    console.log(pos.coords.latitude); // Muestra la latitud
    console.log(pos.coords.longitude); // Muestra la Longitud
}

const err = e => console.log(e); // Muestra el error que ocurra

const options = {
    maximumAge: 0, // Se pedirá La información real (sino usa la del caché)
    timeout: 3000, // Tiempo que tarda en darnos La información (en ms)
    enableHighAccuracy: true // Activa La alta precisión
}

geolocation.getCurrentPosition(posicion, err, options);
```

## Historial

Utiliza el objeto **window** para acceder al historial por medio de su propiedad **History**, la cual tiene los siguientes métodos:

- **back()** vuelve hacia la página anterior (como el botón hacia atrás)
- **forward()** va hacia la página siguiente (como el botón hacia adelante)
- **go()** va al sitio indicado con un numero relativo
  - El cero recarga la página
  - El uno va una página hacia adelante
  - El menos uno va una página hacia atrás
- **pushState()** modifica la URL y conserva la info
- **replaceState()** modifica la URL y no la conserva

Al llamar al objeto, se puede obtener la información del historial.

```
const historial = window.history;
console.log(historial);
```

length	cantidad de páginas del historial de navegación actual
scrollRestoration	
state	

La función **pushState** permite agregar argumentos a la página haciendo una nueva entrada en el historial y genera el evento **popstate** (**replaceState** no lo hace) sin recargar la página.

```
file:///C:/Users/Usuario/Desktop/Curso%20de%20JS/Curso%20de%20JS/CODIGO/index.html
history.pushState({nombre: "pedro"}, "", "?jaja")
undefined
location.href
"file:///C:/Users/Usuario/Desktop/Curso%20de%20de%20JS/Curso%20de%20JS/CODIGO/index.html?jaja"
```

Entonces, se puede acceder al state mediante los siguientes comandos:

```
> history.state  
< null  
> history.pushState({nombre:"pedro","","?jaja"})  
< undefined  
> history.state  
< > {nombre: "pedro"}
```

Se puede agregar un escuchador que detecte cuando se hizo un cambio en el state:

```
addEventListener("popstate", (e)=>{  
    console.log(e.state)  
})
```

## FileReader

Permite la lectura de archivos y datos proporcionados por el usuario (a diferencia de fetch que permitía la lectura de archivos ya fijados en el servidor).

- `readAsText()` permite leer un archivo de formato texto

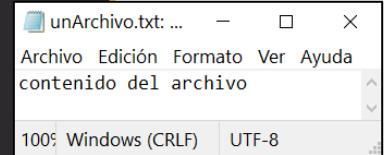
```
<input id="input-file" multiple="" type="file" name="file-input" required="">
```

El atributo `multiple` permite la selección de más de un archivo

```
const inputFile = document.getElementById("input-file");

inputFile.addEventListener("change", () => { // Evento de cambio en el input
    const file = inputFile.files[0]; // Obtiene el primer/único archivo seleccionado
    readFile(file);
});

const readFile = file => {
    const reader = new FileReader();
    reader.readAsText(file);
    reader.addEventListener("load", progressEvent => { // resultado de lectura
        console.log(progressEvent);
    });
}
```



De esta manera se obtiene un **ProgressEvent** cuyo `currentTarget` contiene el **FileReader** con el contenido del archivo de texto seleccionado.

Este método también puede ser útil para cargar archivos JSON en formato string.

```
// Resultado de lectura
reader.addEventListener("load", progressEvent => {
    const result = progressEvent.currentTarget.result;
    const json = JSON.parse(result);
    console.log(json);
});
```

```
▼ ProgressEvent {isTrusted: true, lengthComputable: true, readyState: "loaded", result: "contenido del archivo", target: FileReader}
  isTrusted: true
  bubbles: false
  cancelBubble: false
  cancelable: false
  composed: false
  ▼ currentTarget: FileReader
    error: null
    onabort: null
    onerror: null
    onload: null
    onloadeddata: null
    onloadstart: null
    onprogress: null
    readyState: 2
    result: "contenido del archivo"
```

```
▼ {language: 'JavaScript'} ⓘ
  language: "JavaScript"
  ▶ [[Prototype]]: Object
```

Un Archivo.json

```
1  {
2   |   "language": "JavaScript"
3 }
```

- `readAsDataURL()` permite leer imágenes, videos, etc.

```
const inputFile = document.getElementById("input-file");
const imagesContainer = document.querySelector(".images-container");

inputFile.addEventListener("change", () => {
    // Obtener todos los archivos seleccionados
    const files = inputFile.files;
    readFile(files);
});

const readFile = files => {
    for (let file of files) {
        const reader = new FileReader();
        reader.readAsDataURL(file);
        reader.addEventListener("load", progressEvent => {
            const result = progressEvent.currentTarget.result;
            const newImage = `<img src=${result}>`;
            imagesContainer.innerHTML += newImage;
        });
    }
}
```

```
.images-container {
    display: grid;
    grid-template-columns: repeat(auto-fill, minmax(250px, 1fr));
    grid-auto-rows: 100px;
    grid-gap: 6px;
}

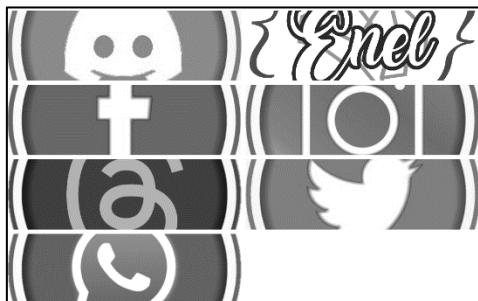
.image {
    width: 100%;
    height: 100%;
    object-fit: cover;
}
```

```
img:first-child {
    grid-column: 1 / 3;
    grid-row: 1 / 3;
}

img:first-child {
    grid-column: 1 / 3;
    grid-row: 1 / 3;
}

img:nth-child(6) {
    grid-column: 2 / 4;
    grid-row: 3 / 5;
}

img:nth-child(10) {
    grid-row: span 2;
}
```



Puede ser necesaria hacer una validación del tipo de archivo que se está seleccionando para que se ejecute correctamente la función adecuada para cada tipo de archivo.

Esta API se puede utilizar junto con Drag & Drop

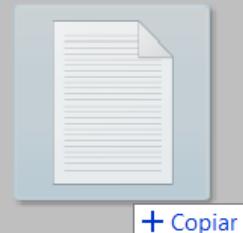
```
<div class="zona-arrastre">Arrastre un archivo aquí</div>
```

```
const colorDragableZone = (element, inZone) => {
  if (inZone) {
    element.style.color = "#4c3688";
    element.style.border = `4px dashed #4c3688`;
    element.style.backgroundColor = "#c0bfef";
    element.innerText = "Ya puede soltar el archivo";
  } else {
    element.style.color = "#6e52bb";
    element.style.border = `4px dashed #6e52bb`;
    element.style.backgroundColor = "#e8e8e8";
    element.innerText = "Arrastre un archivo aquí";
  }
}
```

```
.zona-arrastre {
  font-family: sans-serif;
  padding: 60px 20px;
  border: 4px dashed #6e52bb;
  background-color: #e8e8e8;
  max-width: 500px;
  color: #6e52bb;
  margin: auto;
  margin-top: 30px;
  text-align: center;
}
```

Arrastre un archivo aquí

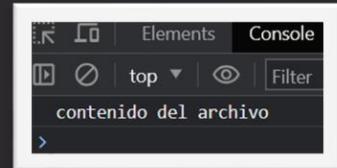
Ya puede soltar el archivo



+ Copiar

```
const zonaArrastre = document.querySelector(".zona-arrastre");
zonaArrastre.addEventListener("dragleave", e => colorDragableZone(e.target, false));
zonaArrastre.addEventListener("dragenter", e => colorDragableZone(e.target, true));
zonaArrastre.addEventListener("dragover", e => e.preventDefault()); // Habilitar drop
zonaArrastre.addEventListener("drop", e => {
  e.preventDefault(); // Deshabilita que el archivo se abra en otra pestaña
  colorDragableZone(e.target, false);
  const archivoArrastrado = e.dataTransfer.files[0];
  cargarArchivo(archivoArrastrado);
});

const cargarArchivo = file => {
  const reader = new FileReader();
  reader.readAsText(file);
  reader.addEventListener("load", e => {
    const text = e.currentTarget.result;
    console.log(text); _____
  });
}
```



Para cargar archivos que no sean texto plano, se debe utilizar:

```
const cargarArchivo = file => {
    const reader = new FileReader();
    reader.readAsDataURL(file);
    reader.addEventListener("load", e => {
        const url = URL.createObjectURL(file);

        // En caso de que sea una imagen:
        const img = document.createElement("IMG");
        img.setAttribute("src", url);

        // Pega la imagen en el body
        document.getElementsByTagName("body")[0].appendChild(img);
    });
}
```

Para el caso de un archivo de video, se lo debe tratar de la siguiente forma:

```
const cargarArchivo = file => [
    const reader = new FileReader();
    reader.readAsArrayBuffer(file);
    reader.addEventListener("load", e => {

        // Cargamos el video a un formato valido
        const videoBlob =
            new Blob([new Uint8Array(e.currentTarget.result)], {type: "video/mp4"});
        const url = URL.createObjectURL(videoBlob);

        // En caso de que sea una imagen:
        const video = document.createElement("VIDEO");
        video.setAttribute("src", url);

        // Pega la imagen en el body
        document.getElementsByTagName("body")[0].appendChild(video);

        video.play(); // Reproduce el video
    });
]
```

Algunos archivos pueden demorar un tiempo en cargar, por lo que puede ser de utilidad crear una barra de progreso para que el usuario no crea que la página se congeló o que falló algo.

```
reader.addEventListener("progress", e=>{ // Se ejecuta cada 4% de archivo cargado
    //console.log(`${e.loaded} cargado de ${e.total}`); // En numeros absoultos
    console.log(` ${(e.loaded * 100 / e.total).toFixed(0)}%`); // En porcentaje sin decimales
});
```

VER 04:02 (master) para una interfaz

## IndexedDB

Almacena información en el navegador de forma similar a LocalStorage utilizando **CRUD**, orientado a objetos, asíncrono y trabaja con eventos del DOM.

No-SQL, trabaja con clave-valor al igual que LocalStorage.

Create  
Read  
Update  
Delete

La Indexed Data Base se crea de la siguiente manera

```
// Crear (o abrir) una Base de Datos      nombre , version
const IDBRequest = window.indexedDB.open("myDataBase", 1);

IDBRequest.addEventListener("upgradeneeded", () => {
    // Esto se ejecuta sólo si la base de datos aún no existe
    // Normalmente aquí se crean las Tablas (Object stores)
    console.log("Creado correctamente");
});

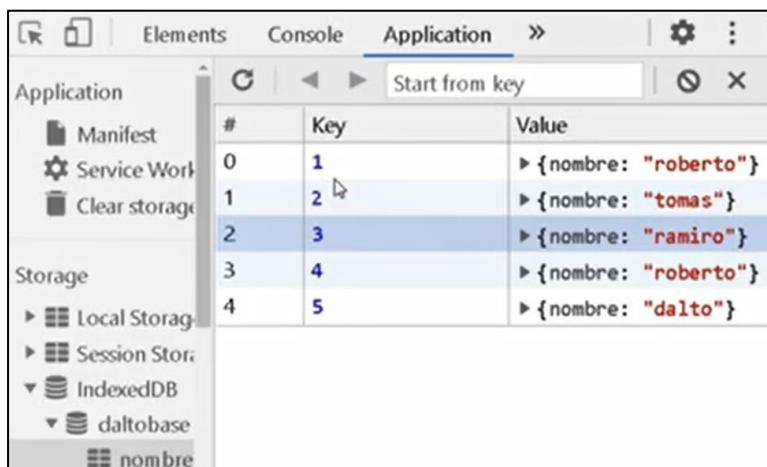
IDBRequest.addEventListener("success", () => console.log("Abierto correctamente"));
IDBRequest.addEventListener("error", () => console.log("Ocurrió un error"));
```

Para crear las Tablas / Object stores, se procede de la siguiente manera

```
IDBRequest.addEventListener("upgradeneeded", () => {
    const db = IDBRequest.result; // Data Base
    db.createObjectStore("nombres", { // Crear una 'Tabla' llamada "nombres"
        autoIncrement: true // key que se autoincrementa cada vez que se agrega algo
    });
});
```

\* Existe otra forma de trabajar sin autoIncrement que es KeyPath, el cual establece que el key será un string definido por nosotros en vez de un key que se autoincrementa automáticamente

Los objetos que se encuentren se pueden visualizar en la consola del navegador.



The screenshot shows the Chrome DevTools Application tab. On the left, there's a sidebar with 'Manifest', 'Service Worker', and 'Clear storage' under 'Application'. Below that, 'Storage' is expanded, showing 'Local Storage', 'Session Storage', and 'IndexedDB'. Under 'IndexedDB', there's a section for 'daltoibase' which contains a table named 'nombre'. The table has columns '#', 'Key', and 'Value'. The data is as follows:

#	Key	Value
0	1	▶ {nombre: "roberto"}
1	2	▶ {nombre: "tomas"}
2	3	▶ {nombre: "ramiro"}
3	4	▶ {nombre: "roberto"}
4	5	▶ {nombre: "dalto"}

Teniendo una indexDataBase llamada "myDataBase" y una tabla (object store) llamado "nombres", se puede configurar las operaciones del CRUD:

```
const addObject = object => {
  const db = IDBrequest.result; // DataBase
  const IDBtransaction = db.transaction("myDataBase", "readwrite");
  const objectStore = IDBtransaction.objectStore("nombres");
  objectStore.add(object);
  IDBtransaction.addEventListener("complete", () => {
    console.log("Se ha añadido el objeto");
  });
}
```

```
const readObjects = () => {
  const db = IDBrequest.result; // DataBase
  const IDBtransaction = db.transaction("myDataBase", "readonly");
  const objectStore = IDBtransaction.objectStore("nombres");
  const cursor = objectStore.openCursor();
  cursor.addEventListener("success", () => {
    if (cursor.result) {
      console.log(cursor.result.value); // Muestra el objeto Leído
      cursor.result.continue(); // Lee el siguiente objeto
    } else {
      // Una vez que termine de leer todo, ejecuta este bloque
      console.log("Todos los datos fueron leídos");
    }
  });
}
```

```
// Si el objeto ya existe, lo modifica
// Si el objeto no existe, lo crea
const editObject = (key, object) => {
  const db = IDBrequest.result; // DataBase
  const IDBtransaction = db.transaction("myDataBase", "readwrite");
  const objectStore = IDBtransaction.objectStore("nombres");
  objectStore.put(object, key);
  IDBtransaction.addEventListener("complete", () => {
    console.log("Se ha modificado el objeto");
  });
}
```

```
const deleteObject = key => {
  const db = IDBrequest.result; // DataBase
  const IDBtransaction = db.transaction("myDataBase", "readwrite");
  const objectStore = IDBtransaction.objectStore("nombres");
  objectStore.delete(key);
  IDBtransaction.addEventListener("complete", () => {
    console.log("Se ha eliminado el objeto");
  });
}
```

## MatchMedia

Permite utilizar **@media** de CSS en JavaScript.

```
@media only screen and (max-width: 500px) {  
}
```

```
const mq = matchMedia("(max-width: 500px)");
```

Se verifica que se cumple mediante su atributo *matches*:

Y tiene un evento que se dispara cada vez que cambie

```
if (mq.matches) {  
    // Hay < 500px de ancho  
}
```

```
mq.addEventListener("change", mq => {  
    console.log("La resolución cambió");  
});
```

No es recomendable utilizar esta API para aplicar estilos ni aplicar media-queries que puedan usarse en CSS, sino para cambiar clases u otras funciones que no puedan hacerse en CSS directamente.

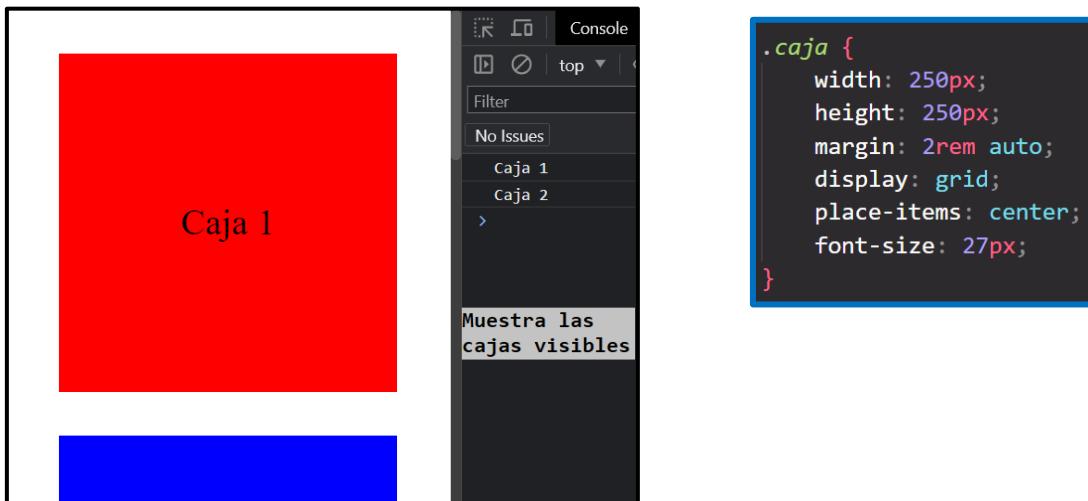
## Intersection Observer

Se utiliza para verificar que algo esté en el viewport del navegador, es decir, si algo está visible en la pantalla.

```
const observer = IntersectionObserver(callback, options);
```

Se puede usar de la siguiente manera:

```
<div class="caja" style="background-color: red">Caja 1</div>
<div class="caja" style="background-color: blue">Caja 2</div>
<div class="caja" style="background-color: green">Caja 3</div>
<div class="caja" style="background-color: yellow">Caja 4</div>
<div class="caja" style="background-color: purple">Caja 5</div>
```



```
const htmlElements = document.querySelectorAll(".caja");

const verifyVisibility = entries => { // entries es un array
    for (const entry of entries) {

        // isIntersecting es true cuando el elemento se visualiza
        if (entry.isIntersecting) {
            const element = entry.target;
            console.log(element.textContent);
        }
    }
}

const observer = new IntersectionObserver(verifyVisibility);

for (const caja of htmlElements) {
    observer.observe(caja); // Observa el html element
}
```

Se puede configurar el observador mediante **options**.

Por ejemplo, se puede establecer que se detecte un elemento como visualizado incluso antes de que realmente sea visible (por ejemplo: haciendo que se dispare el evento 30 píxeles arriba)

```
const options = { // es un objeto
  rootMargin: "30px" // Detectará el htmlElement 30 píxeles antes
}

const observer = new IntersectionObserver(verifyVisibility, options);
```

Esto puede ser útil para crear **Lazy Loads** (cargar más contenido en la página antes de llegar al final, como sucede en aplicaciones sociales como Instagram que nunca se llega al final de la página porque, al acercarse, se añaden más elementos)

VER 05:45 master

## Notifications 06:13

Es una forma de enviar notificaciones. Requiere que el usuario acepte recibir notificaciones para funcionar.

## Navigator

Contiene información del navegador.

Se pueden acceder a las propiedades siguientes:

```
navigator.appCodeName;           // No siempre es correcto
navigator.appName;              // No siempre es correcto
navigator.appVersion;            // No siempre es correcto
navigator.userAgent;             // No siempre es correcto
navigator.geolocation;           // Objeto GeoLocation
navigator.hardwareConcurrency;   // Núcleos del procesador
navigator.language;              // Idioma del usuario o del navegador
navigator.languages;             // Array de idiomas del usuario
navigator.mimeTypes;             // Tipos de archivos que se pueden usar en NodeJS
navigator.onLine;                // Verifica que se esté conectado a internet
navigator.cookieEnabled;         // Verifica que se hayan aceptado cookies
navigator.permissions;           // Permisos del navegador
navigator.platform;              // win32 o win64, Mac, Linux
navigator.plugins;               // Plugins instalados
navigator.product;               // Gecko
navigator.serviceWorker;
```

También tiene los siguientes métodos:

- [getUserMedia\(\)](#) permite acceder a la cámara y/o micrófono, lo cual puede reemplazarse con mediaStream, mediaDevice o mediaRecorder.
- [sendBeacon\(\)](#) usado para transferir, de forma asíncrona, conjuntos pequeños de datos HTTP del navegador al Servidor.
- [javaEnabled\(\)](#) devuelve True o False si Java está habilitado.
- [vibrate\(\)](#) activa función de vibración del dispositivo.
- [registerContentHandler\(\)](#)
- [registerProtocolHandler\(\)](#)
- [requestMediaKeySystemAccess\(\)](#)

## Memoization

Acorta tiempos de ejecución de funciones repetidas.

Es decir, se almacenará el resultado de una función en un array (usado como caché) y luego, cada vez que se ejecute esa misma función con los mismos parámetros, en vez de ejecutarse todo el proceso nuevamente, lo que hará será devolver el resultado almacenado en el array (Caché).

```
const cache = [];
const memoizer = func => {
    return e => {
        const index = e.toString();
        if (cache[index] === undefined)
            cache[index] = func(e);
        return cache[index];
    }
}
```

De esta manera, se crea el array que funcionará como caché de funciones.

Cada vez que se ejecute la función, si esta no está almacenada, se guardará el resultado obtenido en el array caché. Si la función está almacenada, directamente devuelve el resultado ya conocido.

Se usa de la siguiente manera:

```
// Definimos La función Larga
function funcionLarga(...params) {
    // proceso que demorará en finalizar
}

// Definimos La función dentro del memoizer
const memo = memoizer(funcionLarga);

// Ejecutamos La función desde el memoizer
memo(parámetros);
```

## Caché

Almacena información temporalmente.

Por ejemplo, puede guardar cómo se ve una página web en el navegador del usuario para que, al ingresar nuevamente en la página en otro momento, se ahorre el tiempo de carga de la página desde el servidor y muestre directamente lo que se encuentra en el caché.

Dicho caché se almacena dentro de **Cache Storage**, el cual puede contener un conjunto de **ObjectStore** (como indexedDB).

#	Name	Response-Type	Content-Type	Content-Length	Time Cached	Vary Header
	object-store-name					

Cada ObjectStore contiene:

- **Name** nombre del recurso / objeto / página.
- **Response-Type** tipo de información que contiene.
- **Content-Type** tipo de contenido (ej: text/html).
- **Content-Length** tamaño del contenido almacenado.
- **Time cached** momento en que se almacenó el objeto en caché.

Para acceder al caché, se hace mediante promises.

```
caches.open("object-store-name")
  .then(cache => {
    // code
  });
}
```

El objeto **cache** obtenido en el then tiene los siguientes métodos:

- **add(*request*)** añade un recurso (ej: URL) y la guarda en el caché.

```
cache.add("index.html");
```

- **addAll(*requests*)** permite añadir un conjunto de recursos usando un array.

```
cache.addAll(["index.html", "style.css", "script.js"]);
```

- **delete(*request, options*)** elimina un elemento del ObjectStore del caché, devolviendo un response de promise true si lo logra, false si no.

```
cache.delete("index.html").then(response => {
  // code
});
```

- **keys(*request, options*)** devuelve un response de promise de las keys del ObjectStore.
- **match(*request, options*)** devuelve un response de promise de la primera coincidencia encontrada en el ObjectStore.
- **matchAll(*request, options*)** devuelve un array de responses de promise de todas las coincidencias encontradas en el ObjectStore.
- **put(*request, response*)** permite modificar un recurso que ya se encuentra en el caché.

```
caches.open("archivos-estaticos").then(cache => {
  fetch("index.html").then(res=>{
    cache.put("index.html",res)
  })
})
```

## Web Workers

Permite la ejecución de procesos simultáneamente, en paralelo.

Esto es útil para los casos en que tenemos procesos que pueden demorar un tiempo, como por ejemplo, la lectura de datos en una Base de Datos.

Mientras la página esté ejecutando un proceso, se congelará hasta que termine, no actualizando elementos de la interfaz ni permitiendo que el usuario pueda usar la página de ninguna manera (pudiendo hacer creer al usuario que la página dejó de funcionar).

```
funcionQueDemoraMuchoEnFinalizar();
// La página se congela
// Ningún elemento de la interfaz se actualiza
```

- **Dedicated worker**

Permite la ejecución de código en otro archivo de JavaScript, haciendo que la interfaz principal no se congele.



```
const worker = new Worker("./dedicatedworker.js");
```



```
console.log("Hola");
```



Sin embargo, el dedicatedworker.js no tiene las mismas características que un código de JavaScript normal.

En primer lugar, no tiene como base el objeto global **window**.

Además, hay que tener en cuenta que, internamente, se está ejecutando un bucle que permite la intercomunicación entre un worker y su origen, por lo que es necesario terminarlo con el método **terminate**.



```
const worker = new Worker("./dedicatedworker.js");

/* acciones con el worker */

worker.terminate();
```

Aclaración: La política **same-origin** prohíbe que se acceda a un JS que se encuentre en un protocolo / host / puerto diferente del origen.

- **postMessage(params)** Permite el envío de parámetros de un archivo a otro.

JS script.js

```
const worker = new Worker("./dedicatedworker.js");

worker.postMessage("Aloja"); // Mensaje enviado a dedicatedworker.js

worker.addEventListener("message", e => {
    console.log(e); // Objeto MessageEvent
    console.log(e.data); // Mensaje recibido de dedicatedworker.js
    worker.terminate(); // Terminamos el bucle luego del mensaje
});
```

JS dedicatedworker.js

```
addEventListener("message", e => {
    console.log(e); // Objeto MessageEvent
    console.log(e.data); // "Aloja"
    postMessage("Este mensaje será devuelto a script.js");
});
```

- Service worker

Para acceder al ServiceWorker, se debe hacer por medio de **navigator**.

```
if (navigator.serviceWorker) {  
  
    // Instalar Service Worker en el navegador del usuario  
    // el cual funciona en segundo plano  
    // (incluso si el navegador se cierra)  
    navigator.serviceWorker.register("serviceworker.js");  
}  
else console.log("Service Worker no soportado!");
```

JS script.js

The screenshot shows the Chrome DevTools Application tab with the Service Workers section selected. It lists a single worker named "serviceworker.js" which is currently "activated". Below the worker details, there are buttons for "Push", "Sync", and "Periodic Sync" along with their respective message inputs. On the left side of the screenshot, there is a separate developer console window showing the result of the JavaScript code execution.

```
> console.log(navigator.serviceWorker);  
  
ServiceWorkerContainer  
  ▾ controller: ServiceWorker  
    onerror: null  
    onstatechange: null  
    scriptURL: "http://127.0.0.1:5500/serviceworker.js"  
    state: "activated"  
    ▶ [[Prototype]]: ServiceWorker  
    oncontrollerchange: null  
    onmessage: null  
    onmessageerror: null  
    ▶ ready: Promise {<fulfilled>; serviceWorkerRegistration}  
    ▶ [[Prototype]]: ServiceWorkerContainer
```

Por otro lado, en el archivo que funcionará como ServiceWorker, se pueden escribir instrucciones con una sintaxis específica (como **self** en vez de **this**) para obtener los distintos eventos de su ciclo de vida.

```
self.addEventListener("evento", e => {  
    . . .  
});
```

JS serviceworker.js

Algunos eventos del ServiceWorker son:

- **message** Recibe la comunicación entre la página y el serviceworker que se realiza con el método `postMessage`.

```
const serviceworker = navigator.serviceWorker;
if (serviceworker) {

    // Instalar o abrir
    serviceworker.register("serviceworker.js");

    // Enviar mensaje al ServiceWorker
    serviceworker.ready.then(response => {
        response.active.postMessage("MENSAJE");
    });

    // Recibir mensaje del ServiceWorker
    serviceworker.addEventListener("message", e => {
        console.log(e.data);
    });
}

} else console.log("Service Worker no soportado!");
```

JS script.js

```
self.addEventListener("message", e => {
    console.log("Mensaje recibido:");
    console.log(e.data);

    // Enviar mensaje al script de origen
    e.source.postMessage("OTRO MENSAJE");
});
```

JS serviceworker.js

- **install** Cuando se instala / activa el service worker por primera vez.  
Puede ser útil para cachear recursos estáticos.
- **activate** Luego de instalarse. Puede ser útil para limpiar caché antiguo.

```
const version = "version 1"; // ejemplo
self.addEventListener("install", installed => {
  console.log("Instalando ServiceWorker");
  caches.open(version).then(cache => {
    cache.add("index.html")
    .then(response => {
      console.log("Información cacheada");
    }).catch(e => {
      console.log(e);
    });
  });
});

self.addEventListener("activate", e => {
  console.log("Activando ServiceWorker");
  caches.keys().then(key => Promise.all(
    key.map(cache => {
      if (cache !== version) {
        console.log("caché antiguo borrado");
        return caches.delete(cache);
      }
    })
  ));
});
```

En este caso, si modificamos la versión del caché al instalarse el service worker, notaremos que se crean dos ObjectStores.

Cache Storage
version 1 - http://localhost
version 2 - http://localhost

- **error** Cuando ocurre un error en el service worker.

- **fetch** Cada vez que se realiza una solicitud de red. Permite manejar e interceptar las solicitudes e incluso trabajar desde el caché.

Una opción de uso de este evento es poder mostrar la página incluso si no hay conexión a internet (la almacenada en caché).

```
self.addEventListener("fetch", e => {
  e.respondWith(async () => {
    const respuestaEnCache = await caches.match(e.request);
    if (respuestaEnCache) return respuestaEnCache;
    return e.request;
  })
});
```

- **push** Intercepta las notificaciones push del servidor.
- **notificationclick** Intercepta el clickeado de una notificación.
- **sync** Los eventos de sincronización permiten ejecutar tareas en segundo plano incluso cuando la página no está abierta, como sincronizar datos con el servidor cuando se restablece la conexión.

- Shared worker
- Abstract worker

## Cookies

Las cookies son datos que se guardan en el navegador del usuario.

Existen distintos tipos de cookies:

- ✓ Exceptuadas
  - De entrada de usuario
  - De identificación en sesiones
  - De seguridad
  - De reproductor multimedia
  - De sesión para equilibrar carga
  - De personalización de interfaz
  - De complemento (plug-in) para intercambiar contenidos sociales
- ✓ No exceptuadas
- ✓ Según entidad que las gestione: propias o de otra página externa
- ✓ Según el tiempo que permanecen activadas
- ✓ Según finalidad
  - Análisis: para medir cantidad de usuarios
  - Publicitarias: muestra publicidad según las búsquedas del usuario
  - Sociales: para muestreo de widgets

VER 8:15



## Descargas

El usuario puede realizar descargas de archivos de texto mediante el siguiente código:

```
function sendToDownload(text="", fileName="file.txt") {  
    // Crea un nuevo Blob con el contenido que deseas guardar  
    const blob = new Blob([text], { type: "text/plain" });  
  
    // Crea un enlace para descargar el archivo  
    const url = window.URL.createObjectURL(blob);  
  
    // Crea un elemento de enlace y configura sus atributos  
    const a = document.createElement("a");  
    a.href = url;  
    a.download = fileName;  
    a.click(); // Simula hacer clic en el enlace para iniciar la descarga  
  
    window.URL.revokeObjectURL(url); // Libera el objeto URL creado  
}
```

## Objeto Screen

Es un objeto que permite acceder a algunas propiedades de la pantalla

```
AnchoTotal = screen.width //ancho total de la pantalla  
AlturaTotal = screen.height //altura total de la pantalla  
  
AnchoDisponible = screen.availWidth //ancho disponible de la pantalla  
AlturaDisponible = screen.availHeight //altura disponible de la pantalla  
  
Resolucion = screen.pixelDepth //resolución de color de la pantalla  
Profundidad = screen.colorDepth //profundidad de bits de la paleta de colores
```

innerHeight // pantalla disponible

## Objeto Canvas

Permite la creación de gráficos 8:54

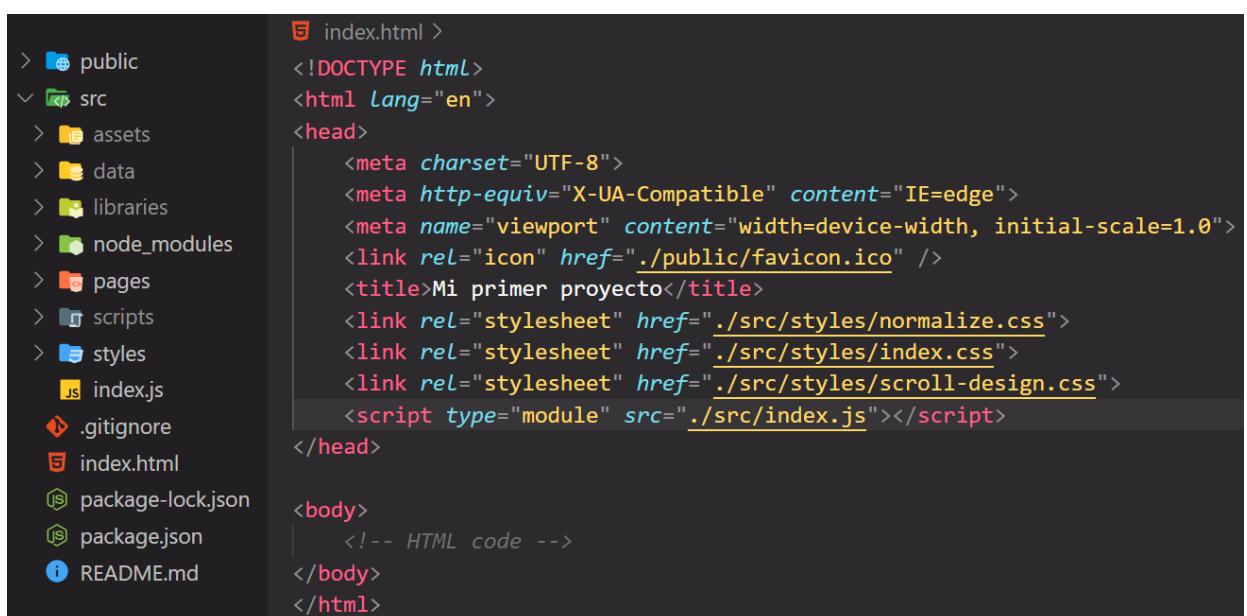
- strokeRect
- strokeStyle
- fillRect
- fillStyle
- lineWidth
- moveTo
- lineTo
- closePath
- beginPath

## Web Paint

# Proyecto

La forma de organizar un proyecto puede ser muy variada, especialmente cuando se trabaja con frameworks. Sin embargo, puede ser de utilidad conocer las partes fundamentales de un proyecto.

- Root (carpeta raíz) Contiene el proyecto web junto con archivos de configuración.
  - index.html Página principal del sitio web.
  - readme.md Información para el repositorio de GitHub.
  - package.json Configuración de NodeJS.
  
- public Contiene recursos estáticos que no serán procesados por el WebPack.
  
- src Contiene todo el código fuente del sitio web.
  - ✓ Assets Contiene recursos estáticos del sitio web.
    - Images Imágenes estáticas del sitio web.
    - Icons Íconos del sitio web (incluido favicon.png)
    - Fonts Fuentes tipográficas personalizadas.
  
  - ✓ Pages Contiene archivos **HTML** individuales para cada página.
  - ✓ Styles Contiene archivos **CSS** que controlan la apariencia del sitio.
  - ✓ Scripts Contiene archivos **JavaScript** que añaden interactividad.
  - ✓ Libraries Almacena frameworks de terceros.
  - ✓ Data Almacena datos en formato **JSON** u otros que se necesiten.
  - ✓ Config / Settings Para archivos de configuración como variables de entorno.
  - ✓ Uploads En caso de que se permita al usuario cargar archivos.
  - ✓ Test Para pruebas automatizadas.
  - ✓ node\_modules Contiene las dependencias de **NodeJS** instalados con **NPM**.



The screenshot shows a file explorer on the left and a code editor on the right. The file explorer displays a project structure:

```

> public
< src
  > assets
  > data
  > libraries
  > node_modules
  > pages
  > scripts
  > styles
    > index.js
  .gitignore
  index.html
  package-lock.json
  package.json
  README.md

```

The code editor shows the content of the index.html file:

```

index.html >
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <link rel="icon" href="./public/favicon.ico" />
  <title>Mi primer proyecto</title>
  <link rel="stylesheet" href="./src/styles/normalize.css">
  <link rel="stylesheet" href="./src/styles/index.css">
  <link rel="stylesheet" href="./src/styles/scroll-design.css">
  <script type="module" src="./src/index.js"></script>
</head>
<body>
  <!-- HTML code -->
</body>
</html>

```