Repaso JS

¿Qué es JavaScript?

JavaScript (JS) es un lenguaje de programación ligero y flexible que se utiliza principalmente para añadir interactividad a las páginas web, permitiendo controlar el comportamiento del navegador, actualizar el contenido de la página sin recargar, manipular HTML y CSS, y crear experiencias de usuario dinámicas.

Cómo añadir JavaScript en HTML

Existen varias maneras de incluir JavaScript en un documento HTML:

1. **Script en línea (inline)**: Escribir el código JavaScript directamente dentro de una etiqueta `<script>` en el archivo HTML.

```
<html>
<body>
    <h1>Hola Mundo</h1>
    <script>
        alert("Hola desde JavaScript en línea!");
    </script>
</body>
</html>
```

2. **Archivo externo**: Crear un archivo `.js` separado y enlazarlo en el HTML usando la etiqueta `<script src="ruta_del_archivo.js"></script>`. Esta es la forma recomendada para mantener el código organizado.

```
<html>
<body>
    <h1>Hola Mundo</h1>
    <script src="app.js"></script>
</body>
</html>
```

3. **Script en el `<head>` o `<body>`**: La etiqueta `<script>` puede ir en el `<head>` o al final del `<body>`. Colocarlo al final del `<body>` ayuda a que el HTML se cargue primero, mejorando el rendimiento.

```
<html>
<head>
    <script src="app.js"></script>
</head>
<body>
    <h1>Hola Mundo</h1>
```

```
</body>
</html>
```

Conceptos Básicos de JavaScript

1. Variables

Las variables almacenan datos y se definen usando `let`, `const` o `var`:

- `let`: para variables que pueden cambiar de valor.
- `const`: para variables constantes, que no cambian.
- `var`: una forma antigua de declarar variables (menos recomendable).

Ejemplos:

```
let nombre = "Ana";
const PI = 3.1416;
var edad = 25;
```

Nota: Existe una gran diferencia entre las variables `let` y `var` como ves en el siguiente bloque de código donde en función del **scope o alcance** de la variable podemos acceder o modificar su valor en según que casos:

```
function ejemploVarLet() {
    var mensaje = "Hola desde var";
    let saludo = "Hola desde let";

    if (true) {
        var mensaje = "Var dentro del bloque"; // Esta reasignación afecta
el `mensaje` fuera del bloque.
        let saludo = "Let dentro del bloque";  // Esta `let` es
independiente y solo existe en este bloque.

        console.log(mensaje); // "Var dentro del bloque".
        console.log(saludo);  // "Let dentro del bloque".
    }

    console.log(mensaje); // "Var dentro del bloque" (el valor cambió
globalmente dentro de la función).
    console.log(saludo);  // "Hola desde let" (no fue afectado por el
bloque `if`).
}

ejemploVarLet();
```

```
function ejemploVarLet() {

    if (true) {
        var var_creada_if = "var creada en el if";
        let let_creada_if = "let creada en el if";

        console.log(var_creada_if); // "var creada en el if".
        console.log(let_creada_if);  // "let creada en el if".
    }

    console.log(var_creada_if); //Muestra el mensaje del var creada en el if que existe fuera del bloque. Variable accesible en el total del código, tiene alcance global.
    console.log(let_creada_if); //Da error ya que no existe la variable fuera del bloque. Variable solo accesible en el bloque, tiene alcance de bloque.
}

ejemploVarLet();
```

Otro aspecto a tratar cuando hablamos de `var` y `let` es el **hoisting o elevación**. El hoisting es un comportamiento en JavaScript donde las declaraciones de variables (y funciones) se mueven ("elevan") a la parte superior de su contexto de ejecución antes de que se ejecute el código. Esto significa que puedes usar una variable antes de declararla, aunque su valor será `undefined` si se usa antes de la asignación.

- **`var`**: Las variables declaradas con `var` son elevadas a la parte superior de su contexto de ejecución (función o global), y se inicializan con `undefined`. Esto permite que la variable sea accesible antes de su declaración en el código.
- **`let`**: Las variables declaradas con `let` también son elevadas, pero no pueden ser utilizadas antes de su declaración, lo que genera un **ReferenceError** si se intenta acceder a ellas antes de ser inicializadas.

```
// Ejemplo con var
console.log(variableVar); // undefined
var variableVar = "Soy una variable con var";
console.log(variableVar); // "Soy una variable con var"

// Ejemplo con let
console.log(variableLet); // ReferenceError: Cannot access 'variableLet' before initialization
let variableLet = "Soy una variable con let";
console.log(variableLet); // Este código no se ejecuta debido al error anterior
```

En este caso:

1. **`var`**:

- En el primer `console.log(variableVar);`, el resultado es `undefined`, ya que `variableVar` ha sido elevada, pero aún no ha sido inicializada con un valor.
- Posteriormente, se asigna el valor "Soy una variable con var", que es lo que se muestra en el segundo `console.log`.

2. `let`:

   - Al intentar acceder a `variableLet` antes de su declaración, se produce un **ReferenceError**, ya que `let` no permite el acceso a la variable antes de su inicialización. Esto genera un error y el código posterior no se ejecuta.

Este comportamiento de hoisting es importante a considerar al escribir código en JavaScript, ya que puede llevar a confusiones y errores si no se comprende bien cómo funcionan `var` y `let`.

2. Tipos de Datos

Principales tipos de datos en JavaScript:

- **Números**: `let edad = 30;`
- **Cadenas de texto (Strings)**: `let saludo = "Hola";`
- **Booleanos**: `let esVerdadero = true;`
- **Arreglos (Arrays)**: `let colores = ["rojo", "azul", "verde"];`
- **Objetos**: `let persona = { nombre: "Ana", edad: 25 };`
- **Undefined**: una variable declarada sin valor.
- **Null**: una variable que intencionalmente no tiene valor.
- **NaN (Not a Number)**: Representa un valor que no es un número, aunque esté en una operación aritmética. `NaN` aparece cuando una operación matemática no tiene un resultado válido, por ejemplo, al intentar convertir una cadena de texto que no contiene números a un valor numérico.

  ```
  let resultado = "hola" * 3; // resultado será NaN
  ```

  Para verificar si un valor es `NaN`, se puede usar la función `isNaN(valor)`:

  ```
  console.log(isNaN("hola" * 3)); // true
  ```

  Para ver el tipo de dato de una variable podemos emplear el operador `typeof`.

  ```
  console.log(typeof edad); //Es de tipo number
  ```

3. Métodos de Interés para Números y Cadenas

3.1 Métodos para Números

- `Math.round()`: Redondea un número al entero más cercano. Si el decimal es 0.5 o mayor, se redondea hacia arriba.

```
let numero = 4.5;
let redondeado = Math.round(numero); // 5
```

- `Math.ceil()`: Redondea un número hacia arriba, al siguiente entero más cercano, sin importar el valor decimal.

```
let numero = 4.1;
let redondeadoArriba = Math.ceil(numero); // 5
```

- `Math.floor()`: Redondea un número hacia abajo, al entero más cercano.

```
let numero = 4.9;
let redondeadoAbajo = Math.floor(numero); // 4
```

- `Math.trunc()`: Elimina la parte decimal de un número y devuelve solo la parte entera sin redondear.

```
console.log(Math.trunc(4.9));  // 4
console.log(Math.trunc(4.1));  // 4
console.log(Math.trunc(-4.9)); // -4
console.log(Math.trunc(-4.1)); // -4
console.log(Math.trunc(4));    // 4
```

- `Math.abs()`: Devuelve el valor absoluto de un número, eliminando el signo negativo.

```
let numero = -5;
let absoluto = Math.abs(numero); // 5
```

- `Math.random()`: Genera un número pseudoaleatorio entre 0 (inclusive) y 1 (exclusive). Para un rango específico, combina con `Math.floor()`.

```
let aleatorio = Math.random(); // Ejemplo: 0.123456
let entre1y10 = Math.floor(Math.random() * 10) + 1; // Entre 1 y 10
```

- `Math.pow(base, exponente)`: Eleva un número a una potencia.

```
let base = 2;
let exponente = 3;
let resultado = Math.pow(base, exponente); // 8
```

- `Math.sqrt()`: Calcula la raíz cuadrada de un número.

```
let numero = 16;
let raizCuadrada = Math.sqrt(numero); // 4
```

3.2 Métodos para Cadenas

- **Declaración de una variable string**: Puedes usar comillas simples, dobles o comillas invertidas (template literals).

```
let mensaje = "Hola, mundo!";
```

- **Conversión a mayúsculas y minúsculas**: Utiliza `.toUpperCase()` y `.toLowerCase()`.

```
let texto = "Hola Mundo";
let mayusculas = texto.toUpperCase(); // "HOLA MUNDO"
let minusculas = texto.toLowerCase(); // "hola mundo"
```

- **Declaración de un número y conversión a string**: Asigna un valor numérico y convierte a cadena usando `String()` o `toString()`.

```
let numero = 123;
let cadenaDesdeString = String(numero); // "123"
```

- **Extraer la longitud de una cadena**: Utiliza la propiedad `.length`.

```
let mensaje = "Hola, mundo!";
let longitud = mensaje.length; // 13
```

- **Obtener el tercer elemento de una cadena**: Utiliza la notación de corchetes o el método `.charAt()`.

```
let texto = "JavaScript";
let tercerCaracter = texto[2]; // "v"
let tercerCaracterCharAt = texto.charAt(2); // "v"
```

- **Verificar si existe una palabra en la cadena**: Usa el método `.includes()`.

```
let texto = "Hola, mundo!";
let contieneHola = texto.includes("Hola"); // true
```

- **Uso de `indexOf()`**: Devuelve la posición de la primera aparición de una subcadena.

```
let texto = "Hola, mundo!";
let posicion = texto.indexOf("mundo"); // 6
```

- **Uso de `slice()`**: Extrae una parte de una cadena.

```
let texto = "JavaScript es genial";
let subcadena = texto.slice(0, 10); // "JavaScript"
```

- **Uso de `replace()` y `replaceAll()`**: Reemplaza subcadenas en una cadena.

```
let texto = "Hola, mundo!";
let nuevoTexto = texto.replace("mundo", "JavaScript"); // "Hola,
JavaScript!"
```

```
let texto = "Hola, mundo! Mundo es genial.";
let nuevoTexto = texto.replaceAll("Mundo", "JavaScript"); // "Hola,
mundo! JavaScript es genial."
```

- **Uso de `trim()`**: Elimina espacios en blanco al principio y al final de una cadena.

```
let texto = "   Hola, mundo!   ";
let textoSinEspacios = texto.trim(); // "Hola, mundo!"
```

- **Uso de `split()`**: Divide una cadena en un array de subcadenas.

```
let texto = "uno, dos, tres";
let array = texto.split(", "); // ["uno", "dos", "tres"]
```

- **Uso de `substring()`**: Devuelve una parte de la cadena entre dos índices.

```
let texto = "JavaScript es genial";
let subcadena = texto.substring(0, 10); // "JavaScript"
```

- **Uso de `.concat()`**: Une dos o más cadenas.

```
let saludo = "Hola, ";
let nombre = "Martín";
let mensaje = saludo.concat(nombre); // "Hola, Martín"
```

- **Uso de `search()`**: Busca una expresión regular en una cadena y devuelve la posición de la primera coincidencia. Si no se encuentra, devuelve `-1`.

```
let texto = "JavaScript es genial";
let posicion = texto.search("genial"); // 12
```

- **Uso de `match()`**: Busca una coincidencia en una cadena con una expresión regular y devuelve un array con las coincidencias encontradas. En caso contrario devuelve null.

```
 let correo = "mgilblanco@edu.xunta.gal";
 let expresionRegular = /^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]
{2,}$/;

 let coincidencias = correo.match(expresionRegular);

 if (coincidencias) {
    console.log("El correo es válido.");
 } else {
    console.log("El correo no es válido.");
 }
```

4. Arrays y Recorrido de Estos

Un **array** es una estructura de datos que permite almacenar múltiples elementos en una sola variable. Los elementos se almacenan en una secuencia, y se pueden acceder por su índice.

Declaración de Arrays

- **Array vacío**:

```
let arrayVacio = [];
```

- **Array con elementos**:

```
let frutas = ["manzana", "banana", "cereza"];
```

- **Usando `new Array()`**:

```
let numeros = new Array(1, 2, 3, 4);
```

Métodos de Interés para Arrays

- `push()`: Añade un elemento al final del array.

```
frutas.push("naranja"); // ["manzana", "banana", "cereza", "naranja"]
```

- `pop()`: Elimina y devuelve el último elemento del array.

```
let ultimaFruta = frutas.pop(); // "naranja"
```

- `shift()`: Elimina y devuelve el primer elemento del array.

```
let primeraFruta = frutas.shift(); // "manzana"
```

- `unshift()`: Añade un elemento al inicio del array.

```
frutas.unshift("fresa"); // ["fresa", "banana", "cereza"]
```

- `indexOf()`: Devuelve el índice del primer elemento que coincida con el valor especificado, o `-1` si no existe.

```
let indiceCereza = frutas.indexOf("cereza"); // 2
```

- `includes()`: Devuelve `true` si el array contiene el valor especificado, `false` en caso contrario.

```
let existeBanana = frutas.includes("banana"); // true
```

- `slice()`: Extrae una porción del array y devuelve un nuevo array con los elementos seleccionados.

```
let subArray = frutas.slice(1, 3); // ["banana", "cereza"]
```

- `splice()`: Añade, elimina o reemplaza elementos en el array en una posición específica.

```
frutas.splice(1, 1, "mango"); // ["fresa", "mango", "cereza"]
```

- `concat()`: Combina dos o más arrays y devuelve un nuevo array.

```
let otrasFrutas = ["pera", "piña"];
let todasFrutas = frutas.concat(otrasFrutas); // ["fresa", "mango",
"cereza", "pera", "piña"]
```

- `join()`: Une todos los elementos de un array en una cadena, separados por un separador especificado.

```
let cadenaFrutas = frutas.join(", "); // "fresa, mango, cereza"
```

- `sort()`: Ordena los elementos de un array en orden ascendente. Por defecto, ordena alfabéticamente. Para ordenar numéricamente, se debe pasar una función de comparación.

```
let frutas = ["manzana", "banana", "cereza"];
frutas.sort(); // ["banana", "cereza", "manzana"]

let numeros = [3, 1, 4, 1, 5];
numeros.sort((a, b) => a - b); // [1, 1, 3, 4, 5]
```

- `reverse()`: Invierte el orden de los elementos en el array.

```
frutas.reverse(); // ["manzana", "cereza", "banana"]
```

- `reduce()`: Aplica una función a un acumulador y a cada elemento del array (de izquierda a derecha), reduciéndolo a un único valor. Es útil para sumar, multiplicar o combinar valores.

```
let numeros = [1, 2, 3, 4];
let sumaTotal = numeros.reduce((acumulador, numero) => acumulador +
numero, 0);
console.log(sumaTotal); // 10
```

- `some()`: Devuelve `true` si al menos un elemento en el array cumple con la condición dada.

```
let numeros = [1, 2, 3, 4];
let hayImpares = numeros.some(numero => numero % 2 !== 0); // true
```

- `every()`: Devuelve `true` si todos los elementos en el array cumplen con la condición dada.

```
let numeros = [2, 4, 6];
let todosPares = numeros.every(numero => numero % 2 === 0); // true
```

- `flat()`: Aplana un array de arrays en un único array. Se le puede pasar un nivel de profundidad (por defecto, `1`).

```
let arrayAnidado = [1, [2, 3], [4, [5, 6]]];
let arrayPlano = arrayAnidado.flat(2); // [1, 2, 3, 4, 5, 6]
```

Recorrido de Arrays

- **Bucle `for`**: Recorre el array usando el índice de cada elemento.

```
for (let i = 0; i < frutas.length; i++) {
    console.log(frutas[i]);
}
```

- **Bucle `for...of`**: Recorre el array directamente por cada elemento.

```
for (const fruta of frutas) {
    console.log(fruta);
}
```

- **Bucle `forEach()`**: Ejecuta una función para cada elemento del array.

```
 frutas = ["pera", "manzana", "naranja"]

 frutas.forEach(fruta => console.log(fruta));
 frutas.forEach((fruta, indice) => console.log(fruta, indice));
```

- `map()`: Aplica una función a cada elemento del array y devuelve un nuevo array con los resultados.

```
 let frutasMayusculas = frutas.map(fruta => {return
fruta.toUpperCase()});
 console.log(frutasMayusculas); // [ "PERA", "MANZANA", "NARANJA" ]
```

Nota: La diferencia principal entre `forEach` y `map` en JavaScript radica en su propósito y en el valor de retorno:

- **`forEach()`**: Ejecuta una función para cada elemento del array, pero **no devuelve un nuevo array**. Simplemente itera sobre los elementos y es útil cuando quieres realizar acciones (como imprimir o modificar elementos en su lugar) sin crear una nueva colección.
- **`map()`**: Aplica una función a cada elemento del array y **devuelve un nuevo array** con los resultados. Es útil cuando quieres transformar los elementos y obtener una nueva versión del array original.

```
let numeros = [1, 2, 3, 4];

// Ejemplo con `forEach`
numeros.forEach((numero, index) => {
   numeros[index] = numero * 2;  // Modifica los valores en el array
original
});
console.log(numeros); // [2, 4, 6, 8]

// Ejemplo con `map`
let numerosDoblados = numeros.map(numero => numero * 2);
console.log(numerosDoblados); // [4, 8, 12, 16] - Nuevo array con los
valores transformados
console.log(numeros); // [2, 4, 6, 8] - Array original sin cambios
adicionales
```

5. Operadores

Operadores básicos en JavaScript:

- **Aritméticos**: `+`, `-`, `*`, `/`, `%`, `**`.
- **Asignación**: `=`, `+=`, `-=`, etc.
- **Comparación**: `==`, `===`, `!=`, `!==`, `>`, `<`, `>=`, `<=`
- **Lógicos**: `&&` (AND), `||` (OR), `!` (NOT)

6. Estructuras de Control

- **Condicionales**: `if`, `else`, `else if`, `switch`

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

```
    console.log("Es menor de edad");
}
```

También puedes usar `else if` y `switch` para manejar múltiples condiciones:

```
let dia = "lunes";
switch (dia) {
    case "lunes":
        console.log("Inicio de semana");
        break;
    case "viernes":
        console.log("Fin de semana próximo");
        break;
    default:
        console.log("Día intermedio");
}
```

Nota: Mención especial tiene el operador ternario que al igual que en otros lenguajes, tambien existe.

```
let edad = 17;

let resultado = (edad >= 18) ? "Es mayor de edad" : "Es menor de
edad";
```

- **Bucles**: `for`, `while`, `do...while`
  - **Bucle `for`**:

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

  - **Bucle `while`**:

    ```
    let k = 0;
    while (k < 5) {
        console.log("Iteración número: " + k);
        k++;
    }
    ```

  - **Bucle `do...while`**:

    ```

    ```

```
let j = 0;
do {
    console.log("Iteración número: " + j);
    j++;
} while (j < 5);
```

- **Bucle `for...of`** (para iterar sobre arrays o strings):

```
const array = ["manzana", "banana", "cereza"];
for (const fruta of array) {
    console.log(fruta);
}

for (const letra of array[0]) {
    console.log(letra);
}
```

- **Bucle `for...in`** (para iterar sobre propiedades de un objeto):

```
const persona = {
    nombre: "Juan",
    edad: 30,
    ciudad: "Madrid"
};

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

- **Ventanas y alertas**: `alert`, `confirm`, `prompt`
  - **Ventana `alert`**:

```
alert("¡Bienvenido a la página!");
```

La función `alert` muestra un mensaje en una ventana emergente con un solo botón ("Aceptar"). Es útil para informar al usuario sobre algún evento o advertencia. En el ejemplo anterior, el usuario verá una ventana emergente con el mensaje “¡Bienvenido a la página!” y un botón "Aceptar".

  - **Ventana `confirm`**:

```
let respuesta = confirm("¿Estás seguro de que quieres borrar este
archivo?");
```

```
if (respuesta) {
    console.log("Archivo borrado");
} else {
    console.log("Acción cancelada");
}
```

La función `confirm` muestra una ventana emergente con un mensaje y dos opciones: "Aceptar" y "Cancelar". Retorna `true` si el usuario hace clic en "Aceptar" y `false` si hace clic en "Cancelar". En este ejemplo, si el usuario selecciona "Aceptar", `respuesta` será `true` y el mensaje “Archivo borrado” se mostrará en la consola; si selecciona "Cancelar", `respuesta` será `false` y se mostrará “Acción cancelada”.

- **Ventana `prompt`**:

```
let nombre = prompt("¿Cuál es tu nombre?", "Anónimo");

if (nombre) {
    console.log(`Hola, ${nombre}!`);
} else {
    console.log("No ingresaste un nombre.");
}
```

La función `prompt` muestra una ventana emergente con un campo de texto para que el usuario ingrese un valor. Tiene dos botones, "Aceptar" y "Cancelar". Si el usuario hace clic en "Aceptar", `prompt` devuelve el valor ingresado como una cadena de texto; si hace clic en "Cancelar", devuelve `null`. En el ejemplo anterior, el mensaje “¿Cuál es tu nombre?” aparece con "Anónimo" como valor predeterminado. Si el usuario ingresa un nombre y selecciona "Aceptar", el nombre se almacenará en la variable `nombre` y se mostrará en la consola. Si el usuario hace clic en "Cancelar" o deja el campo vacío, la consola mostrará “No ingresaste un nombre”.

7. Funciones

Las funciones encapsulan bloques de código que pueden reutilizarse. Se pueden declarar de varias maneras:

- **Declaración de función**:

```
function saludar(nombre) {
   return `Hola, ${nombre}!`;
}
console.log(saludar("Carlos"));
```

Tambien puedo darle un valor por defecto a los parámetros.

- **Expresión de función**:

```
function saludar(nombre = "Anónimo") {
   return `Hola, ${nombre}!`;
}
console.log(saludar());
```

- **Funciones de flecha (arrow functions)**:

```
const multiplicar = (a, b) => a * b;
console.log(multiplicar(3, 4));

const cuadrado = a => a * a;
console.log(cuadrado(2));
```

- **Uso de `rest` para múltiples parámetros**:

```
let resultado = 0;
const sumaTodos = (...numeros) => {
   numeros.forEach(numero => {
      resultado += numero;
   });
   console.log(resultado);
};

sumaTodos(1, 2, 3, 4); //10
sumaTodos(1, 2, 3, 4, 5); // 15
```

- **Uso de funciones anónimas**:
  - En este ejemplo, se asigna una **función anónima** a la constante `saludo`. Al llamar a `saludo()`, la función ejecuta el `console.log`, mostrando el mensaje en la consola.

```
const saludo = function(){
   console.log("Función anónima, la puedo llamar con saludo()");
}

saludo();
```

  - Este ejemplo usa una **función anónima** dentro de `setInterval`. La función se ejecuta cada segundo y muestra en consola el valor de `veces`, que incrementa en cada ejecución.

```
let veces = 0;

setInterval(function(){
   console.log("Me ejecuto cada segundo", veces);
```

```
    veces++;
}, 1000);
```

- Aquí se usa una **función flecha anónima** dentro de `setInterval` para hacer lo mismo que el ejemplo anterior. Las funciones flecha son una forma más concisa de escribir funciones anónimas. En `setInterval` tendremos dos parámetros que son, la función a ejecutar y el tiempo periódico de cuando se ejecuta la función, esta función que estamos definiendo como parámetro es lo que en javascript se conoce como callback.

```
let veces = 0;

setInterval(() => {
    console.log("Me ejecuto cada segundo", veces);
    veces++;
}, 1000);
```

- Del mismo modo que en el caso anterior podríamos hacer lo siguiente:

```
let veces = 0;

const funcionAnonima = function() {
    console.log("Me ejecuto cada segundo", veces);
    veces++;
};

setInterval(funcionAnonima, 1000);
```

```
let veces = 0;

const funcionAnonima = () => {
    console.log("Me ejecuto cada segundo", veces);
    veces++;
};

setInterval(funcionAnonima, 1000);
```

Nota: Si queremos utilizar unda función como callback que tenga un parámetro no podemos hacer lo siguiente:

```
const saludo = (nombre = "Anónimo") => {
    console.log(`Como estás ${nombre}`);
};

setInterval(saludo("Martín"), 1000);
```

Tenemos obligatoriamente que definir una función anónima y dentro de ella llamar a esa función de callback con parámetro.

```
const saludo = (nombre = "Anónimo") => {
   console.log(`Como estás ${nombre}`);
};

setInterval(() => {saludo("Martín")}, 1000);
```

8. Manipulación del DOM (Document Object Model)

El DOM permite interactuar con elementos HTML desde JavaScript.

- **Seleccionar elementos**:

```
let titulo = document.getElementById("titulo");
let parrafos = document.querySelectorAll("p");
```

- **Modificar contenido**:

```
titulo.textContent = "Nuevo título";
```

- **Cambiar estilos**:

```
titulo.style.color = "blue";
```

- **Eventos**: Escuchar y responder a eventos como clics o movimientos del ratón.

```
titulo.addEventListener("click", () => {
    alert("¡Título clickeado!");
});
```