DWC

(Desarrollo Web en entorno cliente)

Índice

1.- Añadiendo Scripts al HML

Los programas de JavaScript se pueden insertar en cualquier parte de un documento HTML con la ayuda de la etiqueta <script>.

Por ejemplo:

```
<!DOCTYPE HTML>
<html>
<body>
  <p>Antes del script...</p>
  <script>
    alert( 'Hello, world!' );
  </script>
  <p>...Después del script.</p>
</body>
</html>
```

La etiqueta <script> contiene código JavaScript que se ejecuta automáticamente cuando el navegador procesa la etiqueta. Tiene algunos **atributos que rara vez se usan hoy en día**, pero aún se pueden encontrar en el código antiguo:

El atributo type:<script type=…>

El antiguo estándar HTML, HTML4, requería tener un type. Por lo general era type="text/javascript". Ya no se requiere. **Ahora, se puede usar para módulos JavaScript.**

El atributo language:<script language=…>

Este atributo estaba destinado a mostrar el idioma del script. Este atributo ya no tiene sentido porque JavaScript es el idioma predeterminado. No hay necesidad de usarlo.

2.- Scripts externos

Si tenemos mucho código JavaScript, podemos ponerlo en un archivo separado. Los archivos de script se adjuntan a HTML con el atributo src que es una ruta absoluta o relativa al script en nuestro sitio web o.también podemos dar una URL externa completa.

```
<script src="./js/script1.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/ajax.js"></script>
```

A tener en cuenta:

- Como regla general, solo los scripts más simples se colocan en HTML. **Los más complejos residen en archivos separados**.
- El beneficio de un archivo separado es que el navegador lo descargará y lo almacenará en su caché. Otras páginas que hacen referencia al mismo script lo tomarán del caché en lugar de descargarlo, por lo que el archivo se descarga solo una vez. Eso reduce el tráfico y hace que las páginas sean más rápidas.

Al incluir archivos de JavaScript externos estamos añadiendo su contenido completo al HTML, es como si hubiéramos definido su código en el HTML dentro de la etiqueta script que lo importa.

2.1.- Módulos

En el **EmacScript de 2015 (ES6)** se incorporan una serie de novedades que **cambian radicalmente el funcionamiento de JavaScript**, una de ellas son la incorporación de los módulos.

En JavaScript, las importaciones de módulos son una característica clave para organizar y reutilizar código. Permiten dividir un proyecto web en archivos separados, llamados módulos, y cargar solo las partes del código que se necesitan en cada página. Esto mejora la modularidad, la legibilidad y el mantenimiento del código.

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

Beneficios de las importaciones de módulos:

- **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.

Uso de módulos:

En el uso de módulos utilizaremos imports para poder incluir elementos en nuestros ficheros de javascript que habrán sido exportados en otros ficheros.

- Para exportar datos o funciones utilizamos la palabra export y después el dato o la función que queremos exportar
- Para importar un módulo ESM, se utiliza la palabra clave import seguida de las partes que se quieren importar entre llaves {} y el nombre del módulo precedido por la palabra from.

Por ejemplo, podríamos definir un módulo para los datos de nuestra aplicación y otro modulo para utilidades que utilizaremos en nuestra aplicación.

datos.js

Definimos una lista de datos de **artículos** que exportamos para que pueda ser utilizada en otros ficheros de javascript

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

utilidades.js

Definimos un constante ver **mensaje** y una función **verMensaje** que muestra un alert con el mensaje. Sólo exportamos la función

```
const mensaje="Bienvenido a mi Web"

export function verMensaje() {
  alert(mensaje);
}
```

Lo habitual es que el html incluyamos un archivo js que sea el punto de entrada de nuestro código en el HTML, generalmente se suele llamar main.js. Este fichero será el que realizará los imports de los demás archivos y tendrá la lógica de nuestra aplicación.

main.js

Importamos la variable artículos y la función verMensaje. Definimos en método onload de la página web para que al cargarse la página se ejecute la función verMensaje y hacer un alert de la longitud de la variable datos. De esta manera podremos ver que artículos y verMensaje son accesibles desde el fichero main.js

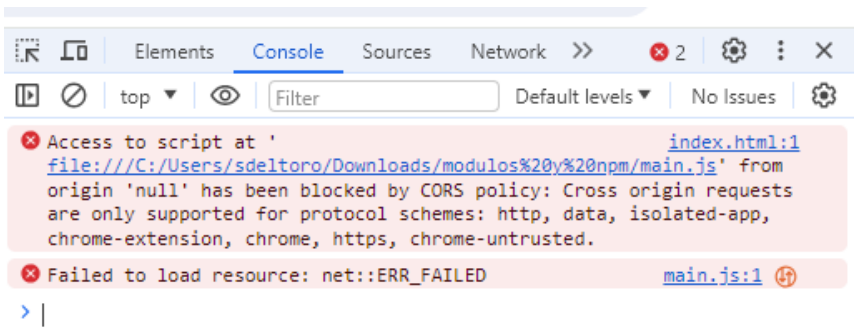
```
import {articulos}  from "./datos.js"
import {verMensaje} from "./utilidades.js"


window.onload=function(){
    verMensaje()
    alert("Total articulos: " + articulos.length)
}
```

Hasta ahora hemos creados nuestros ficheros independientes y los hemos reutilizado en el main.js, pero necesitamos incluirlos en el HTML. Para ello incluiremos el fichero main.js en el archivo index.html. Debemos indicar que este tipo script es de tipo module.

index.html

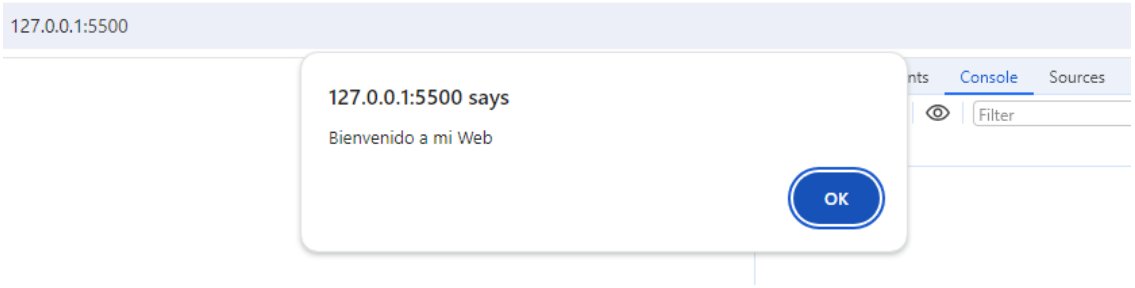
```
<!DOCTYPE html>
<html lang="es">
 <head>
  <meta charset="UTF-8">
  <script src="./js/main.js" type="module"></script>
 </head>
 <body>
  <h3>Demo de modulos</h3>
 </body>
</html>
```

Si abrimos en el navegador nuestro archivo index.html, podremos ver que el resultado no es el esperado.



Para poder trabajar con módulos deberemos de abrir nuestros archivos html en un servidor web.

Utilizaremos el live-server que lleva incorporado Visual Studio Code.



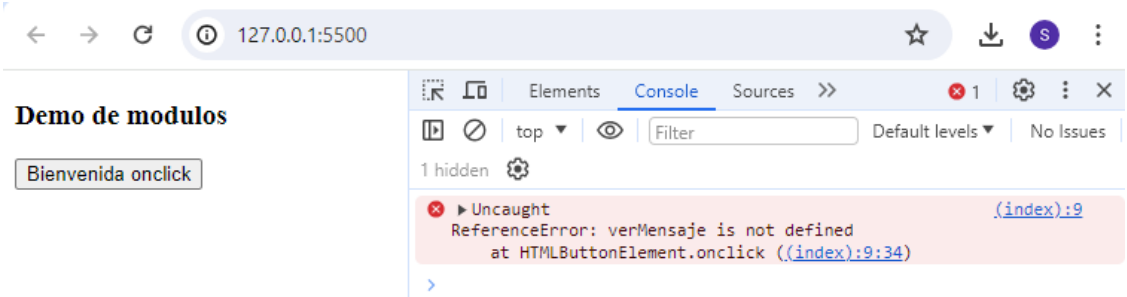
Ahora ya funciona.

Ámbito de los Módulos

Podíamos haber añadido un botón a la página web para indicarle en el evento onclick que ejecute la función verMensaje

```
<!DOCTYPE html>
<html lang="es">
 <head>
  <meta charset="UTF-8">
  <script src="./js/main.js" type="module"></script>
 </head>
 <body>
  <h3>Demo de modulos</h3>
  <button onclick="verMensaje()">Bienvenida onclick</button>
 </body>
</html>
```

Si cargamos la página index.html nos dará error al pulsar sobre el botón.



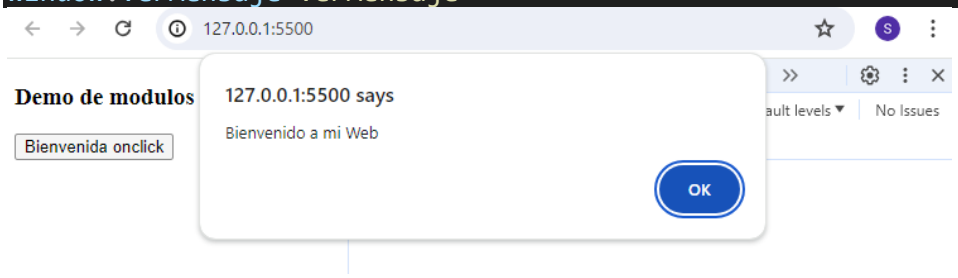
Hemos visto que no es problema de la importación porque sí que se utiliza al cargarse la página y funciona.

El problema es que al importar un módulo se ha creado un ámbito para su uso que el HTML no puede alcanzar. La asignación del evento onclick en la etiqueta del botón no encuentra la función verMensaje.

Esto quiere decir que la función está disponible, pero para poder utilizarse desde el código, hay dos soluciones a este problema:

- **La primera** es hacer visible la función verMensaje para el HTML **incorporándola al objeto Window**. Añadimos a main.js

```
window.verMensaje=verMensaje
```



Esta opción no es recomendable porque nos hace perder las ventajas del uso de módulos.

- **La segunda** es utilizar las importaciones sólo en el código y evitar referencias en las etiquetas de HTML. Para ello **utilizaremos el DOM**.

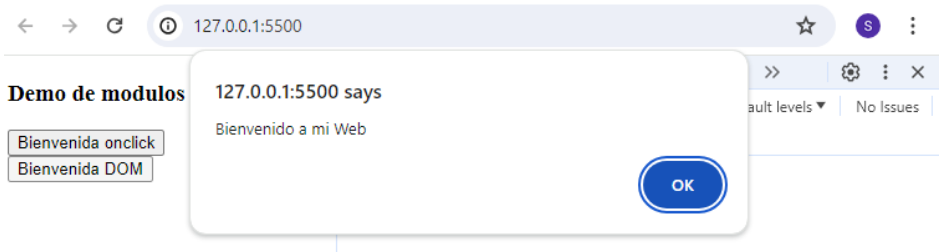
En el html

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

En main.js

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

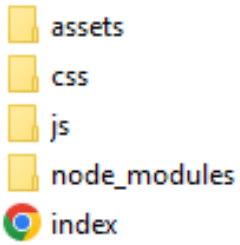
De esta manerá asignamos el evento onclick desde código atraves del DOM y los listeners de eventos



3.- Estructura de un proyecto

Al final nuestro proyecto Web va a ser un conjunto de archivos HTML, CSS, Javascript (nuestros y de terceros), imágenes y diversos recursos. Para poder organizarlo de una forma coherente deberemos utilizar algún tipo de estructura de carpetas que nos facilite la gestión.

Una sencilla sería crear una carpeta para cada tipo de archivo y dejar en la raíz del proyecto el fichero index.html



A medida que nuestro proyecto vaya creciendo necesitaremos utilizar librerías de terceros, este proceso puede ser bastante pesado de hacer manualmente, para evitar esto podemos utilizar un gestor de dependencias que nos facilite esta labor.

3.1.- npm

npm (abreviatura de **Node Package Manager**) es el administrador de paquetes por defecto para **Node.js**, un entorno de ejecución para JavaScript. Está diseñado para facilitar la instalación, actualización y gestión de dependencias de software para proyectos

Características principales de npm:

- **Gran repositorio de paquetes:** npm tiene un amplio repositorio de paquetes de código abierto, lo que permite encontrar fácilmente las herramientas y bibliotecas necesarias para nuestro proyecto.
- **Instalación sencilla:** Con solo un comando, podemos instalar paquetes desde el repositorio de npm en nuestro proyecto.
- **Actualización automática:** Podemos actualizar los paquetes instalados a la última versión con un solo comando.

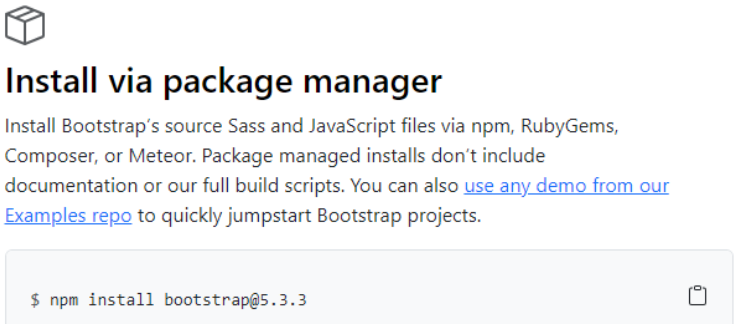
- **Gestión de dependencias:** npm te ayuda a gestionar las dependencias de tu proyecto, asegurándote de que todas las versiones de los paquetes sean compatibles entre sí.
- **Scripts personalizados:** Podemos definir scripts personalizados en tu proyecto para automatizar tareas como compilación, pruebas o despliegue.

¿Cómo funciona npm?

npm utiliza un archivo llamado **package.json** para almacenar información sobre nuestro proyecto y sus dependencias. Cuando instalamos un paquete, npm agrega una entrada a este archivo con la información del paquete y su versión. También descarga el código del paquete y lo instala en el proyecto. El código del paquete se almacena dentro de la carpeta **node_modules**

Para usar npm, necesitamos tener Node.js instalado en nuestro ordenador.

Por ejemplo, para poder utilizar bootstrap en nuestro proyecto web descargaríamos el paquete con npm

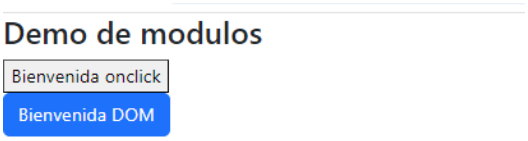


Una ejecutado el npm install aparecerá en nuestra carpeta de node_modules. Para poder utilizarlo deberemos incluir la ruta al archivo css en el archivo index.html

```
<link rel="stylesheet"
      href="./node_modules/bootstrap/dist/css/bootstrap.min.css">
```

Ahora ya podríamos utilizar los estilos de bootstrap en nuestros archivos html.

```
<button class="btn btn-primary" id="btn">Bienvenida DOM</button>
```



Uso de scripts personalizados

También podemos utilizar scripts personalizados en nuestros proyectos, por ejemplo, hemos visto que el uso de módulos conlleva tener que utilizar un servidor web. Live-Server viene como extensión de VSCode y desde ahí podemos lanzar nuestro proyecto.

Si quisiéramos poder ejecutar nuestro proyecto desde fuera del IDE deberíamos instalar el paquete live-server y posteriormente ejecutarlo.

Podríamos hacer un script llamado serve que nos permitiera hacer todo esto y después lanzarlo desde la terminal con npm

Instalacion

```
npm -g install live-server
```

Script personalizado

```
"scripts": {
    "serve": "live-server --browser=Chrome"
}
```

Ejecución desde consola

```
npm run serve
```

4.- Estructura del código

Lo primero que estudiaremos son los bloques de construcción de código. La estructura esencial del código es muy similar a la utilizada en Java o C

Declaraciones

Las declaraciones son construcciones de sintaxis y comandos que realizan acciones. Podemos tener tantas declaraciones en nuestro código como queramos. Las declaraciones se pueden separar con un punto y coma. Por lo general, las declaraciones se escriben en líneas separadas para que el código sea más legible.

Punto y coma

Se puede omitir un punto y coma en la mayoría de los casos cuando existe un salto de línea. *En la mayoría de los casos, una nueva línea implica un punto y coma. ¡Pero "en la mayoría de los casos" no significa "siempre"!*

Hay situaciones en las que JavaScript "no puede" asumir un punto y coma donde realmente se necesita. Los errores que ocurren en tales casos son bastante difíciles de encontrar y corregir.

Un ejemplo de error.

```
alert("There will be an error")
[1, 2].forEach(alert)
```

No es necesario pensar en el significado de los corchetes [] y forEach aún. Los veremos más adelante. En esencia, lo que hacemos es recorrer un array de dos elementos y mostrarlos con un alert.

El error ocurre porque JavaScript no asume un punto y coma antes de los corchetes [...]. Entonces, debido a que el punto y coma no se inserta automáticamente, el código se trata como una sola declaración. Así es como el motor lo ve:

```
alert("There will be an error")[1, 2].forEach(alert)
```

La solución en este caso sería añadir el punto y coma después del alert

```
alert("There will be an error");
[1, 2].forEach(alert)
```

Se recomienda poner punto y coma entre las declaraciones, incluso si están separadas por líneas nuevas.

Comentarios

A medida que pasa el tiempo, los programas se vuelven cada vez más complejos. Se hace necesario agregar *comentarios* que describan lo que hace el código y por qué.

Los comentarios se pueden poner en cualquier lugar de un script. No afectan su ejecución porque el motor simplemente los ignora:

- Los comentarios de una línea comienzan con dos caracteres de barra diagonal //.
- Los comentarios de varias líneas comienzan con una barra diagonal y un asterisco /* y terminan con un asterisco y una barra diagonal */.

Los comentarios aumentan la huella general del código, pero eso no es un problema en absoluto. Hay muchas herramientas que minimizan el código antes de publicar en un servidor de producción. Quitan los comentarios, por lo que no aparecen en los scripts de trabajo. Por lo tanto, los comentarios no tienen ningún efecto negativo en la producción.

Use estrict

Esta es una directiva incluida en el ES5 que obliga a tener que declarar las variables de forma explícita, de lo contrario generara un error
La directiva es una cadena: **"use strict"** o **'use strict'** que se coloca en la parte superior de un script.

Si utilizamos módulos, use strict está activado y deberemos definir todas nuestras variables

5.- Variables

En JavaScript, **la declaración de variables no es obligatoria** en el sentido estricto de la palabra. El lenguaje permite usar variables sin declararlas explícitamente. Sin embargo, esta práctica no se recomienda y puede generar errores inesperados. Además, con la directiva **use strict** pasa a ser obligatorio la declaración de variables

Las variables son elementos fundamentales en la programación que permiten almacenar datos para su posterior uso. En JavaScript, las variables se declaran utilizando palabras clave específicas y se les asigna un nombre y un valor.

Declaración de variables

Para declarar una variable, se utilizan las palabras clave var, let o const, seguidas del nombre de la variable y un operador de asignación (=) y el valor inicial.

Diferencias entre var, let y const:

- **var:** Es la palabra clave más antigua y tiene un alcance global, lo que significa que la variable puede ser accedida desde cualquier parte del código. Sin embargo, su uso puede generar problemas de alcance y es recomendable utilizar let o const en su lugar.
- **let:** Tiene un alcance de bloque, lo que significa que la variable solo es accesible dentro del bloque en el que se declara. Esto ayuda a evitar problemas de alcance y mejora la legibilidad del código.
- **const:** Se utiliza para declarar variables con valores constantes que no se pueden modificar después de su inicialización. Es útil para definir valores fijos como constantes físicas o matemáticas.

6.- 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.
A pesar de ser un lenguaje de tipado dinámico, JavaScript ofrece varios tipos de datos básicos para representar diferentes tipos de información. Estos tipos de datos son esenciales para estructurar y organizar los datos.

Tipos de datos primitivos

Los tipos de datos primitivos son los más básicos y representan valores simples, son los siguientes:

1. **Números:** Representan valores numéricos enteros o decimales
2. **Cadenas de texto:** Representan secuencias de caracteres.
3. **Booleanos:** Representan valores true o false.
4. **Null:** Representa la ausencia de cualquier valor, un valor nulo o "vacío".

5. **Undefined:** Representa un valor indefinido, generalmente cuando una variable no ha sido inicializada o no existe.
6. **Symbol:** Un tipo de dato especial introducido en ES6 para representar valores únicos e inmutables.

Tipos de datos compuestos

Los tipos de datos compuestos son estructuras que almacenan colecciones de datos o información más compleja. Los tipos de datos compuestos en JavaScript son:

1. **Objetos:** Representan estructuras de datos que contienen propiedades y métodos.
2. **Arrays:** Representan listas de valores de cualquier tipo. Se utilizan para almacenar colecciones de elementos como números, cadenas de texto, objetos, etc.

El operador typeof

El operador typeof en JavaScript se utiliza para determinar el tipo de dato de una variable. Devuelve una cadena que indica el tipo de dato.

El operador typeof puede devolver los siguientes valores:

- "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.

El operador typeof se utiliza principalmente para verificar el tipo de dato de un valor antes de realizar operaciones o tomar decisiones en el código. Por ejemplo, se puede utilizar para comprobar si una variable contiene un número antes de intentar realizar una operación matemática.

```
let numero = "10";

if (typeof numero === "number") {
  console.log("El valor es un número");
  // Se puede realizar una operación matemática con numero
} else {
  console.log("El valor no es un número");
  // Se debe convertir el valor a un número
}
```

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**.

7.- Conversiones de tipo

JavaScript nos permite realizar conversión de tipos para poder realizar operaciones con los datos correctos. Existen dos tipos de conversiones de tipos:

1. Conversión implícita (coerción 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
```

2. Conversión explícita (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.

Convertir una cadena a un número:

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

Disponemos también de las funciones parseInt(texto) y parseFloat(texto) que convierten un número leído como texto en el número correspondiente en formato entero o decimal. Si el texto no se puede convertir a número devuelven NaN

Convertir un número a una cadena:

```
let numero = 20;
let texto = String(numero); // texto se convierte en "20" (cadena)
console.log(texto); // Imprime: 20
```

Convertir a Boolean

La función Boolean() convierte cualquier valor a un valor booleano. Si el valor es "truthy" (considerado como verdadero en JavaScript), se convierte en true. Si el valor es "falsy" (considerado como falso en JavaScript), se convierte en false.

Valores "truthy":

- Cualquier número excepto 0.
- Cualquier cadena de texto excepto la cadena vacía "".
- true.
- Cualquier objeto.

Valores "falsy":

- El número 0.
- La cadena vacía "".
- false.
- null.
- undefined.

```
let valor1 = 10; // truthy
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
```

8.- Los operadores en JavaScript

Los operadores en JavaScript son símbolos o palabras clave que se utilizan para realizar operaciones con valores, variables o expresiones. Se pueden clasificar en diferentes grupos según su función:

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 **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

9.- Condicionales

Las condicionales en JavaScript son estructuras de control de flujo que permiten ejecutar diferentes secciones de código en función de un valor booleano (true o false). Son esenciales para tomar decisiones y controlar el comportamiento del programa en base a diferentes condiciones.

Sentencia if-else:

La sentencia if-else es la estructura condicional más básica en JavaScript. Se utiliza para evaluar una condición y ejecutar un bloque de código si la condición es verdadera (true), o un bloque de código alternativo si la condición es falsa (false).

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

Sentencia switch:

La sentencia switch se utiliza para evaluar una expresión y ejecutar un bloque de código específico en función del valor de la expresión. Es útil cuando se tienen múltiples opciones o casos a considerar.

```
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 (? :):

El operador condicional ternario es una forma abreviada de escribir una sentencia if-else en una sola línea. Se utiliza para evaluar una condición y asignar un valor en función del resultado de la condición.

```
let resultado = (condicion) ? valor_verdadero : valor_falso;
```

10.- Bucles

Los bucles en JavaScript son estructuras de control de flujo que permiten ejecutar repetidamente un bloque de código un número determinado de veces o hasta que se cumpla una condición específica. Son esenciales para automatizar tareas repetitivas y procesar conjuntos de datos de manera eficiente.

Bucle for:

El 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 while:

El 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);
```

Bucle for...in:

El 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:

El 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);
}
```