SINTAXIS DEL LENGUAJE

Añadir Scripts al HTML

Hay dos formas de hacer referencia a código JavaScript:

- Código en el propio archivo HTML:

```
<script>

alert("Hola Mundo");

</script>
```

- Código en un fichero externo:

```
<script src="./js/index.js"></script>
<!-- La etiqueta type="text/javascript" no es necesaria -->
```

> **Note**
>
> La etiqueta `script` debe incluirse antes del final del cierre de la etiqueta `body` del HTML.

Módulos

Con los módulos los ficheros de JS podrán exportar e importar código y/o datos de otros archivos de JS. Un módulo puede exportar y a la vez importar. Esto implica numerosos beneficios:

- **Modularidad**: Divide el código en partes reutilizables, mejorando la organización y el mantenimiento.
- **Reutilización**: Permite importar y usar código de otros módulos, evitando la duplicación de código.
- **Encapsulación**: Oculta la implementación interna de un módulo, exponiendo solo la interfaz pública.
- **Carga asíncrona**: Carga los módulos solo cuando se necesitan, mejorando el rendimiento.

¿Cómo usar?

1- Exportar un elemento:

```
export const articulos = [
        {id:1, nombre:"articulo1"},
        {id:2, nombre:"articulo2"},
        {id:3, nombre:"articulo3"}
];
```

2- Importar un elemento:

```
import {articulos} from "./datos.js"
```

3- Uso en el HTML:

```
<script src="./js/main.js" type="module"></script>
<!-- Hay que indicar que es de tipo módulo -->
```

Ámbito de los módulos

A veces al importar un módulo se crea un ámbito que el HTML no puede alcanzar.

La solución más efectiva es a través del ***DOM*** *(Document Object Element)*

Ejemplo:

- HTML:

```
<button id="btn">Bienvenida DOM</button>
```

- JS:

```
document.getElementById("btn").addEventListener("click", verMensaje)
```

Estructura de un proyecto

./CarpetaProyecto
/assets -> contiene las imágenes, vídeos y otros ficheros.
/css -> contiene los estilos CSS.
/js -> contiene los ficheros JS.

/node_modules -> contiene los módulos node (npm)
index.html -> página inicial de HTML.

Variables

```
var persona = "javi"; // variable de alcance global
let persona = "javi"; // variable de alcance en bloque
const persona = "javi"; // variable definida como constante, tras sun
inicialización no se puede modificar
```

Tipos de datos

JavaScript es un lenguaje de tipado dinámico, lo que significa que no es necesario especificar el tipo de dato de una variable al declararla. El tipo de dato se asigna automáticamente en función del valor inicial.

- **Tipos de datos primitivos**: son los más básicos y representan tipos de datos simples.
  - **Números**: valores numéricos o decimales.
  - **Cadenas de texto**: secuencia de carácteres.
  - **Booleanos**: true o false.
  - **Null**: ausencia de valor.
  - **Undefined**: valor no definido.
  - **Symbol**: valores únicos e inmutables.
- **Tipos de datos compuestos**: estructuras que almacenan colecciones de datos:
  - **Objetos**: estructuras de datos que contienen propiedades y métodos.
  - **Arrays**: representan listas de valores de cualquier tipo.
- **El operador TypeOf**: determina el tipo de dato de una variable. Devuelve una cadena que indica el tipo de dato.
  - "**string**": Para cadenas de texto.
  - "**number**": Para números.
  - "**boolean**": Para valores booleanos (true o false).
  - "**object**": Para objetos, arrays y funciones.
  - "**undefined**": Para variables no declaradas o valores no inicializados.
  - "**symbol**": Para valores de tipo Symbol.
  - "**function**": Para funciones.

> **Tip**
>
> **typeof** no distingue entre diferentes tipos de objetos, como arrays o funciones. Ambos se consideran como tipo "object". Si necesitaramos distinguir entre tipos de objetos

específicos, podemos utilizar **instanceof**.

Conversiones de tipo

- **Conversiones implícitas (corrección de tipos):** La conversión implícita ocurre automáticamente cuando JavaScript intenta realizar una operación con valores de diferentes tipos de datos. En estos casos, JavaScript convierte los valores al tipo de dato que sea necesario para la operación.

```
let numero = 10;
let texto = "5";

let suma = numero + texto; // suma se convierte en "105" (cadena)
console.log(suma); // Imprime: 105
```

- **Conversiones explícitas (casting)**: La conversión explícita se realiza utilizando funciones específicas para convertir un valor de un tipo de dato a otro. Se utiliza cuando se desea un control preciso sobre el tipo de dato resultante.

```
public class stringToNumber {
        let texto = "10";
        let numero = Number(texto); // numero se convierte en 10 (número)
        console.log(numero); // Imprime: 10
}

public class numberToString {
        let numero = 20;
        let texto = String(numero); // texto se convierte en "20" (cadena)
        console.log(texto); // Imprime: 20
}

public class convertToBoolean { // función Boolean()
// valores truthy -> números excepto 0, cadenas de texto excepto "", tru,
objetos
        let valor1 = 10; // truthy

// valores falsy -> número 0, cadena vacía "", flase, null, undefined
        let valor2 = ""; // falsy

        let booleano1 = Boolean(valor1); // booleano1 se convierte en true
        let booleano2 = Boolean(valor2); // booleano2 se convierte en false
```

```
        console.log(booleano1, booleano2); // Imprime: true false
}
```

Operadores

1. **Operadores aritméticos:**
   - `+`: Suma
   - `-`: Resta
   - `*`: Multiplicación
   - `/`: División
   - `%`: Resto división
   - `++`: Incremento
   - `--`: Decremento
2. **Operadores relacionales:**
   - `==`: Igualdad
   - `!=`: Desigualdad
   - `<`: Menor que
   - `<=`: Menor o igual que
   - `>`: Mayor que
   - `>=`: Mayor o igual que
3. **Operadores lógicos:**
   - `&&`: Conjunción (AND)
   - `||`: Disyunción (OR)
   - `!`: Negación (NOT)
4. **Operadores de asignación:**
   - `=`: Asignación simple
   - `+=`: Asignación por suma
   - `-=`: Asignación por resta
   - `*=`: Asignación por multiplicación
   - `/=`: Asignación por división
   - `%=`: Asignación por resto
5. **Operadores de tipo:**
   - `typeof`: Devuelve el tipo de dato de un valor
   - `instanceof`: Comprueba si un objeto es una instancia de un tipo específico
6. **Operadores de coma (,):**
   - Separa expresiones en una instrucción

- Utilizado en bucles `for` como `for (let i = 0, j = 10; i < 10; i++, j--) { ... }`

7. **Operador condicional ternario (`? :`):**
   - Es una forma abreviada de una instrucción `if-else`
8. **Operadores bit a bit:**
   - `&` : AND bit a bit
   - `|` : OR bit a bit
   - `^` : XOR bit a bit
   - `<<` : Desplazamiento de bits a la izquierda
   - `>>` : Desplazamiento de bits a la derecha
   - `~` : Negación bit a bit

Condicionales

- Sentencia `if-else` :

```
if (condicion) {
 // Código a ejecutar si la condición es verdadera
} else {
 // Código a ejecutar si la condición es falsa
}
```

- Sentencia `switch` :

```
switch (expresion) {
        case valor1:
                // Código a ejecutar si la expresión es igual a valor1
                break;
         case valor2:
                // Código a ejecutar si la expresión es igual a valor2
                break;
         default:
                // Código a ejecutar si la expresión no coincide con ningún
caso
}
```

- Operador condicional `ternario (? :)` :

```
let resultado = (condicion) ? valor_verdadero : valor_falso;
// valor_verdadero = si se cumple
```

```
// valor_falso = si no se cumple
```

Bucles

- Bucle `for`: Se utiliza para ejecutar un bloque de código un número específico de veces. Se basa en tres partes: una inicialización, una condición y una actualización.

```
for (let i = 0; i < 10; i++) {
 // Código a ejecutar 10 veces
}
```

- Bucle `for-in`: Se utiliza para recorrer las propiedades de un objeto y ejecutar un bloque de código para cada propiedad.

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

for (let propiedad in persona) {
 console.log(`${propiedad}: ${persona[propiedad]}`);
}
```

- Bucle `for-of`: Se utiliza para recorrer los valores de un iterable (como un array o una cadena de texto) y ejecutar un bloque de código para cada valor

```
let numeros = [10, 20, 30];

for (let numero of numeros) {
 console.log(numero);
}
```

- Bucle `while`: Se utiliza para ejecutar un bloque de código mientras se cumpla una condición específica. Se evalúa la condición antes de cada iteración del bucle.

```
let i = 0;

while (i < 10) {
 // Código a ejecutar mientras i sea menor que 10
 i++;
}
```

- Bucle `do-while`: El bucle do-while es similar al bucle while, pero la diferencia es que el bloque de código se ejecuta al menos una vez, incluso si la condición es falsa al principio.

```
let i = 0;

do {
 // Código a ejecutar al menos una vez
 i++;
} while (i < 10);
```

Con Programación Funcional y/o Funciones de Flecha:

En programación funcional y usando funciones de flecha en JavaScript, en lugar de bucles explícitos, solemos usar métodos de arrays e iteradores, como `map`, `forEach`, `filter`, `reduce`, etc., que permiten manipular datos de una manera más declarativa. A continuación, se muestran ejemplos de cómo implementar los diferentes tipos de bucles en un estilo funcional en JavaScript:

1. Bucle `for`

En lugar de usar un bucle `for` para ejecutar un bloque de código un número específico de veces, podemos usar el método `Array.from` para crear un array de cierto tamaño y luego aplicar un método funcional como `forEach`.

```
Array.from({ length: 10 }).forEach((_, i) => {
  // Código a ejecutar 10 veces
  console.log(`Iteración ${i + 1}`);
});
```

2. Bucle `for-in`

Para iterar sobre las propiedades de un objeto, podemos usar `Object.keys`, `Object.values` o `Object.entries` con `forEach`. Esto nos permite recorrer cada propiedad o valor sin un bucle explícito `for-in`.

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

Object.entries(persona).forEach(([propiedad, valor]) => {
  console.log(`${propiedad}: ${valor}`);
});
```

3. Bucle `for-of`

Si necesitamos recorrer los valores de un iterable, podemos usar `forEach` o `map` en caso de que queramos transformar los valores del array.

```
const numeros = [10, 20, 30];

numeros.forEach(numero => console.log(numero));
```

4. Bucle `while`

Los bucles `while` no tienen una correspondencia directa en programación funcional, pero si queremos ejecutar un bloque mientras se cumpla una condición, podemos simularlo usando recursión. Sin embargo, aquí mostramos un ejemplo de cómo podría hacerse:

```
const repetirHasta = (condicion, accion, i = 0) => {
  if (!condicion(i)) return;
  accion(i);
  repetirHasta(condicion, accion, i + 1);
};

// Ejemplo de uso:
repetirHasta(i => i < 10, i => console.log(`Iteración ${i + 1}`));
```

5. Bucle `do-while`

Para el bucle `do-while`, podemos modificar la función anterior para ejecutar la acción al menos una vez antes de verificar la condición, como en el siguiente ejemplo:

```
const repetirAlMenosUnaVez = (condicion, accion, i = 0) => {
  accion(i);
  if (!condicion(i)) return;
  repetirAlMenosUnaVez(condicion, accion, i + 1);
};

// Ejemplo de uso:
repetirAlMenosUnaVez(i => i < 10, i => console.log(`Iteración ${i + 1}`));
```