JAVASCRIPT

Carlos Maldonado

5IV7

JavaScript: El Lenguaje de la Web Dinámica

JavaScript (a menudo abreviado como JS) es la piedra angular del desarrollo web moderno. Es un lenguaje de programación que permite a los desarrolladores implementar características complejas en las páginas web. Cada vez que una página web hace algo más que mostrar información estática, como mostrar actualizaciones de contenido en tiempo real, mapas interactivos, animaciones gráficas en 2D/3D o videos, puedes estar seguro de que JavaScript está involucrado.

Nacido en 1995 de la mano de Brendan Eich en Netscape, fue creado en tan solo 10 días bajo el nombre de Mocha. Posteriormente se renombró a LiveScript y, finalmente, a JavaScript, en una decisión de marketing para capitalizar la popularidad del lenguaje Java de Sun Microsystems en ese momento, aunque ambos lenguajes son muy diferentes y no tienen relación directa.

Inicialmente, su propósito era añadir interactividad simple a las páginas web (como validar formularios o crear efectos visuales básicos), pero su evolución ha sido monumental. Hoy en día, gracias a entornos de ejecución como **Node.js**, JavaScript ha trascendido el navegador y se utiliza también en el desarrollo de servidores (backend), aplicaciones móviles, de escritorio e incluso en el Internet de las Cosas (IoT).

Características Principales

- **Lenguaje Interpretado:** A diferencia de los lenguajes compilados (como C++ o Java), el código JavaScript es procesado línea por línea por el navegador en tiempo de ejecución. Esto facilita el desarrollo y la depuración.
- **Tipado Débil y Dinámico:** Las variables en JavaScript no están atadas a un tipo de dato específico. Una variable puede contener un número y más tarde un texto sin generar un error.
- **Orientado a Objetos:** Aunque su enfoque es diferente al de lenguajes como Java o C#, JavaScript está basado en prototipos, lo que le permite tener objetos, herencia y polimorfismo.
- **Multiparadigma:** Soporta diferentes estilos de programación, incluyendo la programación orientada a objetos, imperativa y funcional.

La Evolución a través de ECMAScript: El Estándar detrás del Lenguaje

Una de las razones de la confusión inicial sobre JavaScript y su poderío fue su caótica "guerra de navegadores" en los años 90. Tanto Netscape (con JavaScript) como Microsoft (con su propia versión llamada JScript) implementaban características de forma independiente, lo que llevaba a que las páginas web funcionaran en un navegador, pero no en otro.

Para solucionar este problema de incompatibilidad, se decidió estandarizar el lenguaje. En 1997, JavaScript fue presentado a **ECMA International**, una organización de estandarización, y la especificación resultante se denominó **ECMAScript (ES)**.

ECMAScript no es el lenguaje, sino el estándar o la especificación que el lenguaje JavaScript debe seguir. Los navegadores (Chrome, Firefox, Safari) son los responsables de implementar motores de JavaScript (como V8, SpiderMonkey) que cumplan con dicha especificación. Esto garantiza que el código JavaScript se comporte de la misma manera en todos los entornos.

Desde su creación, ECMAScript ha evolucionado a través de varias versiones, siendo algunas de las más importantes:

- **ES3 (1999):** La base del JavaScript "clásico" durante casi una década. Introdujo expresiones regulares y el bloque try/catch.
- **ES5 (2009):** Una actualización masiva que modernizó el lenguaje. Añadió el "Modo Estricto" ('use strict';), soporte para JSON, y nuevos métodos para arrays como forEach(), map(), filter() y reduce (), que sentaron las bases para una programación más funcional.
- **ES6 / ECMAScript 2015 (2015):** La mayor actualización en la historia de JavaScript y el punto de inflexión que lo consolidó como un lenguaje moderno y robusto para aplicaciones a gran escala. Introdujo cambios sintácticos y conceptuales enormes, entre ellos:
    - **let y const:** Nuevas formas de declarar variables con ámbito de bloque, solucionando problemas históricos de var.
    - **Funciones Flecha (=>):** Una sintaxis más concisa para escribir funciones que además maneja el contexto (this) de una manera más intuitiva.

    - **Clases (class):** Una forma más limpia y familiar de implementar la programación orientada a objetos sobre el sistema de prototipos de JavaScript.
    - **Promesas (Promise):** Un objeto para manejar operaciones asíncronas de una forma mucho más elegante que los callbacks (ver sección de asincronía).
    - **Módulos (import/export):** Un sistema nativo para organizar el código en archivos reutilizables.
    - **Desestructuración** y **Parámetros por defecto**.
- **ES2016 en adelante (ESNext):** Desde 2015, ECMA adoptó un ciclo de lanzamientos anual. Cada año se publican nuevas características que han sido aprobadas por el comité técnico TC39. Esto permite una evolución más gradual y constante del lenguaje. Características notables post-ES6 incluyen async/await, el operador de propagación para objetos (...), y Promise.prototype.finally().

Esta constante evolución es lo que ha permitido a JavaScript pasar de ser un simple lenguaje de scripting a la potente herramienta que es hoy, capaz de impulsar complejas aplicaciones del lado del cliente y del servidor.

¿Es JavaScript un Lenguaje de Programación?

Sí, sin lugar a duda, **JavaScript es un lenguaje de programación completo y de alto nivel**.

A menudo surge la confusión porque, en sus inicios, se le asociaba únicamente con tareas sencillas en el navegador, en contraste con lenguajes "más serios" como Java o C++ que se usaban para crear aplicaciones complejas. Sin embargo, esta visión está completamente desactualizada.

Un lenguaje de programación se define como un sistema formal que, mediante un conjunto de instrucciones, permite a un programador controlar el comportamiento físico y lógico de una máquina. JavaScript cumple con creces esta definición:

1. **Tiene una sintaxis y semántica definidas:** Posee un conjunto de reglas gramaticales para escribir instrucciones válidas.
2. **Maneja variables y estructuras de datos:** Permite almacenar y manipular información en memoria a través de variables, arrays, objetos, etc.
3. **Implementa lógica y algoritmos:** A través de sus estructuras de control (condicionales y bucles), permite crear algoritmos complejos para la toma de decisiones y la repetición de tareas.
4. **Es "Turing completo":** Esto significa que puede ser utilizado para resolver cualquier problema computacional que pueda ser resuelto por una máquina de Turing, la definición teórica de un computador.

Por lo tanto, aunque su ámbito de aplicación principal es la web, su capacidad para procesar datos, ejecutar lógica compleja y controlar el comportamiento de un entorno (el navegador, un servidor) lo califica plenamente como un lenguaje de programación robusto y versátil.

Clasificación de sus Comandos (Instrucciones)

Los "comandos" en JavaScript se conocen formalmente como **sentencias** o **instrucciones** (*statements*). Estas son las unidades de código que realizan una acción. Se pueden clasificar de varias maneras, pero una de las más útiles es por su función:

1. Declaración

Se utilizan para declarar variables y constantes, que son los contenedores de datos.

- var: Declara una variable (con ámbito de función).
- let: Declara una variable (con ámbito de bloque, introducida en ES6).
- const: Declara una constante, cuyo valor no puede ser reasignado (ámbito de bloque).
- function: Declara una función.
- class: Declara una clase para la programación orientada a objetos.

2. Asignación

Se utilizan para asignar un valor a una variable. El operador principal es =.

- x = 10;
- nombre = "Ana";

3. Estructuras de Control

Dirigen el flujo de ejecución del programa. Se explican en detalle en la siguiente sección.

- Condicionales (if, else, else if, switch).
- Bucles (for, while, do...while, for...in, for...of).
- Manejo de errores (try, catch, finally, throw).

4. Llamadas a Funciones

Ejecutan el código contenido dentro de una función.

- alert("Hola Mundo");
- console.log(miVariable);
- miFuncion(parametro1, parametro2);

5. Expresiones

Cualquier fragmento de código que produce un valor. Una expresión puede ser tan simple como un número (42) o tan compleja como una combinación de variables y operadores ((x * y) / 2).

Estructuras de Control que Utiliza

Las estructuras de control son fundamentales en cualquier lenguaje de programación, ya que permiten que un programa tome decisiones y repita acciones. JavaScript implementa las estructuras de control estándar.

Condicionales

Permiten ejecutar bloques de código solo si se cumplen ciertas condiciones.

- **if / else / else if**: Es la estructura condicional más común. Evalúa una condición y ejecuta un bloque de código si es verdadera. Opcionalmente, puede ejecutar otros bloques si la primera condición es falsa (else) o si se cumplen otras condiciones (else if).

```
let edad = 18;
if (edad >= 18) {
  console.log("Es mayor de edad.");
} else {
  console.log("Es menor de edad.");
}
```

- **switch**: Permite evaluar una expresión y ejecutar diferentes bloques de código basados en el valor de esta. Es útil cuando hay múltiples casos posibles para una sola variable.

```
let dia = new Date().getDay(); // Devuelve un número del 0 (Domingo) al 6 (Sábado)
switch (dia) {
  case 0:
    console.log("Hoy es Domingo.");
    break;
  case 6:
    console.log("Hoy es Sábado.");
    break;
  default:
    console.log("Es un día de semana.");
}
```

Bucles (Loops)

Permiten repetir un bloque de código varias veces.

- **for**: Repite un bloque de código un número determinado de veces. Es ideal cuando se sabe de antemano cuántas iteraciones se necesitan.

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

- **while**: Repite un bloque de código mientras una condición especificada sea verdadera. La condición se evalúa antes de cada iteración.

```
let contador = 0;
while (contador < 3) {
  console.log("Contador: " + contador);
  contador++;
}
```

- **do...while**: Similar a while, pero garantiza que el bloque de código se ejecute al menos una vez, ya que la condición se evalúa *después* de la iteración.

```
let i = 10;
do {
  console.log(i);
  i++;
} while (i < 5);
```

¿Qué es el DOM?

El **DOM (Document Object Model)**, o Modelo de Objetos del Documento, es una interfaz de programación para documentos HTML y XML. Representa la estructura de una página web como un árbol de objetos, donde cada nodo del árbol corresponde a una parte del documento (un elemento, un atributo, un texto, etc.).

Cuando un navegador web carga una página HTML, no la lee como un texto plano. En su lugar, crea un modelo en memoria de esa página: el DOM. Este modelo es una representación viva del documento, lo que significa que puede ser manipulado.

JavaScript es el lenguaje que nos permite interactuar y manipular el DOM.

Gracias al DOM, con JavaScript podemos:

- **Leer contenido:** Obtener el texto de un párrafo o el valor de un campo de formulario.
- **Modificar contenido:** Cambiar el texto de un encabezado o la imagen de una etiqueta <img>.
- **Cambiar estilos:** Modificar el color de fondo de un elemento, su tamaño o su visibilidad.
- **Crear y eliminar elementos:** Añadir nuevos párrafos, botones o divs a la página, o quitarlos dinámicamente.
- **Reaccionar a eventos:** Ejecutar código cuando un usuario hace clic en un botón, mueve el ratón o presiona una tecla.

La API del DOM: Selección y Manipulación Avanzada

La "interfaz de programación" que el DOM ofrece se conoce como la **API del DOM**. Consiste en un gran conjunto de objetos y métodos que JavaScript puede utilizar. Los más importantes se centran en seleccionar elementos de la página y manipularlos.

1. Métodos de Selección de Elementos: JavaScript necesita una forma de "apuntar" a un nodo específico del árbol DOM para poder leerlo o modificarlo.

- **document.getElementById('id-del-elemento')**: El método más rápido y específico. Devuelve el único elemento que coincide con el ID proporcionado.
- **document.getElementsByClassName('nombre-clase')**: Devuelve una *colección* de elementos (similar a un array) que tienen la clase CSS especificada.

- **document.getElementsByTagName('nombre-etiqueta')**: Devuelve una *colección* de elementos que coinciden con el nombre de la etiqueta HTML (ej. 'p' para todos los párrafos).
- **document.querySelector('selector-css')**: Un método moderno y muy potente. Devuelve el **primer** elemento que coincide con el selector CSS especificado. El selector puede ser simple ('#titulo') o complejo ('div.card > p').
- **document.querySelectorAll('selector-css')**: Similar al anterior, pero devuelve una *colección* de **todos** los elementos que coinciden con el selector CSS.

2. Métodos de Manipulación de Elementos: Una vez que un elemento ha sido seleccionado y guardado en una variable, podemos cambiarlo.

- **Modificar Contenido:**
    - elemento.textContent = 'Nuevo texto';: Cambia el contenido de texto de un nodo, ignorando cualquier HTML. Es más seguro y a menudo más rápido.
    - elemento.innerHTML = 'Texto en negrita';: Cambia el HTML interno de un elemento. Permite insertar nuevas etiquetas HTML, pero debe usarse con cuidado para evitar vulnerabilidades de seguridad (XSS).
- **Modificar Atributos:**
    - elemento.setAttribute('class', 'nueva-clase');: Establece o cambia el valor de un atributo.
    - elemento.getAttribute('href');: Obtiene el valor actual de un atributo.
    - elemento.removeAttribute('disabled');: Elimina un atributo.
- **Modificar Estilos:**
    - elemento.style.color = 'red';
    - elemento.style.backgroundColor = '#f0f0f0';
    - elemento.classList.add('clase-activa');: Añade una clase CSS.
    - elemento.classList.remove('clase-inactiva');: Remueve una clase CSS.
    - elemento.classList.toggle('visible');: Añade la clase si no la tiene, y la quita si la tiene.

3. Creación y Eliminación de Nodos: El DOM no es estático; podemos alterarlo por completo.

- **document.createElement('nombre-etiqueta')**: Crea un nuevo nodo de elemento en memoria (ej. document.createElement('p')).
- **nodoPadre.appendChild(nuevoNodo)**: Añade el nuevoNodo como el último hijo del nodoPadre.
- **nodoPadre.insertBefore(nuevoNodo, nodoReferencia)**: Inserta el nuevoNodo antes del nodoReferencia dentro del nodoPadre.
- **nodoPadre.removeChild(nodoHijo)**: Elimina un nodoHijo del nodoPadre.

Este control granular sobre cada parte del documento es lo que permite a las aplicaciones web modernas ser tan dinámicas y receptivas.

¿Qué es un Framework?

Un **framework** (marco de trabajo) es una estructura conceptual y tecnológica de soporte definido, normalmente con artefactos o módulos de software concretos, que sirve como base para la organización y desarrollo de software.

Imagina que quieres construir una casa. Podrías empezar desde cero: talar árboles, hacer ladrillos, diseñar los planos, etc. O podrías usar un kit de casa prefabricada. Este kit te proporciona los cimientos, las paredes, el techo y una estructura básica (el "marco"). Tú todavía tienes que ensamblarlo, pintarlo, decorarlo y añadir tu toque personal, pero no tienes que construir cada componente desde cero.

Un framework de software es como ese kit:

- **Proporciona una estructura:** Ofrece un esqueleto para tu aplicación.
- **Impone una arquitectura:** Guía la forma en que organizas tu código, promoviendo buenas prácticas.
- **Ofrece funcionalidades preconstruidas:** Incluye librerías y herramientas para tareas comunes (como gestionar rutas en una web, manejar el estado de la aplicación, o comunicarse con un servidor), para que no tengas que "reinventar la rueda".

La principal diferencia con una **librería** es que tu código "llama" a una librería para usar sus funciones, mientras que un framework "llama" a tu código. El framework tiene el control del flujo de la aplicación (Inversión de Control) y te dice dónde debes colocar tu lógica específica.

Frameworks de JavaScript más Utilizados

El ecosistema de JavaScript es famoso por su gran cantidad de frameworks y librerías. Los tres más populares y dominantes en el desarrollo front-end moderno son:

1. React (a menudo llamado React.js o ReactJS)

Desarrollado y mantenido por Meta (antes Facebook), React no es estrictamente un framework, sino una **librería** para construir interfaces de usuario. Sin embargo, se utiliza como la base de muchas arquitecturas de aplicaciones, por lo que a menudo se le menciona en esta categoría. Su principal característica es el uso de un **DOM Virtual** para optimizar las actualizaciones de la interfaz y su **arquitectura basada en componentes**, que permite crear piezas de UI reutilizables.

2. Angular

Desarrollado y mantenido por Google, Angular es un **framework completo y robusto** para crear aplicaciones web complejas y a gran escala. Es mucho más "opinado" que React, lo que significa que impone una estructura más rígida y proporciona soluciones integradas para casi todo (manejo de formularios, enrutamiento, peticiones HTTP, etc.). Utiliza el lenguaje **TypeScript**, un superconjunto de JavaScript que añade tipado estático.

3. Vue.js

Creado por Evan You (un ex-ingeniero de Google), Vue.js es a menudo visto como un punto intermedio entre React y Angular. Es un **framework progresivo**, lo que significa que puedes adoptarlo gradualmente. Es conocido por su curva de aprendizaje suave, su excelente documentación y su flexibilidad. Combina la reactividad y la arquitectura de componentes de React con una sintaxis de plantillas más cercana al HTML tradicional, similar a Angular.

JavaScript Asíncrono: Manejando el Tiempo y las Operaciones Externas

Por naturaleza, JavaScript es un lenguaje de un solo hilo (single-threaded). Esto significa que solo puede hacer una cosa a la vez. En el contexto de un navegador, si JavaScript está ocupado ejecutando una tarea larga y pesada (como procesar una gran cantidad de datos o esperar la respuesta de un servidor), la interfaz de usuario se congelará. El usuario no podrá hacer clic en botones, desplazarse por la página ni interactuar de ninguna manera.

Para solucionar este problema, JavaScript utiliza un modelo asíncrono. Permite que las operaciones de larga duración (como peticiones a una API, lecturas de archivos o temporizadores) se inicien y se ejecuten en segundo plano sin bloquear el hilo principal. Cuando la operación termina, el resultado se maneja. Esto es posible gracias al Bucle de Eventos (Event Loop) del navegador.

A lo largo de los años, las técnicas para manejar la asincronía han evolucionado:

1. Callbacks (Retrollamadas)

El método original. Un callback es simplemente una función que se pasa como argumento a otra función, con la expectativa de que la función callback se ejecute después de que la operación asíncrona haya terminado.

```
function obtenerDatos(callback) {
  setTimeout(() => { // setTimeout simula una petición de red
    const datos = { id: 1, nombre: 'Producto A' };
    callback(datos); // Se ejecuta la función de callback con los datos
  }, 2000); // Espera 2 segundos
}

obtenerDatos(function(misDatos) {
  console.log('Datos recibidos:', misDatos);
});
```

El problema surge cuando se necesitan realizar múltiples operaciones asíncronas en secuencia. Esto conduce al infame "Callback Hell" o "Pyramid of Doom", donde el código se anida cada vez más, volviéndose muy difícil de leer y mantener.

2. Promesas (Promises)

Introducidas en ES6, las promesas son objetos que representan la eventual finalización (o fallo) de una operación asíncrona. Una promesa tiene tres estados:

- **pending:** Estado inicial, la operación aún no ha terminado.
- **fulfilled:** La operación se completó con éxito.
- **rejected:** La operación falló.

Las promesas permiten encadenar operaciones asíncronas de una manera mucho más limpia usando los métodos .then() (para éxito) y .catch() (para error).

```
function obtenerDatosConPromesa() {
 return new Promise((resolve, reject) => {
  setTimeout(() => {
   const exito = true; // Simular éxito o fallo
   if (exito) {
    const datos = { id: 2, nombre: 'Producto B' };
    resolve(datos); // La promesa se cumple con los datos
   } else {
    reject('Error al obtener los datos.'); // La promesa es rechazada
   }
  }, 2000);
 });
}


obtenerDatosConPromesa()
 .then(datos => {
  console.log('Datos recibidos:', datos);
  // Se puede encadenar otra promesa aquí
 })
 .catch(error => {
  console.error('Hubo un error:', error);
 });
```

3. Async/Await

Introducido en ES2017, async/await es "azúcar sintáctico" sobre las promesas. No introduce una nueva funcionalidad, pero permite escribir código asíncrono que se ve y se comporta como código síncrono, haciéndolo mucho más fácil de leer y razonar.

- La palabra clave async se coloca antes de una función. Indica que la función siempre devolverá una promesa.
- La palabra clave await solo puede usarse dentro de una función async. Pausa la ejecución de la función y espera a que una promesa se resuelva (se cumpla o se rechace), y luego reanuda la ejecución con el resultado.

```
async function procesarDatos() {
 try {
  console.log('Iniciando petición de datos...');
  // El código se pausa aquí hasta que la promesa se resuelva
  const datos = await obtenerDatosConPromesa();
  console.log('Datos procesados:', datos);

  // Podríamos hacer otro 'await' aquí para otra operación
  // const otrosDatos = await obtenerOtrosDatos(datos.id);

  return datos; // El valor de retorno se envuelve en una promesa resuelta
 } catch (error) {
  console.error('Error en el procesamiento:', error);
 }
}

procesarDatos();
```

El manejo de la asincronía es fundamental para crear aplicaciones web fluidas y eficientes, y el paso de callbacks a promesas y finalmente a async/await representa una de las mejoras más significativas en la ergonomía del lenguaje JavaScript.

Ejemplos de Código en JavaScript

Ejemplo 1: Manipulación del DOM

Este código cambia el contenido de un encabezado HTML cuando se hace clic en un botón.

HTML:

```
<!DOCTYPE html>
<html>
<head>
 <title>Ejemplo DOM</title>
</head>
<body>
 <h1 id="tituloPrincipal">Hola Mundo</h1>
 <button onclick="cambiarTexto()">Cambiar Saludo</button>

 <script src="app.js"></script>
</body>
</html>
```

JavaScript (app.js):

```
function cambiarTexto() {
 const titulo = document.getElementById('tituloPrincipal');
 titulo.textContent = '¡JavaScript en acción!';
 titulo.style.color = 'blue';
}
```

Ejemplo 2: Uso de Estructuras de Control

Esta función recibe un array de números y devuelve un nuevo array solo con los números pares que son mayores que 10.

JavaScript:

```
function filtrarNumeros(numeros) {
 const resultado = []; // Array vacío para guardar los resultados

 for (const numero of numeros) {
```

```
  if (numero > 10 && numero % 2 === 0) {
    resultado.push(numero);
  }
 }

 return resultado;
}

const listaDeNumeros = [5, 12, 8, 130, 44, 9, 21];
const numerosFiltrados = filtrarNumeros(listaDeNumeros);

console.log(numerosFiltrados); // Resultado esperado: [12, 130, 44]
```

Ejemplo 3: Contador Interactivo

Este ejemplo crea un contador simple que aumenta cada vez que el usuario presiona un botón, mostrando cómo manejar eventos y actualizar el estado.

HTML:

```
<!DOCTYPE html>
<html>
<head>
 <title>Contador</title>
</head>
<body>
 <h2>Contador</h2>
 <p>Clics: <span id="valorContador">0</span></p>
 <button id="botonIncrementar">Incrementar</button>

 <script src="contador.js"></script>
</body>
</html>
```

JavaScript (contador.js):

```
let contador = 0;

const valorDisplay = document.getElementById('valorContador');
const boton = document.getElementById('botonIncrementar');

boton.addEventListener('click', function() {
 contador++;

 valorDisplay.textContent = contador;
});
```

BIBLIOGRAFÍA

- Mozilla. (s.f.). *Guía de JavaScript*. MDN Web Docs. Recuperado de https://developer.mozilla.org/es/docs/Web/JavaScript/Guide. MDN Web Docs
- Mozilla. (s.f.). *JavaScript — Documentación y referencias*. MDN Web Docs. Recuperado de https://developer.mozilla.org/es/docs/Web/JavaScript. MDN Web Docs
- Haverbeke, M. (2018/2024). *JavaScript elocuente* (ed. en español — traducción comunitaria). 3.ª/4.ª ed. Disponible en línea: https://eloquentjavascript.es/ (o PDF). eloquentjavascript.es+1
- Flanagan, D. (2020). *JavaScript: La guía definitiva* (ed. en español). O'Reilly / Anaya (edición en español). (Consultar edición y año exacto en la ficha editorial). O'Reilly Media+1
- Simpson, K. (s.f.). *Tú no sabes JS* (serie). Traducciones y adaptaciones al español (repositorios y ediciones comunitarias). Disponible en traducciones comunitarias (GitHub / GitBook). GitHub+1
- Osmani, A. (2012/2023). *Aprendiendo patrones de diseño en JavaScript* (Learning JavaScript Design Patterns). Ed. O'Reilly — ed. en español disponible en ediciones recientes; útil para patrones y organización de código. libreriaguadalajara.store+1
- midudev / comunidad (2024). *Traducción y edición en español de Eloquent JavaScript* (sitio y recursos de apoyo). https://eloquentjavascript.es/ (versión en español mantenida). eloquentjavascript.es
- W3Schools (s.f.). *Tutorial de JavaScript* (material didáctico con ejemplos, disponible en múltiples idiomas y traducciones). Recuperado de https://www.w3schools.com/js/. W3Schools