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.");
}
```

3.3 Objeto Date

3.2 Objeto Date

El **objeto `Date`** en JavaScript se utiliza para trabajar con fechas y horas. Es muy útil para obtener la fecha actual, manipular fechas y realizar cálculos relacionados con el tiempo.

3.3.1 Creación de un objeto `Date`

Puedes crear una instancia del objeto `Date` de diferentes formas:

```
// Fecha actual
let fechaActual = new Date();
console.log(fechaActual); // Muestra la fecha y hora actual

// Fecha específica (año, mes, día, horas, minutos, segundos, milisegundos)
let fechaEspecifica = new Date(2023, 10, 4, 12, 30, 0, 0); // Meses en JavaScript
van de 0 (enero) a 11 (diciembre)
console.log(fechaEspecifica); // Muestra: "Sat Nov 04 2023 12:30:00 GMT..."
```

```
let fechaCadena = new Date("2024-11-04T12:30:00Z"); // Usando formato de cadena
console.log(fechaCadena); // Muestra: "Mon Nov 04 2024 12:30:00 GMT+0000..."
```

3.3.2 Métodos para obtener información de un `Date`

El objeto `Date` proporciona métodos para extraer partes específicas de la fecha y hora:

```
let fecha = new Date();

console.log(fecha.getFullYear()); // Obtiene el año, por ejemplo: 2024
console.log(fecha.getMonth()); // Obtiene el mes (0 a 11). Por ejemplo, 10
representa noviembre
console.log(fecha.getDate()); // Obtiene el día del mes (1 a 31)
console.log(fecha.getDay()); // Obtiene el día de la semana (0 = domingo, 1 =
lunes, ...)
console.log(fecha.getHours()); // Obtiene la hora actual (0 a 23)
console.log(fecha.getMinutes()); // Obtiene los minutos (0 a 59)
console.log(fecha.getSeconds()); // Obtiene los segundos (0 a 59)
console.log(fecha.getMilliseconds()); // Obtiene los milisegundos (0 a 999)
console.log(fecha.getTime()); // Obtiene el tiempo en milisegundos desde el 1 de
enero de 1970 (timestamp)
```

3.3.3 Métodos para establecer partes de un `Date`

Puedes modificar un objeto `Date` con métodos para establecer sus propiedades:

```
let fechaModificable = new Date();
fechaModificable.setFullYear(2025); // Establece el año a 2025
fechaModificable.setMonth(5); // Establece el mes a junio (5 = junio)
fechaModificable.setDate(15); // Establece el día del mes a 15
fechaModificable.setHours(14); // Establece las horas a 14 (2:00 PM)
fechaModificable.setMinutes(45); // Establece los minutos a 45
fechaModificable.setSeconds(30); // Establece los segundos a 30
console.log(fechaModificable); // Muestra la fecha modificada
```

3.3.4 Métodos útiles para trabajar con fechas

`Date.now()`: Devuelve el número de milisegundos desde el 1 de enero de 1970.

```
console.log(Date.now()); // Muestra un número grande, como 1700389080000
```

`toDateString()` **y** `toTimeString()`: Convierte la fecha o la hora a una cadena más legible.

```
console.log(fecha.toDateString()); // Muestra la fecha en formato "Mon Nov 04
2024"
console.log(fecha.toTimeString()); // Muestra la hora en formato "12:30:00
GMT+0000 (Coordinated Universal Time)"
```

`toISOString()`: Devuelve la fecha en formato ISO (ideal para almacenamiento y transferencia de datos).

```
console.log(fecha.toISOString()); // Muestra la fecha en formato "2024-11-
04T12:30:00.000Z"
```

3.3.5 Comparación de fechas

Se puede comparar dos objetos `Date` con operadores de comparación (>, <, >=, <=) o usando `getTime()` para obtener su valor en milisegundos.

```
let fecha1 = new Date("2024-11-04");
let fecha2 = new Date("2023-11-04");

if (fecha1 > fecha2) {
  console.log("fecha1 es más reciente que fecha2");
} else {
  console.log("fecha2 es más reciente o igual a fecha1");
}
```

Nota: Cuando compares fechas, es mejor usar `getTime()` para mayor precisión.

```
console.log(fecha1.getTime() > fecha2.getTime()); // Devuelve `true` o `false`
```

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 (Modelo de Objeto de Documento) es una representación estructurada de un documento HTML o XML en forma de árbol de nodos. Cada nodo representa una parte del documento, como un elemento, un atributo o un texto. A través de JavaScript, podemos manipular el DOM para modificar dinámicamente el contenido y el estilo de una página web.

8.1 Selección de elementos en el DOM

Para manipular elementos en el DOM, primero necesitamos seleccionarlos. Existen cinco métodos principales para seleccionar elementos en JavaScript: `getElementById`, `getElementsByClassName`, `getElementsByTagName`, `querySelector`, y `querySelectorAll`.

8.2 `getElementById()`

Este método selecciona un elemento del DOM utilizando su `id`, que debe ser único dentro del documento.

```
let caja = document.getElementById("caja");
console.log(caja);
console.log(caja.innerHTML);
console.log(caja.innerText);
```

- `innerHTML`: Devuelve el HTML interno del elemento, incluyendo etiquetas y contenido.
- `innerText`: Devuelve solo el texto visible dentro del elemento.

Podemos modificar el contenido del elemento usando:

```
caja.innerText = "Cajas de Martín";
caja.innerHTML = "<h1>Cajas de Martín</h1>";
```

Esto cambia el texto y estructura del HTML interno en el elemento `caja`.

8.3 `getElementsByClassName()`

Este método selecciona todos los elementos que tienen una clase específica y devuelve una `HTMLCollection`, que es una lista de elementos.

```
let articulos = document.getElementsByClassName("articulo");
console.log(articulos);
console.log(articulos[0]);
```

Al ser una lista, podemos recorrer los elementos seleccionados:

```
for (let e of articulos) {
  e.style.border = "5px solid black";
  e.style.margin = "5px";
}
```

Para usar métodos de arreglos como `forEach`, debemos convertir la `HTMLCollection` a un arreglo usando `Array.from`:

```
let articulosArray = Array.from(articulos);
articulosArray.forEach((e) => {
  e.style.background = "green";
  e.innerHTML += "<a href='#'>enlace</a>";

  let h3 = document.createElement("h3");
  let texto = document.createTextNode("Subtítulo");
  h3.append(texto);
  e.append(h3);
});
```

Esto añade un enlace y un subtítulo en cada elemento con la clase `articulo`.

8.4 `getElementsByTagName()`

`getElementsByTagName` selecciona todos los elementos con una etiqueta HTML específica, como `div`, `p` o `article`, y devuelve otra `HTMLCollection`.

```
let articulosEtiqueta = document.getElementsByTagName("article");
console.log(articulosEtiqueta);
console.log(articulosEtiqueta[0]);
```

Para recorrer estos elementos, usamos un bucle `for`:

```
for (let e of articulosEtiqueta) {
  console.log("Artículo por etiqueta:", e);
}
```

Este método es útil cuando queremos seleccionar y manipular todos los elementos de un tipo específico en el documento.

8.5 `querySelector()`

`querySelector` selecciona el **primer elemento** que coincide con un selector CSS dado, como `#id`, `.clase` o una etiqueta (`div`, `p`).

```
let seleccionarPorIdentificador = document.querySelector("#caja");
console.log(seleccionarPorIdentificador);

let seleccionarPorClase = document.querySelector(".articulo");
console.log(seleccionarPorClase);

let seleccionarPorEtiqueta = document.querySelector("article");
console.log(seleccionarPorEtiqueta);
```

Con este método, podemos seleccionar elementos de forma similar a CSS, pero **solo selecciona el primer elemento coincidente**.

8.6 `querySelectorAll()`

Este método selecciona **todos los elementos** que coinciden con un selector CSS y devuelve una `NodeList`, que podemos recorrer fácilmente con `forEach`.

```
let seleccionarPorClaseTodos = document.querySelectorAll(
  "#articulos .articulo"
);
console.log(seleccionarPorClaseTodos);

let seleccionarPorEtiquetaTodos = document.querySelectorAll("article");
console.log(seleccionarPorEtiquetaTodos);
```

A diferencia de `querySelector`, `querySelectorAll` selecciona **todos los elementos** que coinciden con el selector. Esto es ideal para aplicar estilos o contenido a múltiples elementos de una sola vez. Aquí usamos `forEach` para añadir identificadores únicos a cada `article` seleccionado:

```
seleccionarPorEtiquetaTodos.forEach((e, indice) => {
  let identificador = "idArticulo_".concat(indice);
  e.setAttribute("id", identificador);
  console.log("QuerySelectorAll:", e, "índice:", indice);
});
```

9. El BOM (Browser Object Model)

El **Browser Object Model** (BOM) permite a JavaScript interactuar con el navegador y sus características, como la ventana (window), el historial (history), la ubicación (location), la información del navegador (navigator), y más. A diferencia del **DOM**, que se centra en la estructura del documento HTML, el BOM proporciona herramientas para manipular el entorno de navegación.

9.1 El objeto `window`

El objeto `window` es el contenedor principal del BOM, que representa la ventana del navegador y engloba todos los demás objetos, como `document`, `history`, `location`, y `navigator`. A través de `window`, podemos acceder a una gran variedad de métodos y propiedades útiles para interactuar con el navegador.

```
console.log(window); // Imprime el objeto window completo
console.log("Ancho de la ventana", window.innerWidth); // Ancho en píxeles de la
ventana
console.log("Alto de la ventana", window.innerHeight); // Alto en píxeles de la
ventana
console.log("Sabemos si la ventana está abierta o cerrada", window.closed); //
Verifica si la ventana está abierta o cerrada
console.log("DOM", window.document); // Acceso al documento DOM
console.log("Sabemos el navegador", window.navigator.userAgent); // Muestra el
navegador y sistema operativo
```

- `window.innerWidth` **y** `window.innerHeight`: Devuelven el ancho y alto de la ventana en píxeles.
- `window.closed`: Indica si la ventana está cerrada (`true`) o abierta (`false`).
- `window.navigator.userAgent`: Proporciona información sobre el navegador y el sistema operativo del usuario.

9.2 Métodos de interacción: `alert`, `confirm`, y `prompt`

Estos métodos muestran cuadros de diálogo interactivos para que el usuario pueda responder a mensajes.

```
alert("¡Hola a todos!"); // Muestra un mensaje en un cuadro de alerta

let respuesta = prompt("¿Cómo estás?"); // Muestra un cuadro de entrada y guarda
la respuesta
console.log("Respuesta del usuario:", respuesta);

let confirmacion = confirm("¿Te fue bien el día?"); // Pide al usuario confirmar o
cancelar
console.log("Confirmación del usuario:", confirmacion ? "Sí" : "No");
```

- `alert()`: Muestra un cuadro de alerta con un mensaje.
- `prompt()`: Muestra un cuadro de diálogo para que el usuario ingrese un texto. El valor ingresado se guarda en una variable.
- `confirm()`: Muestra un cuadro de diálogo con opciones para confirmar o cancelar. Devuelve `true` o `false` según la elección del usuario.

9.3 El objeto `navigator`

El objeto `navigator` proporciona información sobre el navegador, el sistema operativo y el estado de la conexión de red.

```
console.log("Objeto navigator", navigator); // Muestra toda la información del navegador
console.log("Conexión a internet", navigator.onLine); // Verifica si hay conexión a internet
if (!navigator.onLine) {
  alert("No tienes conexión a Internet"); // Informa al usuario si no hay conexión
}
```

- `navigator.userAgent`: Devuelve una cadena que describe el navegador y sistema operativo del usuario.
- `navigator.onLine`: Indica si el navegador está conectado a internet (`true`) o no (`false`).

> **Opciones adicionales**:
>
> - `navigator.language`: Muestra el idioma del navegador (por ejemplo, `es-ES` para español).
> - `navigator.platform`: Indica el sistema operativo en el que se ejecuta el navegador.

9.4 El objeto `location`

`location` permite acceder y modificar la URL actual, además de manejar redirecciones y recargar la página.

```
console.log("Objeto location", location); // Muestra toda la información de la URL
console.log("Dirección del navegador", location.href); // URL completa de la página actual
// location.reload(); // Recarga la página
```

- `location.href`: Devuelve o establece la URL actual.
- `location.reload()`: Recarga la página actual.

> **Opciones adicionales**:
>
> - `location.protocol`: Devuelve el protocolo (`http:` o `https:`).
> - `location.host`: Devuelve el nombre del dominio.
> - `location.pathname`: Devuelve la ruta de la página actual.
> - `location.search`: Muestra la cadena de consulta (parámetros GET).

9.5 El objeto `history`

`history` permite acceder al historial de navegación del usuario. A través de este objeto, podemos navegar hacia atrás o adelante en el historial, o saltar a una posición específica.

```
console.log("Objeto history", window.history); // Muestra el historial del
navegador
console.log("Página anterior en el historial");
// window.history.back(); // Regresa una página en el historial
console.log("Página siguiente en el historial");
// window.history.forward(); // Avanza una página en el historial
```

- `history.back()`: Navega a la página anterior en el historial.
- `history.forward()`: Navega a la página siguiente en el historial.

> **Opción adicional**:
>
> - `history.go(n)`: Navega a una posición en el historial (`n` puede ser un número positivo para avanzar o negativo para retroceder).

9.6 Temporizadores: `setTimeout` y `setInterval`

`setTimeout` y `setInterval` son métodos del objeto `window` que permiten ejecutar código después de un cierto tiempo.

- `setTimeout`: Ejecuta una función una sola vez después de un intervalo de tiempo especificado.
- `setInterval`: Ejecuta una función repetidamente en intervalos de tiempo especificados.

```
setTimeout(() => {
  console.log("Este mensaje aparece después de 2 segundos.");
}, 2000);

let contador = 0;
let intervalo = setInterval(() => {
  contador++;
  console.log("Contador:", contador);
  if (contador === 5) {
    clearInterval(intervalo); // Detiene el intervalo después de 5 iteraciones
  }
}, 1000);
```

- `clearInterval(intervalo)`: Detiene un intervalo específico.

10. Eventos en JavaScript

Los eventos en JavaScript permiten ejecutar código en respuesta a acciones del usuario, como hacer clic en un botón, pasar el ratón sobre un elemento, escribir en un campo de texto, entre otras. Con `addEventListener`, puedes asignar un evento a un elemento y definir la función que se ejecutará cuando dicho evento ocurra.

10.1 Eventos de Clic

- `click`: Se dispara cuando se hace clic en un elemento.

```
let boton = document.querySelector("#boton");

boton.addEventListener("click", (event) => {
  alert("Acabas de pulsar en el botón.");
  console.log(event); // Muestra información sobre el evento
});
```

- `dblclick`: Ocurre cuando se hace doble clic sobre un elemento.

```
let boton2 = document.querySelector("#boton2");

boton2.addEventListener("dblclick", (event) => {
  alert("Pulsaste dos veces en el botón.");
  console.log(event); // Información del evento
});
```

10.2 Eventos del Ratón

Estos eventos se activan en respuesta a movimientos o interacciones del ratón sobre un elemento.

- `mouseover`: Se activa cuando el puntero pasa sobre un elemento.

```
let cajaTexto = document.querySelector("#cajaTexto");

cajaTexto.addEventListener("mouseover", (event) => {
  alert("Pasaste por encima.");
  console.log(event);
});
```

- `mousemove`: Ocurre cuando el ratón se mueve sobre un elemento.

```
cajaTexto.addEventListener("mousemove", (event) => {
  console.log("Me muevo en el textarea.");
});
```

- `mouseout`: Se dispara cuando el puntero sale del área de un elemento.

```
cajaTexto.addEventListener("mouseout", (event) => {
  alert("Saliste del textarea.");
  console.log(event);
});
```

10.3 Eventos de Teclado

Los eventos de teclado responden a interacciones con las teclas.

- `keydown`: Se activa cuando se presiona una tecla.

```
let campoTexto = document.querySelector("#campoTexto");

campoTexto.addEventListener("keydown", (event) => {
  console.log("Pulsaste la tecla:", event.key);
});
```

- `keyup`: Ocurre cuando se suelta una tecla.

```
campoTexto.addEventListener("keyup", (event) => {
  console.warn("Soltaste la tecla:", event.key);
});
```

- `keypress`: Se activa mientras se mantiene una tecla presionada.

```
campoTexto.addEventListener("keypress", (event) => {
  console.error("Mantienes la tecla:", event.key);
});
```

10.4 Eventos de Formularios

Los eventos de formulario son importantes para manejar interacciones y validaciones en formularios.

- `submit`: Se activa cuando se intenta enviar un formulario. Se usa `event.preventDefault()` para evitar el envío automático.

```
let formulario = document.querySelector("#formulario");

formulario.addEventListener("submit", (event) => {
  event.preventDefault(); // Evita el envío automático
  let nombre = document.querySelector("#nombre").value;
  let correo = document.querySelector("#correo").value;
  let genero = document.querySelector("#genero").value;
  alert(nombre);
  alert(correo);
  alert(genero);
});
```

- `change`: Ocurre cuando se selecciona un valor nuevo en un elemento `<select>`.

```
let selectGenero = document.querySelector("#genero");
selectGenero.addEventListener("change", () => {
  console.log("Has cambiado el valor del select a:", selectGenero.value);
});
```

- `focus` y `blur`: `focus` se dispara al entrar en un campo de entrada; `blur` al salir de él.

```
let correo = document.querySelector("#correo");

correo.addEventListener("focus", () => {
  console.log("Acabas de entrar al campo de email:", correo.value);
});

correo.addEventListener("blur", () => {
  console.warn("Acabas de salir del foco:", correo.value);
});
```

10.5 Eventos de Ventana y Documento

Algunos eventos ocurren cuando el documento o la ventana del navegador alcanzan ciertos estados de carga.

- `DOMContentLoaded`: Se dispara cuando el contenido HTML está completamente cargado y procesado. Es útil para asegurarse de que el DOM esté disponible antes de ejecutar cualquier script.

```
document.addEventListener("DOMContentLoaded", () => {
  console.log("Se lanza cuando todo el DOM está cargado");
});
```

- `load`: Se activa cuando toda la página, incluidos recursos como imágenes y estilos CSS, han sido completamente cargados.

```
window.addEventListener("load", () => {
  console.log(
    "Toda la página, recursos, imágenes... se han cargado aparte del DOM"
  );
});
```

11. Programación Orientada a Objetos (POO) en JavaScript

La Programación Orientada a Objetos (POO) en JavaScript permite estructurar el código en torno a "objetos" que representan entidades con características y comportamientos. Esto facilita la organización, reutilización y escalabilidad del código, especialmente en proyectos grandes.

11.1 Conceptos Básicos

- **Objeto**: Un conjunto de propiedades y métodos que representan una entidad. En JavaScript, un objeto se define usando llaves `{}` y puede contener pares clave-valor.

```
let persona = {
  nombre: "Juan",
  edad: 30,
  saludar: function () {
    console.log(`Hola, me llamo ${this.nombre}`);
  },
};

persona.saludar(); // "Hola, me llamo Juan"
```

- **Clase**: Es el molde o plantilla para crear múltiples objetos similares. Una clase define las propiedades y métodos que tendrán los objetos instanciados a partir de ella.

```
class Persona {
  constructor(nombre, edad) {
    this.nombre = nombre;
    this.edad = edad;
  }

  saludar() {
    console.log(`Hola, me llamo ${this.nombre}`);
  }
}
```

11.2 Crear y Usar Clases

En JavaScript, se pueden crear clases utilizando la palabra clave `class` y un constructor (`constructor`), que define las propiedades iniciales del objeto.

```
class Persona {
  constructor(nombre, edad) {
    this.nombre = nombre;
    this.edad = edad;
  }

  // Método
  saludar() {
    console.log(`Hola, me llamo ${this.nombre} y tengo ${this.edad} años`);
  }
}

// Crear una instancia de la clase
let persona1 = new Persona("Carlos", 25);
persona1.saludar(); // "Hola, me llamo Carlos y tengo 25 años"
```

11.3 Métodos y Propiedades de Instancia

- **Propiedades de Instancia**: Son las variables definidas en el `constructor` de una clase y que pertenecen a cada instancia de la clase.
- **Métodos de Instancia**: Son funciones definidas dentro de una clase que actúan sobre los datos de esa instancia específica.

```
class Coche {
  constructor(marca, modelo) {
    this.marca = marca;
    this.modelo = modelo;
  }

  arrancar() {
    console.log(`${this.marca} ${this.modelo} está en marcha`);
  }
}

let coche1 = new Coche("Toyota", "Corolla");
coche1.arrancar(); // "Toyota Corolla está en marcha"
```

11.4 Herencia

La herencia permite crear una clase basada en otra clase. Esto significa que la clase hija hereda las propiedades y métodos de la clase padre, pero también puede tener sus propios métodos y propiedades.

```
class Animal {
  constructor(nombre) {
    this.nombre = nombre;
  }

  hacerSonido() {
    console.log(`${this.nombre} hace un sonido`);
  }
}

class Perro extends Animal {
  hacerSonido() {
    console.log(`${this.nombre} ladra`);
  }
}

let perro = new Perro("Rex");
perro.hacerSonido(); // "Rex ladra"
```

11.5 Encapsulamiento

El encapsulamiento oculta los detalles internos de una clase y permite acceder a las propiedades y métodos solo a través de una API pública.

- **Propiedades privadas**: Se utilizan para evitar el acceso directo desde fuera de la clase. En JavaScript, una propiedad privada se define con un # antes de su nombre.

```
class CuentaBancaria {
  #saldo;

  constructor(saldoInicial) {
    this.#saldo = saldoInicial;
  }

  depositar(cantidad) {
    this.#saldo += cantidad;
    console.log(`Nuevo saldo: ${this.#saldo}`);
  }

  // Getter
  verSaldo() {
    return this.#saldo;
  }
}

let cuenta = new CuentaBancaria(100);
cuenta.depositar(50); // "Nuevo saldo: 150"
console.log(cuenta.verSaldo()); // 150
```

11.6 Polimorfismo

El polimorfismo permite que métodos con el mismo nombre en clases diferentes puedan tener implementaciones específicas. En JavaScript, el polimorfismo se usa frecuentemente en la herencia, donde una clase hija puede sobrescribir métodos de la clase padre.

```
class Animal {
  sonido() {
    console.log("Este animal hace un sonido.");
  }
}

class Gato extends Animal {
  sonido() {
    console.log("El gato maúlla");
  }
}

let animal = new Animal();
animal.sonido(); // "Este animal hace un sonido."
```

```
let gato = new Gato();
gato.sonido(); // "El gato maúlla"
```

11.7 Propiedades y Métodos Estáticos

Las propiedades y métodos estáticos pertenecen a la clase en sí, no a las instancias de la clase. Se definen usando la palabra clave `static` y se acceden directamente desde la clase.

```
class Matematica {
  static sumar(a, b) {
    return a + b;
  }
}

console.log(Matematica.sumar(5, 3)); // 8
```

Claro, voy a dividir el código en pequeñas secciones y proporcionar explicaciones detalladas para facilitar la comprensión. Comencemos con los apuntes desglosados:

12. Objetos en JavaScript y JSON

12.1 Objetos en Javascript

12.1.1 Creación y uso de objetos literales

Un **objeto literal** es una estructura que permite almacenar propiedades (datos) y métodos (funciones) bajo un solo nombre.

Ejemplo de un objeto literal:

```
let pelicula = {
  titulo: "terminator", // Propiedad: título de la película
  anho: 1984, // Propiedad: año de lanzamiento
  genero: "ciencia ficción", // Propiedad: género de la película
  protagonista: "Arnold Schwarzenegger", // Propiedad: protagonista principal
  elenco: ["Arnold", "Linda Hamilton", "Resto"], // Propiedad: array de actores

  // Método: función que describe la película usando `this` para acceder a sus
propiedades
  descripcion: function () {
    return `${this.titulo} es una peli de ${this.genero}, protagonizada por
${this.protagonista} y lanzada en el año ${this.anho}`;
  },

  // Método: función flecha que describe la película, accediendo a las propiedades
a través del nombre del objeto
  descripcionDos: () => {
    return `${pelicula.titulo} es una peli de ${pelicula.genero}, protagonizada
```

```
por ${pelicula.protagonista} y lanzada en el año ${pelicula.anho}`;
  },

  // Método que muestra los actores del elenco en la consola
  mostrarElenco: () => {
    console.log("Elenco de actores");
    pelicula.elenco.forEach((nombre) => {
      console.log(nombre); // Itera sobre el array `elenco` y muestra cada nombre
    });
  },

  // Propiedad anidada que representa detalles adicionales de la película
  detalles: {
    duracion: "120 minutos",
    pegi: 18,
  },
};
```

Explicación de métodos y propiedades:

- **Propiedades**: Son variables asociadas al objeto, como `titulo`, `anho`, y `genero`.
- **Métodos**: Son funciones definidas dentro del objeto, como `descripcion` y `mostrarElenco`.
- `this`: Se utiliza dentro de un método para referirse al propio objeto.

12.1.2 Acceso y modificación de propiedades

Puedes acceder a las propiedades de un objeto de dos maneras:

Notación de punto:

```
console.log(pelicula.protagonista); // Muestra: "Arnold Schwarzenegger"
```

Notación de corchetes:

```
console.log(pelicula["genero"]); // Muestra: "ciencia ficción"
```

Agregar nuevas propiedades:

```
pelicula.director = "Martín Gil"; // Añade una nueva propiedad `director`
console.log(pelicula.director); // Muestra: "Martín Gil"
```

Modificar propiedades existentes:

```
pelicula.secuela = "Terminator 2: El juicio final"; // Modifica o añade la
propiedad `secuela`
```

```
console.log(pelicula.secuela); // Muestra: "Terminator 2: El juicio final"
```

12.1.3 Métodos y eliminación de propiedades

Llamar a un método:

```
console.log(pelicula.descripcion()); // Ejecuta y muestra el resultado del método `descripcion`
```

Eliminar una propiedad:

```
console.log(pelicula.detalles); // Muestra: { duracion: "120 minutos", pegi: 18 }
delete pelicula.detalles; // Elimina la propiedad `detalles`
console.log(pelicula.detalles); // Muestra: `undefined` ya que se ha eliminado
```

12.2 JSON (JavaScript Object Notation)

JSON es un formato de texto utilizado para representar objetos de forma estructurada. A diferencia de los objetos de JavaScript, **JSON**:

- No permite funciones como valores.
- Solo permite tipos de datos primitivos, arrays y otros objetos anidados.

Ejemplo de un objeto en formato JSON:

```
let palaPadel = {
  nombre: "Metalbone",
  marca: "Adidas",
  anho: 2022,
  peso: 350,
  forma: "Diamante",
  dureza: "Media",
};

console.log(palaPadel); // Muestra el objeto `palaPadel` completo
```

Conversión de un objeto a JSON y viceversa:

```
let palaDePadelJSON = JSON.stringify(palaPadel); // Convierte el objeto `palaPadel` en una cadena JSON
console.log(palaDePadelJSON); // Muestra el objeto en formato JSON como texto

let objetoConvertido = JSON.parse(palaDePadelJSON); // Convierte la cadena JSON de
```

```
nuevo a un objeto JavaScript
console.log(objetoConvertido); // Muestra el objeto JavaScript convertido
```

12.2.1 Recorrido de un objeto con `for...in`

Puedes iterar sobre las propiedades de un objeto con el bucle `for...in`:

```
let caja = document.querySelector("#datos"); // Selecciona un elemento HTML con id
`datos`

console.log("recorrido de JSON u objeto");
for (let clave in palaPadel) {
  console.log(clave, ":", palaPadel[clave]); // Muestra el nombre de la propiedad
y su valor
  caja.innerHTML += `<p>${clave}: ${palaPadel[clave]}</p>`; // Agrega la propiedad
y valor al contenido de `caja`
}
```

Explicación:

- `for...in`: Recorre todas las propiedades enumerables de un objeto, mostrando tanto la clave (nombre de la propiedad) como su valor.

13. Cookies, `sessionStorage` y `localStorage` en JavaScript

En JavaScript, existen diferentes métodos para almacenar información en el navegador del usuario, cada uno con características y usos específicos. A continuación, se detallan que son las **cookies**, el `sessionStorage` y el `localStorage`, con ejemplos de código para una mejor comprensión.

13.1. Cookies

Las **cookies** son pequeños fragmentos de datos que un sitio web almacena en el navegador del usuario. Se utilizan comúnmente para recordar información entre visitas o para rastrear la actividad del usuario en el sitio.

Características de las cookies:

- Tamaño limitado (generalmente hasta 4KB).
- Tienen una fecha de expiración que determina su duración.
- Pueden ser leídas por el servidor y el cliente.

Ejemplo de creación, lectura y eliminación de cookies:

```
// Crear una cookie
document.cookie =
  "nombreUsuario=Juan; expires=Fri, 01 Jan 2025 12:00:00 UTC; path=/";

// Explicación:
// - `nombreUsuario=Juan`: Define la clave y el valor de la cookie.
```

```
// - `expires`: Fecha de expiración de la cookie.
// - `path=/`: Indica que la cookie es accesible en todo el sitio web.

// Leer cookies
console.log(document.cookie); // Muestra todas las cookies almacenadas en una
cadena de texto.

// Eliminar una cookie (se establece una fecha de expiración pasada)
document.cookie =
  "nombreUsuario=; expires=Thu, 01 Jan 1970 00:00:00 UTC; path=/";
```

Con las cookies se pueden enviar desde JavaScript al servidor y leerlas, por ejemplo en PHP usando `$_COOKIE`. Una vez que configuras una cookie en el navegador con JavaScript, se enviará automáticamente al servidor en las solicitudes HTTP.

1. **Crear una cookie en JavaScript**:

   ```
   document.cookie =
     "usuario=Juan; expires=" +
     new Date(Date.now() + 86400000).toUTCString() +
     "; path=/";
   ```

2. **Leer la cookie en el servidor PHP**:

   ```
   <?php
   if (isset($_COOKIE['usuario'])) {
       echo "El valor de la cookie 'usuario' es: " . $_COOKIE['usuario'];
   } else {
       echo "La cookie 'usuario' no está definida.";
   }
   ?>
   ```

Explicación:

- La cookie creada en JavaScript se enviará automáticamente al servidor en cada solicitud.
- En PHP, puedes acceder al valor de la cookie usando `$_COOKIE['nombre_cookie']`.

Nota: Las cookies no son seguras para almacenar datos sensibles y pueden ser vistas y manipuladas si el navegador lo permite.

Nota 2: Para una creacción y uso más sencillo de las cookies se pueden usar funciones como las siguientes.

```
/*Función para establecer el valor de la cookie*/
function setCookie(name, value, days) {
  let expires = "";

  if (days) {
```

```
    let date = new Date();
    date.setTime(date.getTime() + days * 24 * 60 * 60 * 1000);
    expires = "; expires=" + date.toUTCString();
  }

  document.cookie = name + "=" + (value || "") + expires + "; path=/";
}

/*Función para obtener el valor de la cookie*/
function getCookie(name) {
  let nameEQ = name + "=";
  let ca = document.cookie.split(";");

  for (let i = 0; i < ca.length; i++) {
    let c = ca[i];
    while (c.charAt(0) == " ") c = c.substring(1, c.length);
    if (c.indexOf(nameEQ) == 0) return c.substring(nameEQ.length, c.length);
  }

  return null;
}
```

13.2. `sessionStorage`

El `sessionStorage` es una forma de almacenamiento web que permite guardar datos de forma temporal. La información se elimina cuando se cierra la pestaña o el navegador.

Características del `sessionStorage`:

- Los datos solo persisten mientras dure la sesión del navegador.
- No se comparten entre pestañas o ventanas diferentes.
- Ideal para guardar datos que solo se necesitan mientras se navega por la página.

Ejemplo de uso de `sessionStorage`:

```
// Guardar datos en sessionStorage
sessionStorage.setItem("nombre", "Ana");

// Leer datos de sessionStorage
let nombreGuardado = sessionStorage.getItem("nombre");
console.log(nombreGuardado); // Muestra: "Ana"

// Eliminar un elemento de sessionStorage
sessionStorage.removeItem("nombre");

// Limpiar todo el sessionStorage
sessionStorage.clear();
```

Explicación del código:

- `setItem("clave", "valor")`: Guarda un dato con la clave y valor especificados.
- `getItem("clave")`: Recupera el valor asociado a la clave especificada.
- `removeItem("clave")`: Elimina el dato asociado a la clave especificada.
- `clear()`: Elimina todos los datos almacenados en `sessionStorage`.

13.3. `localStorage`

El `localStorage` es similar al `sessionStorage`, pero con la diferencia de que los datos almacenados persisten incluso después de cerrar el navegador. Esto hace que sea ideal para guardar configuraciones de usuario o preferencias que deben durar más tiempo.

Características del `localStorage`:

- Los datos persisten indefinidamente hasta que se eliminen manualmente.
- Se comparte entre pestañas y ventanas del mismo dominio.
- Ideal para guardar datos que deben mantenerse entre sesiones.

Ejemplo de uso de `localStorage`:

```
// Guardar datos en localStorage
localStorage.setItem("tema", "oscuro");

// Leer datos de localStorage
let temaGuardado = localStorage.getItem("tema");
console.log(temaGuardado); // Muestra: "oscuro"

// Eliminar un elemento de localStorage
localStorage.removeItem("tema");

// Limpiar todo el localStorage
localStorage.clear();
```

Explicación del código:

- `setItem("clave", "valor")`: Guarda un dato con la clave y valor especificados en `localStorage`.
- `getItem("clave")`: Recupera el valor asociado a la clave.
- `removeItem("clave")`: Elimina el dato especificado.
- `clear()`: Elimina todos los datos de `localStorage`.

13.4. Comparación entre Cookies, `sessionStorage` y `localStorage`

Característica	Cookies	`sessionStorage`	`localStorage`
Persistencia	Depende de la expiración	Solo hasta que se cierre la pestaña	Indefinida (hasta eliminación manual)
Capacidad	~4KB	~5-10MB	~5-10MB
Alcance	Cliente y servidor	Solo cliente	Solo cliente

Característica	Cookies	`sessionStorage`	`localStorage`
Propósito	Sesiones, autenticación	Datos temporales de la sesión	Datos persistentes como preferencias

Conclusión: Utiliza **cookies** para datos que necesitan ser enviados al servidor, `sessionStorage` para datos temporales que no deben persistir después de cerrar la sesión y `localStorage` para datos que deben durar más tiempo y no requieren comunicación con el servidor.

14. Asincronía en JavaScript

JavaScript es un lenguaje de programación **síncrono** por defecto, lo que significa que ejecuta una instrucción tras otra de manera secuencial. Sin embargo, JavaScript también permite manejar **asincronía** mediante funciones especiales, lo cual es fundamental para realizar tareas como solicitudes de red, temporizadores, o manejo de archivos sin bloquear el flujo principal del programa.

14.1. Sincronía en JavaScript

En código **síncrono**, cada línea se ejecuta de forma secuencial y se espera a que termine para pasar a la siguiente línea.

- **Ejemplo de código síncrono:**

```
console.log("Hola");
console.log("Medio");
console.log("Adios");
```

- **Explicación**:
  - Cada `console.log` se ejecuta uno tras otro, bloqueando el flujo del programa hasta que se completa cada línea.
  - Resultado:

```
Hola
Medio
Adios
```

14.2. Asincronía en JavaScript

En código **asíncrono**, ciertas funciones permiten ejecutar tareas sin bloquear el flujo principal del programa. JavaScript logra esto gracias a funciones como `setTimeout` y `setInterval`.

- **Ejemplo de código asíncrono con `setTimeout`:**

```
console.log("Hola");
setTimeout(() => {
```

```
  console.log("Medio");
}, 1000);
console.log("Adios");
```

- **Explicación**:
  - `console.log("Hola")` y `console.log("Adios")` se ejecutan de inmediato.
  - `setTimeout` coloca el `console.log("Medio")` en una "cola de espera" para que se ejecute después de 1 segundo (1000 ms), permitiendo que el flujo principal continúe.
  - Resultado:

```
Hola
Adios
Medio
```

- **Ejemplo de código asíncrono con `setInterval`:**

```
console.log("Hola");
setInterval(() => {
  console.log("Medio");
}, 1000);
console.log("Adios");
```

- **Explicación**:
  - `setInterval` ejecuta `console.log("Medio")` cada 1 segundo, sin detener el flujo principal.
  - Resultado:

```
Hola
Adios
Medio
Medio
Medio
...
```

14.3. Diferencias entre Sincronía y Asincronía

Característica	Sincronía	Asincronía
Ejecución	Secuencial	Puede ejecutarse en paralelo
Bloqueo del flujo	Bloquea el flujo hasta completarse	No bloquea el flujo

15. Promesas y Peticiones fetch y AJAX

En JavaScript, las **promesas** son un mecanismo que permite manejar operaciones asíncronas, como peticiones a servidores, de forma más ordenada. Las promesas representan el resultado de una operación que puede completarse en el futuro y tener éxito (ser **resuelta**) o fallar (ser **rechazada**). Usar promesas facilita trabajar con resultados asíncronos y mejora la legibilidad del código.

15.1. Concepto de Promesa

Una **promesa** en JavaScript es un objeto que puede estar en uno de estos tres estados:

- **Pending** (pendiente): el estado inicial, cuando la promesa aún no ha terminado.
- **Fulfilled** (cumplida): la promesa se ha resuelto correctamente y se llama a la función `resolve`.
- **Rejected** (rechazada): la promesa ha fallado y se llama a la función `reject`.

Las promesas se crean usando el constructor `new Promise`, que recibe una función que a su vez recibe dos parámetros:

- **resolve**: una función que se llama cuando la promesa se resuelve exitosamente.
- **reject**: una función que se llama cuando la promesa falla.

15.2. Ejemplo de Promesa en JavaScript

A continuación, veremos un ejemplo de una función que devuelve una promesa llamada `eventoFuturo`:

```
const eventoFuturo = (res) => {
  return new Promise((resolve, reject) => {
    if (res === true) {
      resolve("Promesa resuelta");
    } else {
      reject("Promesa rechazada");
    }
  });
};
```

Explicación del Código

- `eventoFuturo` es una función que recibe un parámetro `res`.
- Dentro de esta función, se crea una nueva promesa con `new Promise`, que recibe una función con dos parámetros: `resolve` y `reject`.
- La función interna verifica el valor de `res`:
    - Si `res` es `true`, se llama a `resolve("Promesa resuelta")`, resolviendo la promesa con el mensaje `"Promesa resuelta"`.
    - Si `res` es `false`, se llama a `reject("Promesa rechazada")`, rechazando la promesa con el mensaje `"Promesa rechazada"`.

15.3. Uso de una Promesa

Para manejar el resultado de una promesa, se usan los métodos `.then` y `.catch`:

- `.then` se ejecuta si la promesa se resuelve correctamente.
- `.catch` se ejecuta si la promesa es rechazada.
- `.finally` se ejecuta siempre.

Ejemplo de uso:

```
const eventoFuturo = (res) => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      let pelicula = {
        titulo: "terminator",
        anho: 1984,
      };
      res === true ? resolve(pelicula) : reject("Error en la respuesta");
    }, 2000);
  });
};

let valor = true;

console.log(eventoFuturo(valor));

eventoFuturo(valor)
  .then((res) => {
    console.log(res);
  })
  .catch((res) => {
    console.log(res);
  })
  .finally(() => {
    console.log("Finalizó el proceso");
  });
```

Explicación del Ejemplo de Uso

1. **Primera llamada**: `eventoFuturo(true)`
    - Como `res` es `true`, la promesa se resuelve y ejecuta la función en `.then`, devuelve el objeto `pelicula`.
2. **Segunda llamada**: `eventoFuturo(false)`
    - Como `res` es `false`, la promesa es rechazada y ejecuta la función en `.catch`, mostrando el mensaje `"Promesa rechazada"`.

15.4. Peticiones AJAX a API y uso de fetch

- **AJAX (Asynchronous JavaScript and XML):** AJAX es una técnica que permite actualizar partes de una página web sin tener que recargarla completamente. Utilizando AJAX, el navegador puede comunicarse de manera asíncrona con el servidor, enviar y recibir datos en segundo plano y luego actualizar la página. Originalmente, AJAX se diseñó para trabajar con XML, pero hoy en día es común que maneje datos en formato JSON, que es más ligero y fácil de manejar en JavaScript. En el contexto de nuestro

ejemplo, aunque no se usa el objeto `XMLHttpRequest` (uno de los métodos más antiguos de AJAX), la función `fetch` emplea el mismo principio de comunicación asíncrona.

- **API (Application Programming Interface):** Una API es un conjunto de reglas que permite que una aplicación se comunique con otra. En el caso de las aplicaciones web, las APIs permiten a los navegadores o aplicaciones web solicitar datos de un servidor (backend) de manera estructurada y segura. Las APIs RESTful (basadas en el protocolo HTTP) son populares y envían datos en formato JSON.

Ejemplo Explicado

A continuación, se muestra una función `fetch` para obtener una lista de usuarios desde la URL dada y luego renderizar sus nombres en la interfaz.

```
const url_usuarios = "https://jsonplaceholder.typicode.com/users";

const buscar = (url, id_listado) => {
  fetch(url)
    .then((res) => res.json()) // Convierte la respuesta a JSON
    .then((data) => {
      let listado = document.querySelector(id_listado); // Selecciona el elemento
donde se insertará la lista
      data.forEach((usuario) => {
        const li = document.createElement("li");
        li.textContent = usuario.name; // Asigna el nombre del usuario
        listado.append(li); // Añade el elemento <li> al listado
      });
    });
};

// Llama a la función con la URL y el ID del elemento HTML donde se insertarán los
datos
buscar(url_usuarios, "#lista_usuarios");
```

Paso a Paso:

1. **Definición de URL y función**: Se define `url_usuarios` con la URL de la API y se crea la función `buscar`, que toma dos parámetros: la URL y el ID del contenedor en el DOM donde se mostrarán los resultados.

2. **Solicitud con `fetch`**: `fetch(url)` inicia la solicitud HTTP. La primera `.then((res) => res.json())` convierte la respuesta en JSON para que sea más fácil de usar en JavaScript.

3. **Recorrido y Renderizado de Datos**: `data.forEach((usuario) => { ... })` recorre el array de usuarios devuelto por la API. Para cada usuario:

    - Crea un elemento de lista (`li`).
    - Asigna el nombre del usuario como texto en el elemento `li`.
    - Agrega cada `li` al contenedor especificado en `id_listado`.