

Contenido

Introducción	2
¿Qué tipado utiliza JavaScript?	2
Fundamentos JavaScript	3
¿Cuál es el propósito de TypeScript?	7
Características	8
¿Cómo ha logrado posicionarse tanto como lenguaje de servidor y cliente?	13

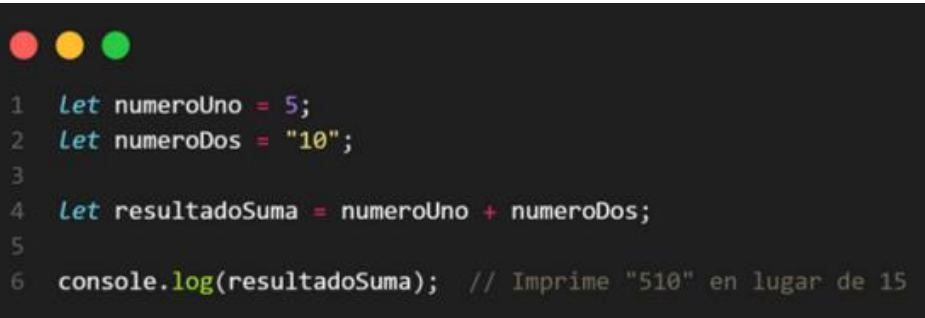
Introducción

El nacimiento de JavaScript se sitúa en 1995, cuando Netscape, un navegador ya obsoleto, experimentó desafíos al interactuar con documentos HTML. Ante esta limitación, **Brendan Eich** creó JavaScript específicamente para Netscape, habilitando una mayor interactividad en los navegadores al superar la restricción de solo enviar datos mediante formularios.

¿Qué tipado utiliza JavaScript?

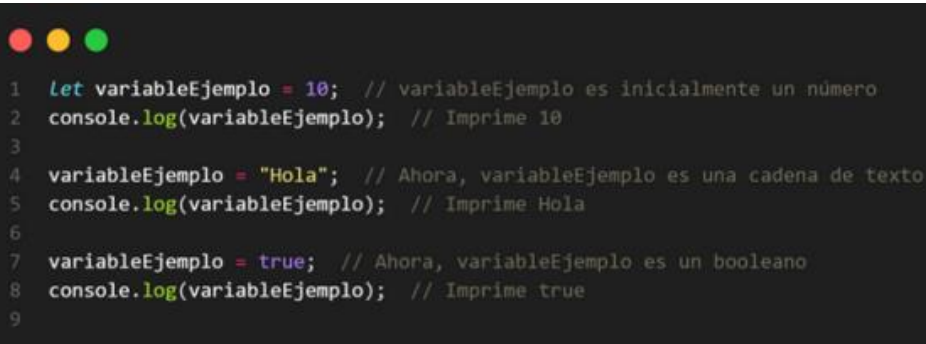
JavaScript es un lenguaje de programación débilmente tipado y dinámicamente tipado, lo que significa que no es necesario especificar el tipo de variable al declararla, y el tipo de datos de una variable puede cambiar durante la ejecución del programa.

- **Tipado débil:** Como se menciona en el párrafo anterior, no necesitamos especificar el tipo de variable. JavaScript permite que una variable comience almacenando un número, luego un string y, finalmente, un boolean, lo que ilustra la ventaja del tipado dinámico.



```
1  let numeroUno = 5;
2  let numeroDos = "10";
3
4  let resultadoSuma = numeroUno + numeroDos;
5
6  console.log(resultadoSuma); // Imprime "510" en lugar de 15
```

- **Tipado dinámico:** Como se menciona en el párrafo anterior, no necesitamos especificar el tipo de variable. JavaScript permite que una variable comience almacenando un número, luego un string y, finalmente, un boolean, lo que ilustra la ventaja del tipado dinámico.



```
1  let variableEjemplo = 10; // variableEjemplo es inicialmente un número
2  console.log(variableEjemplo); // Imprime 10
3
4  variableEjemplo = "Hola"; // Ahora, variableEjemplo es una cadena de texto
5  console.log(variableEjemplo); // Imprime Hola
6
7  variableEjemplo = true; // Ahora, variableEjemplo es un booleano
8  console.log(variableEjemplo); // Imprime true
9
```

Fundamentos JavaScript

Los fundamentos de JavaScript comprenden los conceptos y principios esenciales que un desarrollador debe dominar para trabajar de manera efectiva con este lenguaje de programación. En esta discusión, nos centraremos especialmente en los aspectos más relevantes, aquellos que sientan las bases necesarias para avanzar posteriormente hacia tecnologías como TypeScript o frameworks como React.

DOM

El DOM es una interfaz que proporciona una representación estructurada de un documento web, permitiendo a los desarrolladores interactuar con el contenido de la página de manera dinámica y responder a eventos del usuario.

Ejemplo Práctico:

```
1 // Espera a que el contenido HTML haya sido completamente cargado
2 document.addEventListener("DOMContentLoaded", function() {
3     // Obtén una referencia al elemento con el id "titulo"
4     var tituloElement = document.getElementById("titulo");
5
6     // Cambia el contenido del elemento
7     tituloElement.innerHTML = "¡Hola, Mundo modificado con JavaScript!";
8
9     // Agrega un evento al botón con el id "cambiarTexto"
10    var botonElement = document.getElementById("cambiarTexto");
11    botonElement.addEventListener("click", function() {
12        // Cambia el contenido del elemento cada vez que se hace clic en el botón
13        tituloElement.innerHTML = "¡Texto cambiado!";
14    });
15 });
16
```

En este ejemplo:

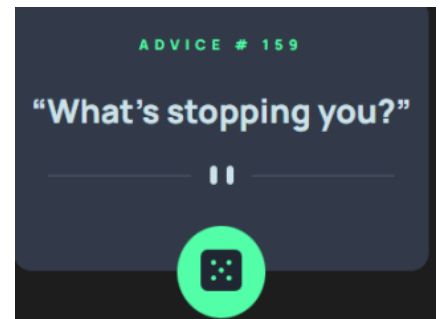
1. Se añade al documento HTML el evento DOMContentLoaded que, al completarse la carga del documento, ejecuta lo contenido dentro de la función anónima.
2. Se declara la variable tituloElemento para almacenar la etiqueta con el id "titulo". Después, se utiliza innerHTML para asignarle el valor especificado en la línea 7 del código.
3. Finalmente, se crea la variable botonElement para almacenar la etiqueta del botón con el id 'cambiarTexto'. Se añade un evento 'click' al botón, de modo que, al hacer clic, se cambia el texto del elemento almacenado en tituloElement.

Funciones

Las funciones en JavaScript son bloques de código reutilizables que realizan tareas específicas. En el contexto de trabajar con APIs, es esencial entender y utilizar la función `fetch` para hacer solicitudes HTTP y obtener datos de servicios externos. Esto permite crear aplicaciones dinámicas que interactúan con información en tiempo real desde APIs.

Ejemplo Práctico:

```
1 // Variables
2 var titulo = document.querySelector(".card-title-id");
3 var textInfo = document.querySelector(".card-text");
4 var footerElement = document.querySelector(".card-footer");
5 var btnCube = document.querySelector('.cube');
6 // Eventos
7 window.addEventListener('DOMContentLoaded', obtenerDatosDeAPI)
8 btnCube.addEventListener('click', obtenerDatosDeAPI);
9
10 // Funciones
11 function obtenerDatosDeAPI() {
12     const apiUrl = 'https://api.adviceslip.com/advice';
13
14     fetch(apiUrl)
15     .then(response => {
16         if (!response.ok) {
17             throw new Error('Error en la solicitud a la API');
18         }
19         return response.json();
20     })
21     .then(data => {
22         const { slip: { id, advice } } = data;
23         titulo.textContent = `ADVICE # ${id}`;
24         textInfo.textContent = advice;
25     })
26     .catch(error => {
27         console.error('Error:', error);
28     });
29 }
```



```
1 'Salida de la API: { slip:
2   { id: 74,
3     advice: \"Work is never as important as you think it is.\" } }'
```

En este ejemplo:

1. Se asignan a las variables `título`, `textInfo`, `footeElement` y `btnCube` las etiquetas correspondientes que se encuentran en la imagen, identificadas con la clase como `'card-title-id'`.
2. La línea 7 indica que, al completarse la carga del DOM, se ejecutará la función `obtenerDatosDeAPI`.

3. Dentro de la función obtenerDatosDeAPI, la variable apiURL almacena la URL de la API para luego recuperar la información correspondiente.
4. La respuesta de la API se visualiza en la segunda imagen, donde el texto aparece con letras de color morado.
5. Finalmente, mostramos la información obtenida al consumir la API, presentada de manera gráfica en la tercera imagen.

Asincronía

La asincronía en JavaScript se refiere a la capacidad del lenguaje para realizar operaciones sin bloquear la ejecución del resto del código. Esto se logra mediante el uso de funciones asíncronas y conceptos como promesas y `async/await`. Al trabajar con APIs, donde las respuestas pueden demorar, comprender la asincronía es esencial. Permite realizar solicitudes a APIs sin interrumpir el flujo principal del programa, asegurando una experiencia de usuario más eficiente y receptiva.

Ejemplo Práctico:

```
1 // Variables
2 var titleElement = document.querySelector(".card-title-id");
3 var textElement = document.querySelector(".card-text");
4 var footerElement = document.querySelector(".card-footer");
5 var botonCube = document.querySelector(".cube");
6
7 // Eventos
8 window.addEventListener('DOMContentLoaded', () => {
9     obtenerDatosDeAPI();
10 });
11
12 // Funciones
13 async function obtenerDatosDeAPI() {
14     const apiUrl = 'https://api.adviceslip.com/advice';
15
16     try {
17         const response = await fetch(apiUrl);
18
19         if (!response.ok) {
20             throw new Error('Error en la solicitud a la API');
21         }
22
23         const data = await response.json();
24         const { slip: { id } } = data;
25
26         // Actualizar el contenido de las etiquetas HTML con los datos de la API
27         titleElement.textContent = `ADVICE # ${id}`;
28         textElement.textContent = data.slip.advice;
29     } catch (error) {
30         console.error('Error:', error);
31     }
32 }
```

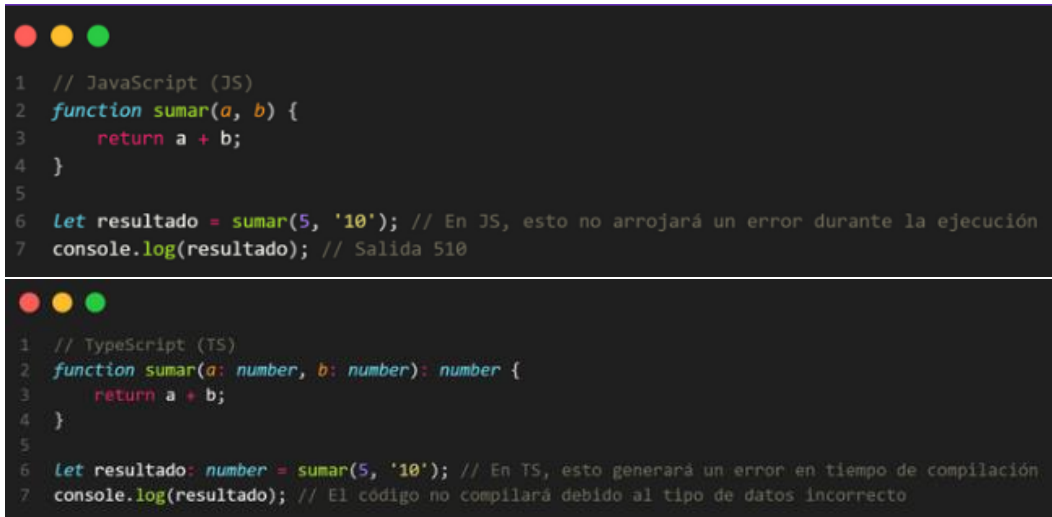
En este ejemplo:

1. obtenerDatosDeAPI utiliza 'async/await para manejar la asincronía de la operación.
2. Luego, en la variable response, aguardamos la respuesta de la API mediante fetch (nos permite leer datos de la API).
3. Si la respuesta es correcta, aguardamos el resultado de la API en la variable data.
4. Por último, realizamos una deestructuración del objeto para luego incorporarlo en el DOM.

¿Cuál es el propósito de TypeScript?

TypeScript es un lenguaje de programación de código abierto desarrollado por Microsoft. Fue creado para abordar ciertas limitaciones de JavaScript, como la carencia de tipado fuerte, que podría dar lugar a errores sutiles difíciles de detectar durante el desarrollo.

Ejemplo Práctico:



```
1 // JavaScript (JS)
2 function sumar(a, b) {
3   return a + b;
4 }
5
6 let resultado = sumar(5, '10'); // En JS, esto no arrojará un error durante la ejecución
7 console.log(resultado); // Salida 510
```

```
1 // TypeScript (TS)
2 function sumar(a: number, b: number): number {
3   return a + b;
4 }
5
6 let resultado: number = sumar(5, '10'); // En TS, esto generará un error en tiempo de compilación
7 console.log(resultado); // El código no compilará debido al tipo de datos incorrecto
```

En este ejemplo:

1. En la primera imagen, observamos un ejemplo típico de una función en JavaScript diseñada para sumar. Esta función recibe dos parámetros, lo que le permite manejar una variedad de tipos de datos, ya sean strings, números decimales (double), booleanos, entre otros.
2. En la segunda imagen, encontramos el código en TypeScript, un lenguaje conocido por su tipado estático. En la función 'sumar', resalta la especificación de tipos para sus parámetros, indicando el tipo de valor que deben recibir. Además, la declaración de retorno asegura que la función devuelva estrictamente un número.

Características

TypeScript (TS) es un superset de JavaScript que agrega características de tipado estático al lenguaje. Aquí hay algunas características relevantes de TypeScript.

- **Tipado Estático:**
 - TypeScript introduce un sistema de tipos estático que permite declarar tipos para variables, parámetros de función y valores de retorno.
 - El tipado estático ayuda a detectar errores en tiempo de compilación en lugar de en tiempo de ejecución, lo que puede facilitar el desarrollo y la depuración.
- **Orientado a objetos:**
 - TypeScript es un lenguaje de programación que favorece la programación orientada a objetos (OOP), proporcionando herramientas que facilitan el desarrollo estructurado.
- **Interfaces:**
 - TypeScript permite la definición de interfaces que pueden ser implementadas por clases. Esto proporciona una forma de definir contratos y estructuras para las clases
- **Modularidad y Espacios de Nombres:**
 - TypeScript admite la modularidad mediante la organización del código en módulos. Los módulos ayudan a dividir el código en partes más pequeñas y lógicas, facilitando la mantenibilidad y la reutilización
 - Los espacios de nombres proporcionan una forma de organizar lógicamente el código, agrupando elementos relacionados bajo un mismo nombre. Esto ayuda a evitar colisiones de nombres y a mejorar la estructura del código.

Ejemplo Práctico:

- **Tipado Estático**

```
1 // JavaScript
2
3 // Función que saluda a una persona
4 function saludar(name) {
5     return `Hello, ${name}!`;
6 }
7
8 // Uso de la función
9 let saludo = saludar("Chris");
10
11 // Imprimir el saludo
12 console.log(saludar);
13
```

```
1 // TypeScript
2
3 // Función que saluda a una persona
4 function saludar(name: string): string {
5     return `Hello, ${name}!`;
6 }
7
8 // Uso de la función
9 let saludo: string = saludar("Chris");
10
11 // Imprimir el saludo
12 console.log(saludo);
13
```

En este ejemplo:

1. En la primera imagen, se muestra una función en JavaScript que acepta un argumento sin conocimiento previo sobre su tipo de dato, debido al tipado dinámico. Esta característica, aunque flexible, puede ser problemática, ya que la función puede recibir inadvertidamente tipos de datos no deseados, generando posibles errores difíciles de detectar durante el desarrollo.
2. En la segunda imagen, la misma función se presenta, pero con la especificación de que el argumento debe ser un string, y el tipo de retorno también está limitado a un string. Esta restricción proporciona mayor claridad y control sobre el tipo de datos que la función puede manejar, reduciendo la posibilidad de errores y mejorando la robustez del código

- **Orientado a objetos**

```
1 // Definición de la clase Perro
2 class Perro {
3     // Propiedades
4     nombre: string;
5     edad: number;
6     raza: string;
7
8     // Constructor
9     constructor(nombre: string, edad: number, raza: string) {
10         this.nombre = nombre;
11         this.edad = edad;
12         this.raza = raza;
13     }
14
15     // Método para ladrar
16     ladrar(): void {
17         console.log("¡Guau, guau!");
18     }
19
20     // Método para mostrar información del perro
21     mostrarInformacion(): void {
22         console.log(`Nombre: ${this.nombre}, Edad: ${this.edad}, Raza: ${this.raza}`);
23     }
24 }
25
26 // Crear una instancia de la clase Perro
27 const miPerro = new Perro("Fido", 3, "Labrador");
28
29 // Llamar a métodos y acceder a propiedades
30 miPerro.ladrar();
31 miPerro.mostrarInformacion();
```

En este ejemplo:

1. En este ejemplo, hemos definido una clase Perro con propiedades como nombre, edad y raza.
2. El constructor se encarga de inicializar estas propiedades cuando creamos una instancia de la clase. También hemos agregado dos métodos: ladrar y mostrarInformacion.

- Interfaces

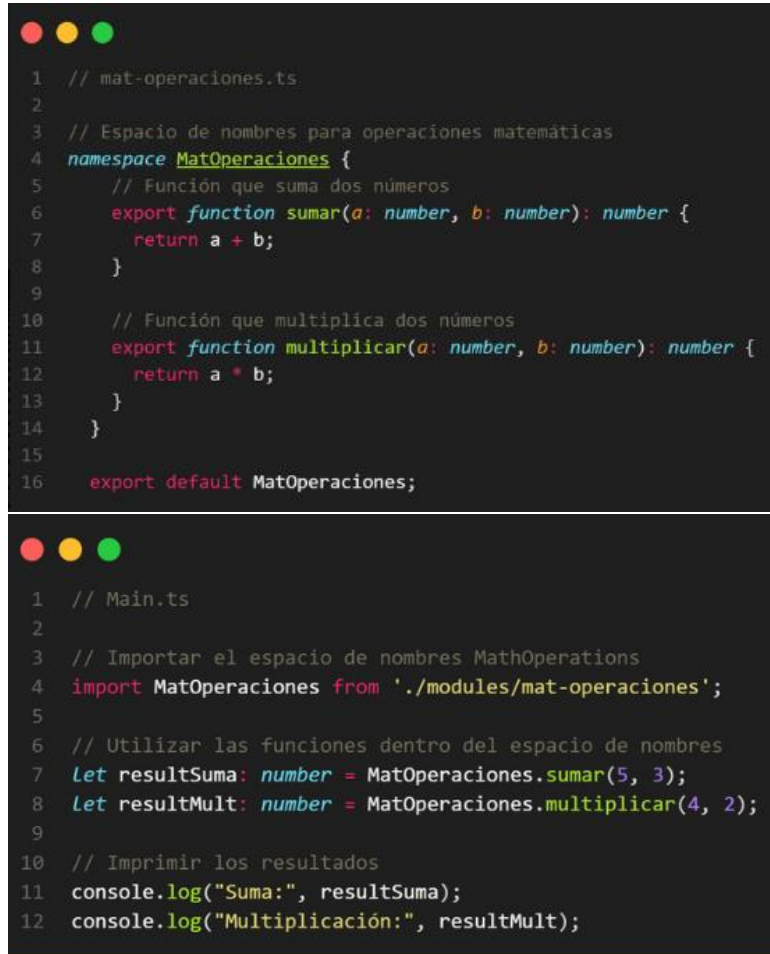
```
1 // Definición de una interfaz para representar a una persona
2 interface Persona {
3     nombre: string;
4     apellido: string;
5     saludar: () => string; // Función que devuelve una cadena
6 }
```

```
1 import Persona from './interfaces/interface';
2
3
4 // Implementación de la interfaz en un objeto
5 let persona: Persona = {
6     nombre: "Juan",
7     apellido: "Pérez",
8     saludar: function() {
9         return `Hola, mi nombre es ${this.nombre} ${this.apellido}.`;
10    }
11 };
12
13 // Uso de la función saludar
14 let saludo: string = persona.saludar();
15
16 // Imprimir el saludo
17 console.log(saludo);
18
```

En este ejemplo:

1. En la primera imagen se presenta una interfaz que funciona como una especie de plantilla. Esta plantilla define el comportamiento que posteriormente adoptará el objeto al que se le asigna.
2. En la segunda imagen, la interfaz 'Persona' se asigna a nuestro objeto 'persona', y posteriormente definimos los atributos de acuerdo con la estructura especificada por la interfaz.

- **Modularidad y Espacios de Nombres**



```
1 // mat-operaciones.ts
2
3 // Espacio de nombres para operaciones matemáticas
4 namespace MatOperaciones {
5     // Función que suma dos números
6     export function sumar(a: number, b: number): number {
7         return a + b;
8     }
9
10    // Función que multiplica dos números
11    export function multiplicar(a: number, b: number): number {
12        return a * b;
13    }
14 }
15
16 export default MatOperaciones;
```

```
1 // Main.ts
2
3 // Importar el espacio de nombres MathOperations
4 import MatOperaciones from './modules/mat-operaciones';
5
6 // Utilizar las funciones dentro del espacio de nombres
7 let resultSuma: number = MatOperaciones.sumar(5, 3);
8 let resultMult: number = MatOperaciones.multiplicar(4, 2);
9
10 // Imprimir los resultados
11 console.log("Suma:", resultSuma);
12 console.log("Multiplicación:", resultMult);
```

En este ejemplo:

1. En la primera imagen, se establece el espacio de nombres 'MatOperaciones', que contiene dos funciones, 'sumar' y 'multiplicar'. Posteriormente, exportamos el espacio de nombres 'MatOperaciones' para acceder y utilizar las funciones definidas en su interior.
2. En la segunda imagen, importamos las funciones ubicadas dentro del espacio de nombres 'MatOperaciones' y posteriormente accedemos a ellas como si fueran atributos del objeto.

¿Cómo ha logrado posicionarse tanto como lenguaje de servidor y cliente?

El éxito de TypeScript y JavaScript como lenguajes tanto en el lado del servidor como en el cliente se debe a varios factores clave.

- **Versatilidad de JavaScript:**
 - JavaScript es el lenguaje principal para el desarrollo web en el lado del cliente, y su popularidad ha crecido enormemente con el auge de las aplicaciones web interactivas, así como los framework que comporten sintaxis de JS.
- **Desarrollo de Aplicaciones del Lado del Servidor:**
 - Con la aparición de Node.js, JavaScript dejó de ser exclusivo del navegador y se convirtió en una opción viable para el desarrollo del lado del servidor. Su modelo de I/O no bloqueante y su naturaleza asíncrona fueron especialmente útiles en situaciones de alto rendimiento.
- **Adopción de TypeScript:**
 - TypeScript, un superset de JavaScript que agrega tipado estático, ha ganado popularidad gracias a su capacidad para mejorar la calidad del código, prevenir errores comunes y facilitar el mantenimiento de proyectos a gran escala.
- **Frameworks Populares:**
 - La adopción de frameworks populares como Angular, React y Vue.js, que utilizan JavaScript/TypeScript, ha contribuido significativamente a su éxito. Estos frameworks permiten el desarrollo rápido y estructurado de aplicaciones web complejas.

En conjunto, estos factores han creado un ecosistema robusto y atractivo que ha llevado a la amplia adopción de JavaScript/TypeScript tanto en el lado del servidor como en el cliente.

Referencias:

- <https://shorturl.at/hER09> (KeepCoding).
- <https://shorturl.at/lnSZ0> (Podcast Spotify - Fernando Herrera).