Mujeres Digitales 2025

JAVASCRIPT



AHK Deutsch-Kolumbianische Industrie- und Handelskammer
Cámara de Industria y Comercio Colombo-Alemana

Fomentado por:
Ministerio Federal de Economía y Energía
en virtud de una decisión del Bundestag alemán

ORGANIZACIÓN KEY



JavaScript

1. Introducción a JavaScript
   - 1.1. ¿Qué es JavaScript? pág 5
   - 1.2. ¿Para qué sirve JavaScript?
   - 1.3. Diferencias entre JavaScript y otros lenguajes de programación
   - 1.4. Inclusión de JavaScript en HTML pág 9
     - 1.4.1. JavaScript en el <head>
     - 1.4.2. JavaScript en el <body> pág 10
     - 1.4.3. JavaScript en un Archivo Externo pág 11
     - 1.4.4. ¿Dónde colocar la etiqueta <script>? pág 12
2. Variables y tipos de datos pág 13
   - 2.1. Declaración de variables (let, const, var)
   - 2.2. Tipos de datos primitivos (number, string, boolean, undefined, null) pág 14
     - 2.2.1. undefined pág 15
     - 2.2.2. null pág 16
   - 2.3. Tipos de datos complejos (array, object) pág 17
     - 2.3.1. array
     - 2.3.2. object pág 21
     - 2.3.3. Operador in pág 22
   - 2.4. Diferencias Entre Tipos Primitivos y Tipos Complejos pág 24
   - 2.5. Función typeof() pág 25
     - 2.5.1. ¿Qué es typeof?
     - 2.5.2. ¿Cómo se usa typeof?
     - 2.5.3. Tipos de Datos que Devuelve typeof pág 27
     - 2.5.4. Casos Especiales y Peculiaridades pág 28
   - 2.6. Valor NaN pág 29
3. Operadores pág 31
   - 3.1. Operadores Matemáticos
   - 3.2. Operadores de Asignación pág 33
   - 3.3. Operadores de Comparación pág 35
   - 3.4. Operadores Lógicos pág 37
     - 3.4.1. AND (&&)
     - 3.4.2. OR (||) pág 38
     - 3.4.3. NOT (!) pág 39
   - 3.5. String Operators pág 39



4. Condicionales pág 40
   4.1. Condicional if y sus variaciones (else, else if)
   4.2. Operador ternario (condición ? true : false) pág 42
   4.3. switch pág 43
5. Ciclos (bucles) pág 45
   5.1. Ciclo for
   5.2. Ciclo while pág 46
   5.3. Ciclo do while pág 47
   5.4. Variable i como iterador en los ciclos pág 48
6. Funciones pág 49
   6.1. Scope (Alcance)
   6.2. Diferencias entre Variable, Parámetro y Argumento pág 50
   6.3. Elemento this pág 54
      6.3.1. this dentro de un objeto pág 55
      6.3.2. this fuera de un objeto
      6.3.3. this en un constructor con new pág 56
      6.3.4. this en arrow functions
   6.4. Funciones flecha (arrow functions) pág 57
   6.5. Funciones anónimas pág 59
7. Características avanzadas en ES6 pág 63
   7.1. Destructuring
   7.2. Spread Operator pág 65
   7.3. Parámetros Rest (...rest) pág 66
   7.4. Template Strings pág 67
      7.4.1. Interpolación de Variables pág 68
      7.4.2. Interpolación de Expresiones
      7.4.3. Cadenas Multilínea
      7.4.4. Uso con Funciones pág 69
      7.4.5. Interpolación de Objetos
      7.4.6. Escapar los Backticks pág 70
      7.4.7. Tagged Template Literals
8. Eventos en JavaScript pág 71
   8.1. Tipos de eventos (click, mouseover, keydown, etc.)
   8.2. Manejo de eventos (addEventListener) pág 78
   8.3. Prevención de eventos (preventDefault) pág 82
9. DOM (Document Object Model) pág 83
   9.1. ¿Cómo funciona el DOM?



9.2. Interacción con el DOM usando JavaScript pág 84
9.2.1. Seleccionar Elementos en el DOM (getElementById, querySelector)
9.2.2. Agregar Elementos al DOM (innerHTML, textContent) pág 86
9.2.3. Modificar Elementos en el DOM (createElement, removeChild) pág 87
9.2.4. Eliminar Elementos del DOM (style) pág 88
9.2.5. Métodos Útiles del DOM
10. JSON en JavaScript pág 89
10.1. Convertir de objeto a JSON (JSON.stringify()) pág 90
10.2. Convertir de JSON a objeto (JSON.parse()) pág 91
10.3. Diferencia entre JSON y XML pág 92
10.4. Agregar, Eliminar y Modificar Datos de un JSON pág 93
10.5. JSON Anidados pág 95
10.6. Ventajas y desventajas de JSON pág 100
11. Expresiones Regulares pág 101
11.1. ¿Cómo se construye una expresión regular?
11.2. Funciones para trabajar con expresiones regulares en JavaScript pág 102
11.3. Caracteres especiales (metacaracteres) pág 103
11.4. Cuantificadores pág 104
11.5. Grupos de captura
11.6. Expresiones regulares avanzadas: Anclas y Lookahead
12. Almacenamiento en JavaScript pág 106
12.1. Cookies
12.2. LocalStorage pág 108
12.2.1. Métodos de LocalStorage (setItem(), getItem(), removeItem())
12.3. SessionStorage pág 110
12.4. Diferencias Cookies, LocalStorage y SessionStorage
13. Fundamentos de POO en JavaScript pág 111
13.1. Conceptos Básicos de la POO pág 112
13.1.1. Objetos
13.1.2. Clases
13.1.3. Constructor pág 113
13.1.4. Instancias pág 115
13.1.5. Funciones y Métodos



13.2. Principios de la Programación Orientada a Objetos pág 117
13.2.1. Encapsulamiento
13.2.2. Herencia pág 118
13.2.3. Polimorfismo pág 119
13.2.4. Abstracción pág 120
14. Introducción a la Asincronía en JavaScript pág 121
14.1. Callback
14.2. Promise pág 122
14.3. Async/Await pág 123
14.4. fetch() pág 124
14.4.1. Métodos HTTP pág 125
14.4.2. Convertir la respuesta pág 126
14.4.3. Uso de async/await con fetch() pág 128
14.5. Códigos de estado HTTP pág 129
14.5.1. Respuestas informativas (1xx)
14.5.2. Respuestas satisfactorias (2xx)
14.5.3. Redirecciones (3xx)
14.5.4. Errores del cliente (4xx) pág 130
14.5.5. Errores del servidor (5xx)
15. Manejo de errores en JavaScript pág 131
15.1. Estructura try, catch, finally
15.2. Lanzamiento de errores (throw) pág 133
16. Interactuando con APIs (API de Rick y Morty) pág 135
16.1. Uso de fetch() para realizar solicitudes
16.2. Uso de async/await para manejo de promesas pág 136
16.3. Manejo de Errores en Fetch pág 137
16.4. Gestión de Respuestas en JSON pág 138
16.5. Mostrar Personajes en una Página HTML pág 139



1. Introducción a JavaScript

1.1. ¿Qué es JavaScript?

JavaScript es un lenguaje de programación que se ejecuta en los navegadores web y que permite hacer las páginas web interactivas. Si HTML es el "esqueleto" de una página web y CSS es su "piel y ropa" (el diseño), JavaScript es lo que le da "vida" a la página, permitiendo que interactúe con el usuario. Se utiliza para crear dinámicas como formularios interactivos, juegos, notificaciones y cualquier acción que requiera una respuesta directa del usuario.

Analogía:

Piensa en una página web como un restaurante:

- HTML: Es la estructura del restaurante: las mesas, sillas, menús.
- CSS: Es el diseño y la decoración: los colores de las paredes, los manteles, etc.
- JavaScript: Son los camareros, quienes interactúan contigo, te traen la comida y toman tus pedidos. Sin JavaScript, el restaurante estaría vacío y sin actividad.

1.2. ¿Para qué se usa JavaScript?

- Interactividad en la web: Crear menús que se despliegan, formularios que validan datos sin recargar la página, etc.
- Controlar multimedia: Play/pause en videos o música.
- Manipulación del DOM: El DOM es la estructura de HTML. JavaScript puede modificar esa estructura (agregar, borrar o cambiar elementos).
- Juegos y gráficos interactivos: Con bibliotecas como Canvas o WebGL.
- Aplicaciones web complejas: Desde sistemas de chat en tiempo real hasta redes sociales.

1.3. Diferencias entre JavaScript y Otros Lenguajes de Programación

JavaScript es uno de los lenguajes de programación más populares y ampliamente utilizados, especialmente en el desarrollo web. A continuación, te ofrezco un análisis comparativo detallado entre JavaScript y otros lenguajes de programación para entender mejor sus características, ventajas, y diferencias.

JavaScript vs. Java

A pesar de sus nombres similares, JavaScript y Java son lenguajes muy diferentes en su propósito y características.



- Tipado:
  - JavaScript: Dinámico, débilmente tipado. No necesitas definir el tipo de una variable al declararla (por ejemplo, `let x = 10;` puede ser un número o luego asignarse como una cadena).
  - Java: Fuerte y estáticamente tipado. Cada variable debe ser declarada con un tipo específico (por ejemplo, `int x = 10;`).
- Compilación vs. Interpretación:
  - JavaScript: Es un lenguaje interpretado, lo que significa que el código se ejecuta directamente en el navegador o el entorno en tiempo real, sin necesidad de compilación previa.
  - Java: Es un lenguaje compilado. El código fuente es compilado en bytecode que luego es ejecutado por la Máquina Virtual de Java (JVM).
- Ámbito de uso:
  - JavaScript: Se usa principalmente para desarrollo web, ejecutándose en el navegador para crear páginas interactivas y dinámicas. Hoy en día, también se usa en el backend con Node.js.
  - Java: Es un lenguaje de propósito general que se utiliza tanto para aplicaciones de escritorio, web (con Java EE), móviles (Android) y sistemas empresariales.
- Paradigma:
  - JavaScript: Soporta programación orientada a objetos (POO), programación funcional y programación basada en eventos.
  - Java: Es más rígido en su enfoque de POO, con una clara estructura de clases y objetos.

JavaScript vs. Python

JavaScript y Python son ambos lenguajes interpretados y dinámicos, pero tienen diferencias clave.

- Sintaxis:
  - JavaScript: Tiene una sintaxis más compleja y tiende a ser menos legible que Python, con símbolos como llaves (`{}`) y punto y coma (`;`).
  - Python: Es conocido por su sintaxis clara y legible, donde la indentación es clave para estructurar el código, eliminando la necesidad de llaves.
- Uso:
  - JavaScript: Se utiliza principalmente en el desarrollo web, aunque con Node.js se ha expandido al backend y otros entornos.
  - Python: Es muy versátil, siendo utilizado en desarrollo web, ciencia de datos, inteligencia artificial, automatización de tareas, y más.



- Paradigmas de programación:
  - Ambos lenguajes soportan programación orientada a objetos y programación funcional. Sin embargo, Python se enfoca más en la simplicidad y en hacer que las tareas complejas sean fáciles de implementar.
- Manejo de errores:
  - JavaScript: Utiliza try...catch para el manejo de excepciones.
  - Python: Utiliza try...except, y su estructura es más fácil de leer y manejar.

JavaScript vs. C++

C++ es un lenguaje de propósito general, de bajo nivel y fuertemente tipado, mientras que JavaScript es dinámico y enfocado principalmente en aplicaciones web.

- Tipado:
  - JavaScript: Dinámico. No necesitas declarar el tipo de variable.
  - C++: Estático. Debes declarar explícitamente el tipo de cada variable (int, float, char, etc.).
- Control de Memoria:
  - JavaScript: Tiene gestión automática de memoria (recolección de basura).
  - C++: El programador debe gestionar manualmente la memoria, lo que puede hacer que sea más eficiente, pero también más propenso a errores como fugas de memoria.
- Compilación vs. Interpretación:
  - JavaScript: Interpretado en tiempo de ejecución.
  - C++: Compilado. El código se convierte en código máquina antes de ejecutarse, lo que lo hace más rápido en tiempo de ejecución que JavaScript.
- Velocidad:
  - C++: Generalmente, C++ es mucho más rápido que JavaScript debido a la compilación y la cercanía al hardware.
  - JavaScript: Es más lento en términos de rendimiento puro, ya que corre en un entorno interpretado.

JavaScript vs. Ruby

Ruby es un lenguaje de scripting orientado a objetos, conocido por su simplicidad y elegancia, mientras que JavaScript es más versátil para el desarrollo web.

- Sintaxis:
  - JavaScript: Más compleja, con muchas reglas y excepciones.
  - Ruby: Tiene una sintaxis simple, clara y fácil de leer, similar a Python.



- Usos:
    - JavaScript: Popular principalmente en el desarrollo web del lado del cliente y del servidor con Node.js.
    - Ruby: Se usa comúnmente con el framework Ruby on Rails, para desarrollo web del lado del servidor, pero su popularidad ha disminuido con el tiempo en comparación con JavaScript.
- Paradigmas de programación:
    - Ambos soportan programación orientada a objetos y funcional, pero Ruby se enfoca más en la productividad y simplicidad del código, mientras que JavaScript puede ser más versátil en su uso.

JavaScript vs. TypeScript

TypeScript es un superconjunto de JavaScript que añade tipado estático y otras funcionalidades avanzadas.

- Tipado:
    - JavaScript: Dinámico, no es necesario declarar tipos de datos.
    - TypeScript: Estático, debes declarar tipos de datos para las variables, funciones, etc.
- Errores en Tiempo de Ejecución vs. Tiempo de Compilación:
    - JavaScript: Muchos errores no se detectan hasta que el código se ejecuta.
    - TypeScript: Detecta errores antes de que el código sea ejecutado, ya que se compila en JavaScript.
- Compatibilidad:
    - TypeScript se compila a JavaScript, lo que significa que cualquier código de JavaScript válido también es código válido en TypeScript, con la ventaja de mayor robustez gracias al tipado.

JavaScript es un lenguaje único por su enfoque en el desarrollo web y su naturaleza interpretada y dinámica. Aunque comparte algunas similitudes con otros lenguajes, como Python y Ruby, en términos de flexibilidad, se distingue por su gran capacidad para interactuar con el DOM, crear aplicaciones web interactivas y ahora también manejar el backend con Node.js. A diferencia de lenguajes como Java o C++, JavaScript es mucho más flexible y fácil de usar, pero esto puede conllevar ciertas limitaciones en rendimiento y control de memoria.



1.4. Inclusión de JavaScript en HTML

1.4.1. JavaScript en el <head>

Puedes incrustar código JavaScript directamente en el <head> de tu documento HTML usando la etiqueta <script>. Este método coloca el código JavaScript en la parte superior del documento.

Ejemplo:

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Ejemplo de JavaScript en <head></title>
    <script>
        function saludar() {
            alert('Hola desde el <head>!');
        }
    </script>
</head>
<body>
    <button onclick="saludar()">Saludar</button>
</body>
</html>
```

Ventajas:

- El código JavaScript se carga y está disponible antes de que se cargue el contenido del <body>, lo cual puede ser útil para scripts que necesitan ejecutarse antes de que el contenido se muestre.

Desventajas:

- Puede ralentizar la carga de la página, ya que el navegador tiene que procesar el JavaScript antes de renderizar el contenido del <body>.
- No es ideal para scripts que interactúan con el contenido del <body>, ya que el contenido puede no estar completamente cargado aún.



1.4.2. JavaScript en el <body>

Otra forma de incluir JavaScript es insertarlo directamente en el <body> de tu documento HTML, usualmente al final del documento antes de la etiqueta </body>. Este método permite que el contenido HTML se cargue antes de ejecutar el JavaScript.

Ejemplo:

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Ejemplo de JavaScript en <body></title>
</head>
<body>
    <button onclick="saludar()">Saludar</button>
    <script>
        function saludar() {
            alert('Hola desde el <body>!');
        }
    </script>
</body>
</html>
```

Ventajas:

- El contenido del <body> se carga antes de que se ejecute el JavaScript, lo que evita errores donde el script intenta interactuar con elementos que aún no existen en el DOM.
- Puede mejorar el tiempo de carga percibido de la página.

Desventajas:

- Si el script es muy grande, puede ralentizar la carga del contenido, ya que el navegador tiene que procesar el JavaScript después de cargar el HTML.



1.4.3. JavaScript en un Archivo Externo

Una forma recomendada de incluir JavaScript es referenciar un archivo .js externo usando la etiqueta <script> con el atributo src. Esto mantiene el HTML limpio y facilita la gestión del código JavaScript.

Ejemplo Archivo HTML:

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Ejemplo de JavaScript Externo</title>
    <script src="script.js" defer></script>
</head>
<body>
    <button onclick="saludar()">Saludar</button>
</body>
</html>
```

Archivo script.js:

```javascript
function saludar() {
    alert('Hola desde el archivo externo!');
}
```

Ventajas:

- Mejora la organización y la legibilidad del código al separar el JavaScript del HTML.
- Permite que el navegador almacene en caché el archivo JavaScript, lo que puede mejorar el rendimiento en visitas posteriores.
- Facilita la reutilización del código JavaScript en múltiples páginas.

Desventajas:

- Si el archivo .js no se carga correctamente, el JavaScript no funcionará en la página.



- Puede ser necesario gestionar múltiples archivos .js, lo que podría complicar el mantenimiento si no se organiza adecuadamente.

1.4.4. ¿Dónde colocar la etiqueta <script>?

Colocar el <script> en el <head>

- Uso:
  - Para scripts que deben ejecutarse antes de que el contenido de la página se cargue (por ejemplo, scripts de configuración o de librerías que no interactúan directamente con el DOM).
- Consideración:
  - Debes usar el atributo defer o async para evitar que el script bloquee la renderización del contenido HTML.

Colocar el <script> al final del <body>

- Uso:
  - Para scripts que interactúan con el contenido de la página. Asegura que todo el HTML esté cargado antes de ejecutar el script, evitando errores de acceso a elementos que aún no existen.

Atributos defer y async

- defer:
  - Carga el archivo JavaScript en paralelo con el HTML, pero lo ejecuta solo después de que todo el HTML se haya parseado.

```
<script src="script.js" defer></script>
```

- async:
  - Carga el archivo JavaScript en paralelo con el HTML y lo ejecuta tan pronto como se haya descargado, sin esperar a que el HTML se termine de parsear.

```
<script src="script.js" async></script>
```

Diferencias

- defer garantiza que el script se ejecute en el orden en que aparece en el documento, después de que todo el HTML se haya cargado.
- async puede ejecutar los scripts en cualquier orden, según cuando se descarguen, lo cual puede ser útil para scripts independientes.



¿Cuándo Usar Cada Método?

- Incrustar JavaScript en el <head>: Útil para scripts que deben estar listos antes de que el contenido de la página se cargue, pero se debe usar defer para evitar bloquear el renderizado.
- Incrustar JavaScript al Final del <body>: Ideal para scripts que interactúan con el contenido de la página, asegurando que el contenido esté disponible antes de ejecutar el JavaScript.
- Referencia a un Archivo Externo: La mejor práctica para mantener el código organizado, mejorar la reutilización y permitir el almacenamiento en caché.

Conclusión

Elegir la ubicación adecuada para tus scripts de JavaScript en HTML depende de tus necesidades específicas. Si deseas que el script se ejecute antes de que el contenido de la página se cargue, incrústalo en el <head> y considera usar defer.

Si prefieres que el script interactúe con el contenido, colócalo al final del <body>. Para una mayor organización y reutilización, usa archivos externos. Cada método tiene sus ventajas y desventajas, y entender cómo y cuándo usarlos te permitirá crear páginas web más eficientes y fáciles de mantener.

2. Variables y Tipos de Datos

Las variables en JavaScript son contenedores que almacenan datos. Puedes pensar en ellas como una caja que almacena información, y cada caja tiene una etiqueta (nombre de la variable) para identificarla.

2.1. Declaración de variables:

- let: Se utiliza para declarar variables cuyo valor puede cambiar.
- const: Se utiliza para declarar variables cuyo valor es constante (no cambia).
- var: Fue la forma antigua de declarar variables antes de ECMAScript 6. Hoy en día se prefiere let y const.

Ejemplo:

```
let nombre = "María";  // Variable que puede cambiar
const edad = 25;       // Variable que no cambiará
var pais = "Colombia"; // Forma antigua de declarar variables
```



- Explicación:
    - let nombre: Esto crea una variable llamada nombre que almacena el valor "María".
    - const edad: Crea una constante llamada edad con valor 25. Una vez que se le asigna, no puedes cambiar este valor.
    - var pais: Aunque se sigue usando, var tiene algunos problemas de comportamiento que veremos más adelante, por eso se recomienda usar let o const.

Analogía:
Piensa en let como una taza de café que puedes rellenar cuando se vacía. En cambio, const es como una taza de cerámica pintada: una vez que eliges su diseño (el valor), no puedes cambiarlo.

2.2. Tipos de Datos Primitivos

Los tipos de datos primitivos son los valores más básicos que puede manejar JavaScript. Estos tipos de datos no son objetos y no tienen métodos. Los principales tipos de datos primitivos en JavaScript son:

- Números: Representa números, ya sean enteros o decimales. Los números se escriben directamente sin comillas.
    - Ejemplo:

```
let edad = 25;
let precio = 19.99;
console.log(edad, precio); // Output: 25 19.99
```

- Cadenas de texto (strings): Representa texto o una secuencia de caracteres. Los valores de tipo string se definen entre comillas simples (' ') o dobles (" ").
    - Ejemplo:

```
let saludo = "Hola, mundo!";
console.log(saludo); // Output: Hola, mundo!
```



- Booleanos: Representa un valor lógico que puede ser verdadero (true) o falso (false). Los valores booleanos se escriben directamente sin comillas.
    - Ejemplo:

```
let esMayor = true;
let esMenor = false;
console.log(esMayor, esMenor); // Output: true false
```

- undefined: Un valor que indica que una variable ha sido declarada pero no se le ha asignado un valor. Las variables no inicializadas tienen el valor undefined.
    - Ejemplo:

```
let nombre;
console.log(nombre); // Output: undefined
```

- null: Representa la ausencia intencionada de un valor o un valor vacío. Es un objeto especial que indica que una variable no tiene ningún valor. null es un valor literal que se usa sin comillas.
    - Ejemplo:

```
let resultado = null;
console.log(resultado); // Output: null
```

Vamos a profundizar en los tipos de datos undefined y null

2.2.1. undefined

undefined es un tipo de dato que indica que una variable ha sido declarada, pero aún no tiene un valor asignado. Es decir, su valor es "indefinido". También ocurre cuando intentas acceder a una propiedad de un objeto o una variable que no existe.



Analogía:
Imagina que tienes una caja vacía con una etiqueta que dice "misterio". Sabes que la caja existe, pero no sabes qué tiene adentro (porque no le has puesto nada). Esto es lo que sucede cuando una variable en JavaScript es undefined.

```
let nombre; // La variable "nombre" ha sido declarada pero no tiene valor.
console.log(nombre); // Imprime: undefined
```

```
let persona = {
  nombre: "Ana",
  edad: 25
};
console.log(persona.apellido); // No existe la propiedad "apellido"
```

- Explicación:
  - En el primer ejemplo, la variable nombre ha sido declarada, pero no se le ha asignado ningún valor, por lo que su valor es undefined.
  - En el segundo ejemplo, intentamos acceder a una propiedad (apellido) que no existe en el objeto persona, por lo que JavaScript devuelve undefined.

Cuando se usa:
- Cuando declaras una variable, pero aún no le has asignado un valor.
- Al intentar acceder a una propiedad o variable que no existe.

2.2.2. null

null es un valor que indica la ausencia intencionada de un valor. A diferencia de undefined, que significa que algo no ha sido definido, null se usa cuando se quiere indicar explícitamente que no hay un valor.

Analogía:
Imagina que tienes una caja etiquetada como "regalo", pero deliberadamente no le has puesto nada adentro. Sabes que no tiene nada porque decidiste dejarla vacía. Esto es como asignar null a una variable: estás diciendo que no contiene ningún valor a propósito.



Ejemplos:

```
let coche = null;
console.log(coche); // Imprime: null
```

```
let mascota = {
  nombre: "Max",
  tipo: null // Sabemos que no tiene un tipo definido aún
};
console.log(mascota.tipo); // Imprime: null
```

- Explicación:
  - En el primer ejemplo, estamos asignando explícitamente el valor null a la variable coche, indicando que no tiene ningún valor por ahora.
  - En el segundo ejemplo, la propiedad tipo del objeto mascota se establece en null porque, intencionadamente, no tiene un valor asignado.

Cuando se usa:

- Cuando intencionadamente quieres asignar un valor vacío a una variable.
- Cuando deseas indicar que algo no tiene un valor válido, pero no es un error o algo indefinido.

2.3. Tipos de Datos Complejos

Los tipos de datos complejos son aquellos que pueden contener múltiples valores y estructuras. A diferencia de los primitivos, los complejos son objetos y pueden contener múltiples propiedades y métodos.

2.3.1. Arreglo (arrays): Un array es una lista ordenada de elementos, que puede contener valores de cualquier tipo (incluyendo otros arrays y objetos). Los arrays se definen con corchetes ([ ]) y los valores están separados por comas.

- Ejemplo:

```
let frutas = ["Manzana", "Banana", "Naranja"];
console.log(frutas[0]); // Output: Manzana
```



Acceder a los elementos del array

Los arrays usan índices para acceder a los elementos, comenzando desde 0.

```
console.log(frutas[0]); // Imprime "manzana"
console.log(frutas[1]); // Imprime "banana"
```

Métodos de Arrays en JavaScript

Un array no solo es una lista de valores, sino que también tiene una serie de métodos que te permiten manipular sus datos. Vamos a ver algunos de los métodos más utilizados:

1. push()

Agrega uno o más elementos al final del array.

```
let frutas = ["manzana", "banana"];
frutas.push("pera");  // Ahora frutas es ["manzana", "banana", "pera"]
```

- Ventaja: Útil cuando necesitas agregar nuevos elementos sin cambiar los ya existentes.
- Analogía: Es como agregar un nuevo libro al final de una estantería. El último puesto siempre está libre para el siguiente.

2. pop()

Elimina el último elemento de un array y lo devuelve.

```
let frutas = ["manzana", "banana", "pera"];
let ultimaFruta = frutas.pop();  // "pera" fue eliminada
```

- Ventaja: Puedes recuperar el último elemento si lo necesitas después de eliminarlo.
- Analogía: Imagina que estás sacando la última prenda de una pila de ropa; siempre tomas la de arriba.

3. shift()

Elimina el primer elemento del array y lo devuelve.



```
let frutas = ["manzana", "banana", "pera"];
let primeraFruta = frutas.shift();  // "manzana" fue eliminada
```

- Ventaja: Ideal cuando solo te interesa remover el primer elemento de la lista.
- Analogía: Es como sacar el primer libro de una fila en una biblioteca. Los demás libros se desplazan hacia adelante.

4. unshift()

Agrega uno o más elementos al inicio del array.

```
let frutas = ["banana", "pera"];
frutas.unshift("manzana");  // frutas ahora es ["manzana", "banana", "pera"]
```

- Ventaja: Útil si necesitas agregar algo al principio sin modificar lo que ya estaba.
- Analogía: Colocar un nuevo libro al principio de la estantería, y mover todos los otros hacia la derecha.

5. slice()

Devuelve una copia de una parte del array, sin modificar el original.

```
let frutas = ["manzana", "banana", "pera", "naranja"];
let algunasFrutas = frutas.slice(1, 3);  // ["banana", "pera"]
```

- Ventaja: Útil cuando solo quieres una parte del array.
- Analogía: Piensa en esto como cortar una rebanada de pizza. No tocas toda la pizza, solo sacas la parte que te interesa.

6. splice()

Cambia el contenido de un array eliminando o añadiendo elementos.

```
let frutas = ["manzana", "banana", "pera", "naranja"];
frutas.splice(1, 2);  // Elimina desde el índice 1, dos elementos
```

- Ventaja: Permite eliminar y agregar elementos en una posición específica.



- Analogía: Es como quitar unas páginas del medio de un libro y reemplazarlas con nuevas.

7. forEach()

Ejecuta una función para cada elemento del array.

```
let frutas = ["manzana", "banana", "pera"];
frutas.forEach(function(fruta) {
  console.log(fruta);
});
// Imprime: "manzana", "banana", "pera"
```

- Ventaja: Útil para realizar acciones en cada elemento del array.
- Analogía: Es como recorrer cada casillero de una fila, abriéndolo y viendo qué hay dentro.

8. map()

Crea un nuevo array con los resultados de llamar a una función para cada elemento.

```
let numeros = [1, 2, 3];
let dobles = numeros.map(function(num) {
  return num * 2;
});
// dobles es [2, 4, 6], numeros sigue siendo [1, 2, 3]
```

- Ventaja: Útil para transformar cada elemento de un array.
- Analogía: Es como aplicar una transformación mágica a cada objeto de una lista, pero sin cambiar la lista original.

9. filter()

Crea un nuevo array con todos los elementos que cumplan con una condición.



```
let numeros = [1, 2, 3, 4, 5];
let pares = numeros.filter(function(num) {
  return num % 2 === 0;
});
// pares es [2, 4], el array numeros sigue igual
```

- Ventaja: Filtra elementos que cumplan con una condición, dejando intacto el array original.
- Analogía: Es como filtrar los ingredientes de una ensalada para quedarte solo con los que te gustan.

10. reduce()

Reduce todos los elementos de un array a un solo valor, aplicando una función.

```
let numeros = [1, 2, 3, 4];
let suma = numeros.reduce(function(total, num) {
  return total + num;
}, 0);
// suma es 10
```

- Ventaja: Útil para acumular valores (sumar, multiplicar, etc.).
- Analogía: Imagina que estás sumando una fila de monedas una por una para obtener el total.

2.3.2. Object (Objeto)

Un objeto es una colección de pares clave-valor, donde cada valor puede ser de cualquier tipo, incluidos otros objetos o arrays. Los objetos se definen con llaves ({ }) y cada par clave-valor está separado por comas. El valor puede ser de cualquier tipo de datos, incluidos otros objetos o funciones.

Existen varias formas de crear un objeto en JavaScript:



- Literal de objeto

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

Aquí, persona es un objeto con propiedades (nombre, edad, profesion) y un método (saludar).

- Objeto creado con new Object()

```
let persona = new Object();
persona.nombre = "Juan";
persona.edad = 30;
persona.profesion = "Desarrollador";
```

2.3.3 Operador in

in es un operador en JavaScript que se utiliza para verificar si una propiedad específica existe dentro de un objeto o si un índice existe dentro de un array.

Ejemplos:



- Con objetos:

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

console.log("nombre" in persona); // true
console.log("altura" in persona); // false
```

Aquí, el operador in está verificando si las propiedades nombre y altura existen en el objeto persona.

- Con arrays:

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

console.log(2 in numeros); // true (el índice 2 existe)
console.log(4 in numeros); // false (el índice 4 no existe, el array tiene 4 elementos
```

En este caso, in está verificando la existencia de un índice en el array, no el valor. Por ejemplo, el índice 2 (tercer elemento) sí existe en el array, pero el índice 4 no.

¿Se usa el ciclo for ... in?

El ciclo for...in es una forma de recorrer las propiedades enumerables de un objeto. Sin embargo, no es comúnmente utilizado para arrays porque recorre propiedades que pueden no estar relacionadas con índices numéricos, por lo que podría incluir métodos o propiedades heredadas.



```
let persona = {
  nombre: "Ana",
  edad: 25,
  profesion: "Ingeniera"
};

for (let clave in persona) {
  console.log(clave + ": " + persona[clave]);
}

// Salida:
// nombre: Ana
// edad: 25
// profesion: Ingeniera
```

En este ejemplo, for...in recorre todas las propiedades del objeto persona y accede tanto a las claves como a los valores.

for...in se sigue usando, pero con precaución, principalmente en objetos. Es útil cuando necesitas iterar sobre todas las propiedades de un objeto (incluyendo propiedades heredadas del prototipo, a menos que se use Object.hasOwn() o hasOwnProperty() para filtrar).

Sin embargo, para recorrer arrays, se recomienda usar for...of o métodos como forEach(), ya que for...in no es ideal para arrays (recorre índices como si fueran propiedades y puede incluir propiedades no numéricas).

2.4. Diferencias Entre Tipos Primitivos y Tipos Complejos

- Primitivos: Son inmutables, lo que significa que su valor no puede cambiar una vez asignado.
- Complejos: Son mutables, lo que significa que sus propiedades o elementos pueden cambiar después de ser asignados.

Estos tipos de datos son fundamentales en JavaScript y se utilizan para modelar la información que se manipula dentro de las aplicaciones.



2.5. Función typeof()

La función typeof es una herramienta en JavaScript que te permite conocer el tipo de dato de una variable o expresión. Esto es útil cuando estás depurando tu código o simplemente necesitas verificar qué tipo de valor está almacenando una variable. Vamos a ver en detalle cómo funciona y cuándo puede ser útil, con ejemplos y analogías sencillas.

2.5.1 ¿Qué es typeof?

typeof es un operador (no una función en el sentido técnico, aunque a menudo se le llama así) que devuelve el tipo de dato de la expresión o variable que le pasas. En otras palabras, te dice si lo que tienes es un número, una cadena de texto, un objeto, un booleano, etc.

Analogía
Imagina que tienes varias cajas en tu habitación: una contiene libros, otra contiene ropa, y otra contiene juguetes. Si etiquetas cada caja correctamente con “Libros”, “Ropa” o “Juguetes”, podrías saber rápidamente qué hay dentro sin tener que abrirla. Typeof es como esa etiqueta que te dice el tipo de contenido sin necesidad de “abrir” la variable.

2.5.2. ¿Cómo se usa typeof?

El uso de typeof es bastante simple. Solo tienes que colocarlo antes de la variable o expresión que quieres evaluar.

Sintaxis: typeof variable;

Ejemplos Básicos de typeof
Vamos a ver algunos ejemplos prácticos para ver cómo funciona:

- Con números

```
let edad = 25;
console.log(typeof edad); // Imprime: "number"
```

En este caso, edad es un número, por lo que typeof devuelve "number".



- Con cadenas de texto

```
let nombre = "Ana";
console.log(typeof nombre); // Imprime: "string"
```

Aquí, la variable nombre contiene una cadena de texto, y typeof nos dice que el tipo es "string".

- Con booleanos

```
let esMayorDeEdad = true;
console.log(typeof esMayorDeEdad); // Imprime: "boolean"
```

esMayorDeEdad es un valor booleano (true o false), por lo que typeof devuelve "boolean".

- Con objetos

```
let persona = {
  nombre: "Carlos",
  edad: 30
};
console.log(typeof persona); // Imprime: "object"
```

Un objeto siempre será identificado por typeof como "object", ya que es una colección de datos más compleja.

- Con null

```
let coche = null;
console.log(typeof coche); // Imprime: "object"
```

Nota importante: Aunque null representa la ausencia de valor, JavaScript lo identifica erróneamente como un "object". Esto es un comportamiento histórico en el lenguaje, y aunque es confuso, es algo que debes tener en cuenta.



- Con undefined

```
let fruta;
console.log(typeof fruta); // Imprime: "undefined"
```

Cuando una variable no ha sido asignada, typeof la clasificará como "undefined".

- Con funciones

```
function saludar() {
  console.log("Hola");
}
console.log(typeof saludar); // Imprime: "function"
```

JavaScript clasifica las funciones con el tipo "function", lo que te permite diferenciarlas de otros tipos de objetos.

2.5.3. Tipos de Datos que Devuelve typeof

Estos son los valores que typeof puede devolver:

Tipo	¿Qué significa?	Ejemplo
"undefined"	La variable no tiene valor asignado.	let x; typeof x === "undefined"
"number"	Un número, ya sea entero o decimal.	let edad = 25; typeof edad === "number"
"string"	Una cadena de texto (string).	let nombre = "Ana"; typeof nombre === "string"
"boolean"	Un valor true o false.	let esVerdad = true; typeof esVerdad === "boolean"
"object"	Un objeto o null.	let persona = {}; typeof persona === "object"
"function"	Una función.	let f = function() {}; typeof f === "function"
"symbol"	Un valor único creado con Symbol().	let sym = Symbol(); typeof sym === "symbol"



"bigint"	Números enteros de precisión arbitraria (grandes).	let bigNum = 123n; typeof bigNum === "bigint"

2.5.4. Casos Especiales y Peculiaridades

- El caso de null: Como mencionamos antes, aunque null representa la ausencia de valor, typeof lo categoriza erróneamente como un objeto ("object"). Esto es un comportamiento legado en JavaScript y uno de sus pocos "errores" históricos.
- Diferencia entre undefined y null:
  - undefined significa que algo no ha sido definido.
  - null es un valor que tú decides asignar intencionadamente para representar la ausencia de valor.
- Funciones se clasifican como "function", no como "object": Aunque las funciones son técnicamente objetos en JavaScript (¡todo lo es!), typeof las clasifica de forma separada como "function".

Ejemplo Práctico con typeof

Imagina que tienes un formulario donde el usuario puede ingresar datos, y antes de enviar la información, quieres verificar qué tipo de datos ha ingresado el usuario.

```
let nombre = "Carlos";
let edad = 28;
let esEstudiante = true;
let direccion = null;

console.log(typeof nombre);       // "string"
console.log(typeof edad);         // "number"
console.log(typeof esEstudiante); // "boolean"
console.log(typeof direccion);    // "object" (aunque esperas que sea null)
```

Aquí estamos verificando el tipo de cada variable para asegurarnos de que el usuario ha ingresado los tipos de datos correctos. Si, por ejemplo, edad resulta ser una cadena de texto en vez de un número, podrías manejar ese caso adecuadamente en tu código.



Resumen

- typeof es una herramienta fundamental para verificar el tipo de dato de una variable o expresión en JavaScript.
- Devuelve valores como "number", "string", "boolean", "object", "function", "undefined", y "symbol".
- Ten cuidado con null, ya que typeof lo clasificará como "object".
- Es útil para depurar y asegurarte de que las variables contienen los tipos de datos esperados.

Con la ayuda de typeof, puedes manejar los tipos de datos de manera más segura y evitar errores comunes en tu código.

2.6. NaN (Not-a-Number)

NaN es un valor especial en JavaScript que significa "Not-a-Number" o "no es un número". Este valor aparece cuando intentas realizar una operación matemática que no tiene sentido o no es válida.

Analogía:
Imagina que intentas hacer una operación matemática como "multiplicar una idea". Como es imposible realizar esa operación con palabras en vez de números, obtienes como resultado algo que no tiene sentido: NaN.

Ejemplos:

```
let resultado = 5 * "hola";
console.log(resultado); // Imprime: NaN
```

```
let numero = "cinco" - 2;
console.log(numero); // Imprime: NaN
```

- Explicación:
  - En el primer ejemplo, intentamos multiplicar el número 5 por la cadena de texto "hola". Como no puedes realizar operaciones matemáticas con palabras, JavaScript devuelve NaN.
  - En el segundo ejemplo, intentamos restar el número 2 de la cadena "cinco". Nuevamente, esto no es válido, por lo que obtenemos NaN.



Cuando se usa

- Aparece cuando una operación matemática no es válida o no tiene sentido.
- Puede ser el resultado de intentar convertir algo que no es numérico en un número.

Comparación undefined, NaN y null

Tipo	¿Qué es?	Ejemplo	Cuándo se usa
undefined	El valor predeterminado de una variable que ha sido declarada pero no tiene valor asignado. También cuando se accede a una propiedad que no existe.	let nombre;	Se usa cuando algo aún no ha sido definido o no tiene un valor asignado.
NaN	Significa "Not-a-Number", y se usa cuando una operación matemática no es válida o no tiene sentido.	5 * "hola"	Aparece en operaciones matemáticas no válidas o cuando un valor no numérico se intenta usar como número.
null	Un valor intencionado que indica la ausencia de valor.	let coche = null;	Se usa cuando quieres establecer explícitamente que una variable o propiedad no tiene un valor, pero no es un error o indefinido.

```
// Ejemplo de undefined
let fruta;
console.log(fruta); // Imprime: undefined (porque no hemos asignado ningún valor

// Ejemplo de NaN
let resultado = 10 / "manzana";
console.log(resultado); // Imprime: NaN (porque no puedes dividir un número por u

// Ejemplo de null
let coche = null;
console.log(coche); // Imprime: null (porque hemos asignado explícitamente "null"
```



Resumen

- undefined: Cuando algo no está definido o no tiene valor asignado.
- NaN: Aparece cuando una operación matemática no es válida.
- null: Es un valor intencional que indica la ausencia de un valor.

3. Operadores

3.1. Operadores Matemáticos

Los operadores matemáticos se utilizan para realizar cálculos con números. Estos operadores te permiten sumar, restar, multiplicar, dividir y realizar otras operaciones matemáticas básicas en tus programas.

- Suma (+)
  Realiza la suma de dos valores.

```
let a = 5;
let b = 3;
let resultado = a + b; // resultado es 8
```

- Resta (-)
  Realiza la resta de dos valores.

```
let a = 10;
let b = 6;
let resultado = a - b; // resultado es 4
```

- Multiplicación (*)
  Realiza la multiplicación de dos valores.

```
let a = 4;
let b = 2;
let resultado = a * b; // resultado es 8
```



- División (/)
  Realiza la división de dos valores.

```
let a = 12;
let b = 4;
let resultado = a / b; // resultado es 3
```

- Módulo (%)
  Calcula el resto de la división de dos valores.

```
let a = 10;
let b = 3;
let resultado = a % b; // resultado es 1
```

- Incremento (++)
  Aumenta el valor de una variable en 1 (variable++ o ++variable)

```
let a = 5;
a++; // a ahora es 6
```

- Decremento (--)
  Disminuye el valor de una variable en 1 (variable-- o --variable)

```
let a = 5;
a--; // a ahora es 4
```

- Exponenciación (**)
  Eleva un número a la potencia de otro número.

```
let a = 2;
let b = 3;
let resultado = a ** b; // resultado es 8
```



Operador	Descripción	Ejemplo	Resultado
+	Suma de dos valores	5 + 3	8
-	Resta de dos valores	10 - 6	4
*	Multiplicación de dos valores	4 * 2	8
/	División de dos valores	12 / 4	3
%	Módulo (resto de una división)	10 % 3	1
++	Incremento (aumenta en 1 el valor de la variable)	let x = 5; x++	6
--	Decremento (disminuye en 1 el valor de la variable)	let x = 5; x--	4
**	Exponenciación (potencia)	2 ** 3	8

```
let suma = 5 + 3;      // 8
let resta = 10 - 6;    // 4
let multi = 4 * 2;     // 8
let div = 12 / 4;      // 3
let mod = 10 % 3;      // 1
let potencia = 2 ** 3; // 8
```

3.2. Operadores de Asignación

Los operadores de asignación se utilizan para asignar valores a las variables y, en algunos casos, para combinar asignación con operaciones matemáticas.
Estos operadores permiten realizar cálculos y actualizar variables de manera concisa.

- Asignación Básica (=)
  Asigna el valor de la derecha a la variable de la izquierda.

```
let a = 5; // a es igual a 5
```

- Asignación con Suma (+=)
  Suma el valor a la variable y asigna el resultado a la misma variable.

```
let a = 5;
a += 3; // a es ahora 8 (5 + 3)
```



- Asignación con Resta (-=)
  Resta el valor de la variable y asigna el resultado a la misma variable.

```
let a = 10;
a -= 4; // a es ahora 6 (10 - 4)
```

- Asignación con Multiplicación (*=)
  Multiplica la variable por el valor y asigna el resultado a la misma variable.

```
let a = 4;
a *= 3; // a es ahora 12 (4 * 3)
```

- Asignación con División (/=)
  Divide la variable por el valor y asigna el resultado a la misma variable.

```
let a = 12;
a /= 4; // a es ahora 3 (12 / 4)
```

- Asignación con Módulo (%=)
  Calcula el módulo de la variable y el valor y asigna el resultado a la misma variable.

```
let a = 10;
a %= 3; // a es ahora 1 (10 % 3)
```

Operador	Descripción	Ejemplo	Resultado
=	Asigna el valor de la derecha a la variable de la izquierda	let x = 5	x es 5
+=	Suma el valor a la variable y asigna el resultado	x += 2 (si x = 5)	x es 7
-=	Resta el valor de la variable y asigna el resultado	x -= 3 (si x = 5)	x es 2
*=	Multiplica la variable por el valor y asigna el resultado	x *= 3 (si x = 4)	x es 12



/=	Divide la variable por el valor y asigna el resultado	x /= 2 (si x = 10)	x es 5
%=	Calcula el módulo (resto) y asigna el resultado	x %= 3 (si x = 10)	x es 1

```
let x = 10;
x += 5; // x es ahora 15
x -= 3; // x es ahora 12
x *= 2; // x es ahora 24
x /= 4; // x es ahora 6
x %= 5; // x es ahora 1
```

3.3. Operadores de Comparación

Los operadores de comparación se utilizan para comparar dos valores y determinar su relación. Estos operadores devuelven un valor booleano (true o false) dependiendo del resultado de la comparación.

- Igualdad (==)
  Compara dos valores para ver si son iguales, realizando conversión de tipos si es necesario.

```
let a = 5;
let b = '5';
console.log(a == b); // true, porque el valor es igual después de la conversión de tip
```

- Igualdad Estricta (===)
  Compara dos valores para ver si son iguales y del mismo tipo.

```
let a = 5;
let b = '5';
console.log(a === b); // false, porque los tipos son diferentes
```



- Desigualdad (!=)
  Compara dos valores para ver si son diferentes, realizando conversión de tipos si es necesario.

```
let a = 5;
let b = 6;
console.log(a != b); // true, porque 5 no es igual a 6
```

- Desigualdad Estricta (!==)
  Compara dos valores para ver si son diferentes y de tipo diferente.

```
let a = 5;
let b = '5';
console.log(a !== b); // true, porque los tipos son diferentes
```

- Mayor que (>)
  Compara dos valores para ver si el primero es mayor que el segundo.

```
let a = 7;
let b = 5;
console.log(a > b); // true, porque 7 es mayor que 5
```

- Menor que (<)
  Compara dos valores para ver si el primero es menor que el segundo.

```
let a = 3;
let b = 5;
console.log(a < b); // true, porque 3 es menor que 5
```

- Mayor o Igual que (>=)
  Compara dos valores para ver si el primero es mayor o igual al segundo.

```
let a = 5;
let b = 5;
console.log(a >= b); // true, porque 5 es igual a 5
```



- Menor o Igual que (<=)
  Compara dos valores para ver si el primero es menor o igual al segundo.

```
let a = 4;
let b = 5;
console.log(a <= b); // true, porque 4 es menor que 5
```

Operador	Descripción	Ejemplo	Resultado
==	Igualdad (realiza conversión de tipo)	5 == '5'	true
===	Igualdad estricta (no realiza conversión de tipo)	5 === '5'	false
!=	Desigualdad (realiza conversión de tipo)	5 != 6	true
!==	Desigualdad estricta (no realiza conversión de tipo)	5 !== '5'	true
>	Mayor que	7 > 5	true
<	Menor que	3 < 5	true
>=	Mayor o igual que	5 >= 5	true
<=	Menor o igual que	4 <= 5	true

3.4. Operadores Lógicos

Los operadores lógicos son fundamentales para tomar decisiones más complejas dentro de las estructuras condicionales. Te permiten combinar múltiples condiciones y controlar el flujo de tu programa de manera más precisa.

Los tres operadores lógicos principales son:

- AND (&&) – Devuelve true si ambas condiciones son verdaderas.
- OR (||) – Devuelve true si al menos una de las condiciones es verdadera.
- NOT (!) – Invierte el valor lógico, es decir, devuelve true si la condición es falsa y false si la condición es verdadera.

3.4.1. Operador AND (&&)

El operador AND requiere que ambas condiciones sean verdaderas para que la expresión general sea verdadera.



```
let edad = 25;
let tieneLicencia = true;

if (edad >= 18 && tieneLicencia) {
    console.log("Puede conducir.");
} else {
    console.log("No puede conducir.");
}
// Output: Puede conducir.
```

- Explicación
  Aquí tenemos dos condiciones:
  - La edad es mayor o igual a 18.
  - La persona tiene una licencia (tieneLicencia es true).

  Ambas deben ser verdaderas para que la persona pueda conducir. Si cualquiera de ellas es falsa, no podrá conducir.

3.4.2. Operador OR (||)

El operador OR devuelve true si al menos una de las condiciones es verdadera.

```
let esMiembro = false;
let tieneEntrada = true;

if (esMiembro || tieneEntrada) {
    console.log("Puede asistir al evento.");
} else {
    console.log("No puede asistir.");
}
// Output: Puede asistir al evento.
```

- Explicación
  Aquí solo una de las condiciones necesita ser verdadera:
  - La persona es miembro (esMiembro).
  - La persona tiene una entrada (tieneEntrada).



Como tiene una entrada, aunque no sea miembro, puede asistir al evento.

3.4.3. Operador NOT (!)

El operador NOT invierte el valor lógico de una condición. Si la condición es true, ! la convierte en false, y viceversa.

```
let estaLloviendo = false;

if (!estaLloviendo) {
    console.log("Puedes salir sin paraguas.");
} else {
    console.log("Lleva paraguas.");
}
// Output: Puedes salir sin paraguas.
```

- Explicación
  El operador ! invierte el valor de estaLloviendo. Como estaLloviendo es false, !estaLloviendo se convierte en true, por lo que el programa imprime "Puedes salir sin paraguas."

3.5. String Operators

Los operadores de cadenas en JavaScript permiten manipular textos y combinarlos de diversas maneras. Estos operadores son útiles para construir mensajes, procesar datos de entrada del usuario y más.

- Concatenación (+)
  Une dos cadenas de texto en una sola.

```
let saludo = "Hola";
let nombre = "Mundo";
let mensaje = saludo + " " + nombre; // "Hola Mundo"
```



- Concatenación y Asignación (+=)
  Une una cadena adicional a la cadena existente y asigna el resultado.

```
let saludo = "Hola";
saludo += " Mundo"; // "Hola Mundo"
```

4. Condicionales en JavaScript

Los condicionales permiten que tu programa tome decisiones basadas en condiciones.

4.1. Condicional if y sus variaciones

Las estructuras condicionales son fundamentales en la programación, ya que permiten que el código tome decisiones y ejecute diferentes bloques de instrucciones basándose en condiciones específicas. En JavaScript, la estructura condicional más básica es la sentencia if, que puede extenderse con else y else if para manejar múltiples escenarios. A continuación, exploraremos en detalle cómo funcionan estas estructuras y cómo utilizarlas eficazmente en tus programas.

Sentencia if

La sentencia if ejecuta un bloque de código si una condición especificada es verdadera. Si la condición es falsa, el bloque de código se omite.

```
let edad = 20;
if (edad >= 18) {
  console.log("Eres mayor de edad");
}
```

- Explicación: Si edad es mayor o igual a 18, se imprime "Eres mayor de edad". Si no, no hace nada.

Sentencia else

El bloque else es una extensión del if que se ejecuta si la condición en el if es falsa. Es decir, si no se cumple la condición del if, el código dentro del else se ejecuta.



```
let edad = 16;
if (edad >= 18) {
  console.log("Eres mayor de edad");
} else {
  console.log("Eres menor de edad");
}
```

- Explicación: Si edad es menor a 18, el programa toma otra acción, en este caso, imprime "Eres menor de edad".

Sentencia else if

La sentencia else if permite evaluar múltiples condiciones. Si la primera condición en el if es falsa, el programa evalúa las condiciones adicionales en los bloques else if. Si ninguna condición es verdadera, se ejecuta el bloque else (opcional).

```
let temperatura = 30;
if (temperatura > 30) {
  console.log("Hace mucho calor");
} else if (temperatura < 15) {
  console.log("Hace frío");
} else {
  console.log("El clima es agradable");
}
```

- Explicación: Aquí se comprueban múltiples condiciones y se ejecuta el bloque de código correspondiente al caso que se cumpla.

Buenas Prácticas al Usar Condicionales

- Claridad en las Condiciones: Asegúrate de que las condiciones sean claras y fáciles de entender. Evita condiciones demasiado complejas que puedan dificultar la lectura del código.
- Orden de las Condiciones: Coloca las condiciones más específicas o más probables al inicio de la cadena de if y else if para optimizar el flujo de ejecución.



- Evita la Redundancia: No repitas condiciones que ya han sido evaluadas. Cada condición en una cadena de if-else if-else debe ser única y no redundante con las anteriores.
- Uso Adecuado de else: Utiliza else para manejar casos por defecto, pero evita depender exclusivamente de él para todas las posibles condiciones. Asegúrate de que todas las posibles ramas sean manejadas adecuadamente.
- Indentación y Formato: Mantén una buena indentación y formato en tu código para mejorar su legibilidad.

Las estructuras condicionales if, else if y else son fundamentales para controlar el flujo de ejecución en tus programas de JavaScript. Permiten que tu código responda de manera dinámica a diferentes condiciones y escenarios, haciendo que tus aplicaciones sean más interactivas y adaptativas. Al dominar estas estructuras, podrás crear lógica más compleja y manejar una variedad de situaciones de manera eficiente.

Uso de Paréntesis para Agrupar Condiciones

Al combinar múltiples operadores lógicos, es recomendable usar paréntesis para asegurar que las condiciones se evalúen en el orden deseado.

```
let hora = 14;
let diaSemana = "Martes";

if ((hora >= 9 && hora <= 17) && (diaSemana !== "Domingo")) {
    console.log("Oficina abierta.");
} else {
    console.log("Oficina cerrada.");
}
// Output: Oficina abierta.
```

4.2. Operador Ternario

El operador ternario es una forma compacta de escribir una sentencia if...else. Se llama ternario porque utiliza tres partes: una condición, una expresión para el valor true y una expresión para el valor false.



Sintaxis: condición ¿ expresiónSiVerdadera : expresiónSiFalsa;

Ejemplo:

```
let edad = 20;

let mensaje = (edad >= 18) ? "Eres mayor de edad." : "Eres menor de edad.";
console.log(mensaje);
// Output: Eres mayor de edad.
```

- Explicación
  Aquí, si edad >= 18 es true, la variable mensaje tomará el valor "Eres mayor de edad". Si es false, tomará el valor "Eres menor de edad."

Ventajas

- Es una forma concisa y limpia de realizar decisiones rápidas.
- Ideal para condiciones simples.

Desventajas

- No es recomendable para condiciones complejas, ya que puede afectar la legibilidad.

4.3. switch

La estructura switch es una alternativa a las sentencias if...else if...else cuando se tienen múltiples posibles valores para una variable o expresión. A menudo es más clara y eficiente en estos casos.



Sintaxis:

```
switch (expresión) {
    case valor1:
        // Código a ejecutar si expresión === valor1
        break;
    case valor2:
        // Código a ejecutar si expresión === valor2
        break;
    default:
        // Código a ejecutar si ninguna condición es verdadera
}
```

Ejemplo:

```
let diaSemana = "Martes";

switch (diaSemana) {
    case "Lunes":
        console.log("Inicio de la semana.");
        break;
    case "Martes":
        console.log("Es martes, día de reunión.");
        break;
    case "Viernes":
        console.log("Último día de la semana laboral.");
        break;
    default:
        console.log("Es un día laboral normal.");
}
// Output: Es martes, día de reunión.
```



- Explicación
  El programa evalúa la variable diaSemana. Como el valor es "Martes", se ejecuta el bloque de código correspondiente a ese caso y luego se detiene el switch con break.

Ventajas de switch
- Ideal para manejar múltiples valores posibles de una variable.
- Mejora la legibilidad cuando se comparan muchos valores.

Desventajas de switch
- No es flexible para comparaciones complejas (por ejemplo, rangos o múltiples condiciones).

5. Ciclos en JavaScript

Los ciclos (o bucles) son herramientas que nos permiten repetir un conjunto de instrucciones varias veces sin tener que escribir el mismo código una y otra vez. Veamos cada uno en detalle.

5.1. Ciclo for

El ciclo for es ideal cuando sabes de antemano cuántas veces necesitas repetir el bloque de código. Su estructura es muy específica y clara:

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

Estructura:
- Inicialización (let i = 0): Aquí defines la variable de control (comúnmente llamada i), que es el contador del ciclo. Por lo general, se inicializa en 0.
- Condición (i < 5): Mientras esta condición sea true, el ciclo seguirá ejecutándose.
- Actualización (i++): Después de cada ejecución del ciclo, se incrementa el valor de i.



Ventajas:

- Ideal cuando se conoce de antemano cuántas veces quieres que el ciclo se ejecute.
- Todo el control del ciclo (inicio, condición, incremento) está en una sola línea, lo que lo hace fácil de leer y entender.

Desventajas:

- No es muy flexible si las condiciones cambian a lo largo del ciclo.

Cuando usarlo: Úsalo cuando necesitas recorrer un número conocido de veces, por ejemplo, para recorrer un array o ejecutar una acción repetida un número específico de veces.

Analogía: Imagina que tienes que contar de 1 a 10 con los dedos de la mano. Sabes que hay un número fijo de dedos (10), así que no necesitas ninguna condición especial: simplemente cuentas del 1 al 10.

5.2. Ciclo while

El ciclo while es más flexible que el for y se utiliza cuando no sabes cuántas veces exactamente se repetirá el ciclo. Se ejecutará mientras la condición sea verdadera.

```
let contador = 0;
while (contador < 5) {
  console.log(contador);
  contador++;
}
```

Estructura:

- Condición: El ciclo continuará ejecutándose mientras esta condición sea verdadera.
- Actualización dentro del ciclo: Debes actualizar manualmente el valor del contador dentro del bloque del ciclo, para evitar un ciclo infinito.

Ventajas:

- Más flexible que el ciclo for, porque no necesitas saber cuántas veces se ejecutará el ciclo antes de comenzar.



Desventajas:

- Si te olvidas de actualizar el contador dentro del ciclo, puedes entrar en un ciclo infinito que nunca termina.
- Menos claro al leerlo que un for, ya que la actualización del contador está separada de la condición.

Cuando usarlo: Úsalo cuando no sabes cuántas veces el ciclo deberá ejecutarse. Por ejemplo, mientras un usuario no ingrese la opción correcta en un formulario, o mientras se espera una acción externa.

Analogía: Es como estar esperando en una fila para el autobús. Sabes que te quedarás en la fila **mientras** el autobús no llegue. No sabes cuántas veces tendrás que mirar si llega el bus, pero sigues esperando hasta que llegue.

5.3. Ciclo do while

El ciclo do while es similar al while, pero la diferencia clave es que garantiza que el código dentro del ciclo se ejecute **al menos una vez**, sin importar si la condición es verdadera o falsa.

```
let contador = 5;
do {
  console.log(contador);
  contador++;
} while (contador < 5);
```

Estructura:

- **Código dentro del bloque do**: Se ejecuta **una vez** antes de verificar la condición.
- **Condición**: Luego de la primera ejecución, la condición se verifica. Si es true, el ciclo se repite. Si es false, el ciclo termina.

Ventajas:

- Garantiza al menos una ejecución del bloque de código.

Desventajas:

- Puede ejecutar el bloque de código una vez más de lo necesario si no tienes cuidado al estructurar la condición.



Cuando usarlo: Úsalo cuando necesitas que el código se ejecute al menos una vez, independientemente de la condición. Por ejemplo, cuando quieres asegurarte de que un mensaje de error se muestre al menos una vez antes de validar la entrada del usuario.

Analogía: Es como intentar abrir una puerta. Intentas abrir la puerta una vez, incluso si piensas que está cerrada. Solo después de intentar abrirla te das cuenta de si está cerrada o no, y decides si vuelves a intentarlo.

Comparación de los ciclos

Ciclo	Ventajas	Desventajas	Cuándo usarlo
for	Estructura clara y concisa; ideal para cuando conoces el número exacto de iteraciones.	Menos flexible si las condiciones cambian dentro del ciclo.	Cuando sabes cuántas veces quieres ejecutar el ciclo (ej., recorrer arrays o repetir una tarea).
while	Más flexible, útil cuando no sabes cuántas veces necesitarás ejecutar el ciclo.	El riesgo de ciclos infinitos si no actualizas el contador.	Cuando no sabes de antemano cuántas veces se ejecutará el ciclo (ej., esperar entrada del usuario).
do while	Asegura que el bloque de código se ejecute al menos una vez.	Puede ejecutar el bloque innecesariamente una vez más si la condición es mal manejada.	Cuando necesitas que el código se ejecute al menos una vez antes de validar la condición.

5.4. Variable i como iterador en los ciclos

La variable i es la más comúnmente utilizada como iterador en los ciclos debido a una convención histórica. Aquí te explico por qué:

- Simplicidad y claridad: i viene de la palabra "index" en inglés, que significa índice. Usar i es una forma corta y reconocida de indicar que estamos trabajando con índices en una lista o array.
- Convención universal: Aunque podrías usar cualquier nombre de variable (como contador, num, etc.), i es común en todo el mundo. Otros programadores sabrán al instante que i se refiere al índice de un ciclo, lo que facilita la lectura del código.



6. Funciones en JavaScript

Las funciones son bloques de código reutilizables. Piensa en ellas como recetas de cocina: una vez que tienes una receta, puedes seguirla una y otra vez para obtener el mismo resultado.

Sintaxis Básica de una Función

```
function saludar() {
  console.log("¡Hola Mundo!");
}
```

- Explicación:
  - function saludar() {}: Declara una función llamada saludar.
  - console.log("¡Hola Mundo!");: Es el contenido de la función. Cada vez que se llame a saludar, imprimirá "¡Hola Mundo!" en la consola.

Llamar a una función: Para usar una función, simplemente escribes su nombre seguido de paréntesis

```
saludar(); // Imprime "¡Hola Mundo!" en la consola
```

6.1 Scope (Alcance)

El scope determina dónde una variable es accesible en el código. Hay dos tipos principales de scope:

- Local: Variables declaradas dentro de una función solo existen dentro de esa función.
- Global: Variables declaradas fuera de cualquier función son accesibles en todo el código.

Ejemplo:



```
let global = "Soy global";

function mostrarVariable() {
  let local = "Soy local";
  console.log(global);  // Funciona, porque "global" es accesible en todo el código.
  console.log(local);   // Funciona, porque estamos dentro de la función.
}

mostrarVariable();
console.log(local);  // Error, porque "local" solo existe dentro de la función.
```

6.2 Diferencias entre Variable, Parámetro y Argumento

Es fácil confundir los términos **variable, parámetro, y argumento**, ya que todos tienen que ver con el manejo de datos en la programación. Vamos a desglosarlos en detalle, usando analogías y ejemplos prácticos para entender mejor cómo se diferencian y cómo se usan.

En programación, cuando trabajamos con funciones, a menudo necesitamos **pasar información** a esas funciones para que hagan su trabajo. Esa información se maneja a través de **parámetros** y **argumentos**. Estos dos conceptos están muy relacionados, pero tienen diferencias importantes. Vamos a desglosarlos paso a paso con ejemplos y una explicación clara para que los entiendas a fondo.

1. Variables

Una **variable** es un contenedor que guarda un valor. Piensa en una variable como una caja con una etiqueta, donde puedes guardar algo para usarlo más tarde. Las variables permiten almacenar datos y luego manipularlos a lo largo de tu programa.

Analogía:
Imagina que tienes una caja con una etiqueta que dice "edad". En esta caja puedes guardar el número de años que tienes. Si tu edad cambia, puedes actualizar el valor en la caja, pero la etiqueta ("edad") sigue siendo la misma.

Ejemplo en JavaScript:



```
let edad = 25; // Declaramos una variable llamada "edad" y le asignamos el valor 25.
edad = 26;     // Actualizamos el valor de "edad" a 26.
console.log(edad); // Imprimirá: 26
```

- Explicación:
- En este ejemplo, edad es una variable. Al principio tiene el valor 25, pero más tarde cambiamos su valor a 26.
- Las variables son fundamentales porque permiten almacenar datos y reutilizarlos en diferentes partes de tu código.

2. Parámetros

Los **parámetros** son "cajas vacías" o nombres que se usan en la **declaración** de una función, y que esperan recibir un valor cuando se llame a la función. Es como un lugar reservado que la función necesita para poder hacer su trabajo correctamente.

Analogía:
Imagina que estás planeando una comida, y necesitas ingredientes. En tu receta, pones "ingrediente1" y "ingrediente2" como espacios reservados, sin saber todavía qué alimentos vas a usar. Los nombres de estos ingredientes en tu receta son los **parámetros**.

Ejemplo en JavaScript:

```
function cocinar(ingrediente1, ingrediente2) {
  console.log("Hoy vamos a cocinar con " + ingrediente1 + " y " + ingrediente2);
}
```

- Explicación:
  En este ejemplo, ingrediente1 y ingrediente2 son **parámetros** de la función cocinar. Estos son espacios reservados que la función necesita para hacer su trabajo. La función aún no sabe qué valores tendrán estos parámetros.

3. Argumentos

Los **argumentos** son los **valores reales** que pasas a la función cuando la llamas. Estos valores llenan los parámetros definidos en la función, permitiendo que la función trabaje con datos específicos.



Analogía:
Siguiendo con la analogía de la receta, cuando decides que "ingrediente1" será "pasta" y "ingrediente2" será "tomate", estás pasando los **argumentos** a tu receta. Estos valores específicos llenan los espacios reservados que tenías en la receta (los parámetros).

Ejemplo en JavaScript:

```
cocinar("pasta", "tomate");
```

- Explicación:
  - Cuando llamas a la función cocinar("pasta", "tomate"), estás proporcionando los **argumentos** "pasta" y "tomate" para los parámetros ingrediente1 e ingrediente2.
  - La función ahora puede usar estos valores para realizar su trabajo.

Comparación Detallada

Concepto	¿Qué es?	Ejemplo	Función
Variable	Un contenedor que almacena datos y que puede ser actualizado o reutilizado.	let edad = 25;	Guarda un valor para usarlo en varias partes del programa.
Parámetro	Un espacio reservado en una función que espera recibir un valor cuando la función sea llamada.	function sumar(a, b) {}	Los nombres a y b son los parámetros que se definirán cuando se llame a la función.
Argumento	Los valores reales que se pasan a la función cuando es llamada para llenar los parámetros.	sumar(5, 3);	Los valores 5 y 3 son los argumentos que se pasan a la función para llenar los parámetros a y b.



Ejemplo Combinado para Aclarar las Diferencias

```
let numero1 = 10; // Variable que guarda el valor 10
let numero2 = 20; // Variable que guarda el valor 20

function sumar(a, b) {  // a y b son parámetros
  return a + b;         // Suma los dos parámetros
}

console.log(sumar(numero1, numero2));  // numero1 y numero2 son los argumentos
```

- Explicación:
  - numero1 y numero2 son **variables** que almacenan los valores 10 y 20.
  - En la declaración de la función sumar, a y b son **parámetros**.
  - Cuando llamamos a la función con sumar(numero1, numero2), estamos pasando los valores de las variables numero1 y numero2 como **argumentos** para los parámetros a y b.

Resumen

- **Variables**: Son contenedores que almacenan datos para ser utilizados en el programa.
- **Parámetros**: Son nombres que actúan como espacios reservados dentro de una función. Esperan recibir valores cuando la función sea llamada.
- **Argumentos**: Son los valores reales que se pasan a los parámetros de una función cuando la llamamos.

Características Importantes de Parámetros y Argumentos

- **Parámetros opcionales**: En JavaScript, si no pasas argumentos suficientes cuando llamas a una función, los parámetros faltantes tendrán un valor de undefined por defecto. Puedes usar esto para hacer que algunos parámetros sean opcionales.



```
function saludar(nombre, saludo = "Hola") {
  console.log(saludo + " " + nombre);
}

saludar("Ana"); // Resultado: Hola Ana
saludar("Pedro", "Buenos días"); // Resultado: Buenos días Pedro
```

- Explicación:
  En este caso, si no proporcionas el argumento saludo, la función usará el valor por defecto "Hola". Así, saludo es un parámetro opcional.

- Número de argumentos y parámetros: Puedes pasar más argumentos que los parámetros definidos, pero esos argumentos adicionales serán ignorados. Si pasas menos argumentos, los parámetros faltantes serán undefined.

```
function multiplicar(a, b) {
  console.log(a * b);
}

multiplicar(5, 3, 10); // Solo se usa 5 y 3, resultado: 15
multiplicar(7); // b será undefined, resultado: NaN
```

- Explicación:
  - En el primer caso, pasamos tres argumentos, pero solo se usan los dos primeros.
  - En el segundo caso, solo pasamos un argumento, por lo que el parámetro b es undefined, lo que resulta en un cálculo inválido (NaN).

6.3. Elemento this

El contexto de this en JavaScript depende de cómo y dónde se invoca la función. this no siempre hace referencia al objeto que contiene el método; su valor depende de cómo se ejecuta la función.



¿Qué es this?

this es una referencia que apunta al contexto actual donde una función está siendo ejecutada. En términos simples, depende de quién invoca la función, lo que significa que su valor puede cambiar según el modo de llamada.

¿Cómo funciona this en diferentes contextos?

6.3.1. Dentro de un objeto

Cuando utilizas this en un método dentro de un objeto, this hace referencia al objeto en el que el método fue llamado.

```
let persona = {
  nombre: "Juan",
  edad: 30,
  saludar: function() {
    console.log(this.nombre); // "this" se refiere a "persona"
  }
};

persona.saludar(); // "Juan"
```

- En este caso, this.nombre es equivalente a persona.nombre porque el método saludar() es invocado a través del objeto persona.

6.3.2. En el contexto global (fuera de un objeto)

Cuando se usa fuera de un objeto, this hace referencia al objeto global (en navegadores, esto es window).

```
function mostrar() {
  console.log(this); // En el navegador, "this" se refiere al objeto global "window"
}

mostrar(); // El valor de "this" es "window" o "global" en Node.js
```



- Cuando llamas a mostrar() sin un contexto específico, this apunta al objeto global.

6.3.3. En un constructor con new

Cuando se usa new, this se refiere a la nueva instancia del objeto que estás creando.

```
function Persona(nombre) {
  this.nombre = nombre;
}

let persona1 = new Persona("Ana");
console.log(persona1.nombre); // "Ana"
```

- Aquí, this en el constructor Persona se refiere a la nueva instancia persona1.

6.3.4. En funciones flecha (arrow functions)

Las funciones flecha no tienen su propio this; heredan el this del contexto en el que fueron creadas.

```
let persona = {
  nombre: "Luis",
  saludar: function() {
    let fn = () => {
      console.log(this.nombre); // "this" es heredado del contexto de "saludar"
    };
    fn();
  }
};

persona.saludar(); // "Luis"
```

- En este caso, this dentro de la función flecha hace referencia al objeto persona, porque hereda el contexto del método saludar.



Cuando usar this y sus advertencias

Es importante tener cuidado con el uso de this en funciones tradicionales, especialmente cuando se pasa como callback, ya que puede cambiar inesperadamente su contexto. Para evitar esto, puedes usar funciones flecha, o bind, call, y apply para controlar manualmente el valor de this.

```
let persona = {
  nombre: "Carlos",
  saludar: function() {
    setTimeout(function() {
      console.log(this.nombre); // Aquí "this" es el objeto global, no "persona"
    }, 1000);
  }
};

persona.saludar(); // undefined
```

Para corregir esto, una función flecha:

```
setTimeout(() => {
  console.log(this.nombre); // Ahora "this" se refiere al objeto "persona"
}, 1000);
```

6.4 Arrow Functions (Funciones Flecha)

Las arrow functions (funciones flecha) son una forma más concisa de escribir funciones en JavaScript, introducidas en ECMAScript 6 (ES6). Son útiles porque hacen el código más limpio y manejan el contexto de this de manera diferente a las funciones tradicionales.



Sintaxis de Arrow Functions

```
(param1, param2, ..., paramN) => {
  // Cuerpo de la función
  return result;
}
```

Si la función solo tiene una línea en el cuerpo, se puede simplificar aún más omitiendo las llaves y el return:

```
(param1, param2) => result;
```

Ejemplo básico de una arrow function:

```
const sumar = (a, b) => a + b;
console.log(sumar(5, 3)); // Resultado: 8
```

- Explicación:
  - sumar es una función flecha que toma dos parámetros (a y b).
  - Retorna la suma de a y b directamente.
  - No es necesario usar return ni llaves {} cuando es una sola línea.

Arrow Functions vs Funciones Tradicionales

Comparémoslas con una función tradicional:

```
// Función tradicional
function sumar(a, b) {
  return a + b;
}

// Arrow function equivalente
const sumar = (a, b) => a + b;
```



- Ventajas:
  - Las arrow functions son más concisas.
  - Mantienen el contexto de this (esto es importante cuando trabajamos con objetos o clases).

Ejemplo con contexto de this:

```
function Persona() {
  this.nombre = "María";
  setTimeout(function () {
    console.log(this.nombre); // undefined, porque `this` no apunta al objeto `Persona`
  }, 1000);
}

const persona = new Persona();
```

Aquí, this dentro de la función regular dentro de setTimeout no se refiere al objeto Persona. Para solucionarlo, podemos usar una arrow function que hereda el valor de this del contexto exterior.

```
function Persona() {
  this.nombre = "María";
  setTimeout(() => {
    console.log(this.nombre); // "María", porque `this` apunta al objeto `Persona`
  }, 1000);
}

const persona = new Persona();
```

6.5 Funciones Anónimas

Las funciones anónimas en JavaScript, como su nombre lo indica, son funciones que no tienen nombre. Son comunes en situaciones donde necesitas una función para un uso temporal, como cuando trabajas con eventos o funciones de orden superior (funciones que aceptan otras funciones como argumentos).

Vamos a sumergirnos en el tema con una explicación paso a paso, acompañada de ejemplos y analogías, para que quede muy claro.



¿Qué es una función anónima?

En términos simples, una función anónima es una función sin nombre que se declara y se puede usar en el mismo lugar donde se necesita. Puedes pensar en ellas como "funciones desechables", ya que se usan una vez y luego no vuelven a ser necesarias.

Ejemplo de una función anónima:

```
(function() {
  console.log("Soy una función anónima");
})();
```

- Explicación:
  - Aquí hemos declarado y ejecutado una función sin nombre.
  - Este tipo de función anónima se ejecuta inmediatamente después de ser creada. Esto es útil cuando quieres ejecutar código de forma instantánea sin necesidad de reutilizar esa función.
  -

Sintaxis de funciones anónimas

Las funciones anónimas se escriben igual que las funciones regulares, pero sin darle un nombre.

```
function () {
  // Código de la función
}
```

Aunque esta función, por sí sola, no tiene sentido (porque no podemos invocarla sin un nombre), normalmente se asigna a una variable o se pasa como argumento a otra función. Veremos más detalles sobre esto más adelante.

¿Dónde y por qué se usan las funciones anónimas?

1. Asignación a variables

Puedes asignar una función anónima a una variable. De esta manera, puedes llamarla usando el nombre de la variable.



```
const saludar = function() {
  console.log("¡Hola!");
};

saludar(); // Resultado: ¡Hola!
```

- Explicación:
  - Hemos asignado una función sin nombre a la variable saludar.
  - Luego llamamos a la función usando el nombre de la variable saludar().

2. Funciones como argumentos

Una de las principales razones por las que se usan funciones anónimas es para pasarlas como argumentos a otras funciones. Esto es muy común en JavaScript, especialmente cuando se trabaja con métodos como setTimeout, map, filter, etc.

Ejemplo con setTimeout:

```
setTimeout(function() {
  console.log("Esto se ejecuta después de 2 segundos");
}, 2000);
```

- Explicación:
  - Aquí estamos pasando una función anónima como argumento a setTimeout, que se ejecutará después de 2000 milisegundos (2 segundos).
  - No necesitamos darle un nombre a la función porque solo se ejecutará una vez.

Ejemplo con map:

```
const numeros = [1, 2, 3, 4, 5];
const dobles = numeros.map(function(numero) {
  return numero * 2;
});

console.log(dobles); // Resultado: [2, 4, 6, 8, 10]
```



- Explicación:
    - Usamos una función anónima dentro de map para iterar sobre el array numeros y devolver un nuevo array con cada número multiplicado por 2.
    - Esta función no necesita nombre porque se usa solo dentro del contexto de map.

3. Funciones Autoinvocadas (IIFE)

Las IIFE (Immediately Invoked Function Expression) son un tipo de función que se ejecuta automáticamente en el momento en que se define.

```
(function() {
  console.log("Esto se ejecuta inmediatamente");
})();
```

¿Para qué sirven?

- Encapsulación de código: A veces, necesitas que ciertas variables o funciones no estén disponibles globalmente, para evitar interferencias con otras partes de tu programa. Las IIFE crean un "alcance" (scope) privado.
- Ejecutar código inmediatamente: Si necesitas ejecutar una función inmediatamente, sin tener que llamarla después, una IIFE es la mejor opción.

Ventajas de las funciones anónimas

- Simplicidad: A veces no necesitas darle un nombre a una función porque solo la vas a usar una vez, por lo que puedes hacer el código más simple.
- Encapsulación: Las funciones anónimas autoinvocadas son útiles para crear un ámbito privado de variables, lo que evita que los valores interfieran con otras partes del código.
- Callbacks: Son ideales cuando trabajas con callbacks (funciones que se ejecutan después de otra función), ya que no necesitas declarar una función por separado si no la vas a reutilizar.

Desventajas de las funciones anónimas

- Difíciles de depurar: Como no tienen nombre, pueden ser más difíciles de identificar en las herramientas de depuración. Si algo falla dentro de una función anónima, la consola de errores no mostrará un nombre claro, solo que falló en una "función anónima".



- No reutilizables: Al no tener nombre, no se pueden reutilizar en otras partes del código. Si quieres usar el mismo comportamiento varias veces, es mejor usar una función con nombre.

7. Características avanzadas en ES6

ES6, o ECMAScript 6, también conocido como ECMAScript 2015, representó una de las actualizaciones más significativas al lenguaje JavaScript desde su creación. Esta versión introdujo una serie de características avanzadas que transformaron la forma en que los desarrolladores escriben y estructuran su código, haciéndolo más legible, eficiente y modular.

Entre las características más destacadas se encuentran la desestructuración (destructuring), el operador de expansión (spread operator) y las template strings. Estas funcionalidades están diseñadas para facilitar tareas comunes, reducir la repetición de código y mejorar la manipulación de estructuras complejas como objetos y arrays.

¿Por qué son importantes?

Antes de ES6, muchas de estas operaciones requerían código más largo y complicado. ES6 introdujo una sintaxis más concisa, lo que permite a los desarrolladores escribir menos código para realizar las mismas tareas, a la vez que mejora la claridad y legibilidad. Estas características también reflejan la evolución de JavaScript hacia un lenguaje más moderno y orientado a patrones comunes en la programación actual.

7.1. Destructuring

El destructuring en JavaScript es una técnica que permite extraer datos de arrays o de objetos y asignarlos a variables de manera más limpia.

Destructuring en objetos
Con el destructuring de objetos, puedes extraer múltiples propiedades a la vez.



```
let persona = {
  nombre: "Maria",
  edad: 28,
  profesion: "Ingeniera"
};

// Extraer propiedades:
let { nombre, edad, profesion } = persona;
console.log(nombre); // "Maria"
console.log(edad); // 28
console.log(profesion); // "Ingeniera"
```

También puedes asignar nuevas variables al extraer:

```
let { nombre: nombrePersona, edad: edadPersona } = persona;
console.log(nombrePersona); // "Maria"
```

Destructuring en arrays

El destructuring de arrays se basa en las posiciones de los elementos.

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

let [primero, segundo, tercero] = numeros;
console.log(primero); // 10
console.log(segundo); // 20
console.log(tercero); // 30
```

Valores por defecto en destructuring

Puedes asignar valores por defecto en caso de que la propiedad o el elemento no exista.

```
let { nombre, apellido = "Desconocido" } = { nombre: "Carlos" };
console.log(apellido); // "Desconocido"
```



7.2. Spread Operator (...)

El spread operator (...) es muy útil para copiar y fusionar objetos y arrays, entre otras cosas.

Spread en Arrays
Expande los elementos de un array en otro array.

```
let numeros = [1, 2, 3];
let nuevosNumeros = [...numeros, 4, 5];
console.log(nuevosNumeros); // [1, 2, 3, 4, 5]
```

También puedes usarlo para pasar elementos de un array como argumentos a una función.

```
function sumar(a, b, c) {
  return a + b + c;
}

let numeros = [1, 2, 3];
console.log(sumar(...numeros)); // 6
```

Spread en Objetos
Expande las propiedades de un objeto en otro objeto.

```
let persona = { nombre: "Carlos", edad: 25 };
let nuevaPersona = { ...persona, profesion: "Diseñador" };

console.log(nuevaPersona);
// { nombre: "Carlos", edad: 25, profesion: "Diseñador" }
```

El spread operator se usa comúnmente para crear copias superficiales de objetos o arrays, lo que ayuda a evitar la mutación accidental del objeto original.



7.3 Parámetros Rest (...rest)

JavaScript también tiene una funcionalidad especial llamada **parámetros rest**, que permite agrupar todos los argumentos adicionales en un solo array. Es útil cuando no sabes cuántos argumentos va a recibir tu función.

```
function sumarTodos(...numeros) {
  let suma = 0;
  for (let numero of numeros) {
    suma += numero;
  }
  console.log(suma);
}

sumarTodos(1, 2, 3, 4, 5); // Resultado: 15
```

- **Explicación**:
- El parámetro ...numeros agrupa todos los argumentos en un array.
- Luego, usamos un ciclo for para sumar todos los números.

Aunque el **spread operator** (...) y los **parámetros rest** (...) en JavaScript parecen similares porque ambos usan tres puntos (...), tienen propósitos y usos diferentes

Propósito:

- **Spread Operator**: Se utiliza para "descomponer" elementos de un array, objeto o cualquier iterable en elementos individuales. Sirve para expandir un array u objeto en lugares donde se esperan múltiples valores (por ejemplo, argumentos de una función o elementos en un array).
- **Parámetros Rest**: Se utilizan para agrupar varios argumentos pasados a una función en un solo array. Recoge un número indefinido de argumentos como un array dentro de una función.

Contexto de Uso:

- **Spread Operator**: Se usa cuando **expandes** los elementos de un iterable. Por ejemplo, cuando combinas arrays, clonas objetos, o pasas elementos como argumentos a una función.
- **Parámetros Rest**: Se usan dentro de una **definición de función** para agrupar todos los argumentos adicionales en un array. Esto permite recibir un número variable de argumentos.



Posición en el Código:

- Spread Operator: Se utiliza **fuera** de la definición de una función, en un contexto donde estás expandiendo los elementos de un iterable.
- Parámetros Rest: Solo se utilizan **dentro** de la definición de una función y solo una vez, como el último parámetro, para capturar los argumentos restantes.

Analogía:

- El **spread operator** es como abrir una caja y esparcir todo lo que contiene (descomponiendo elementos).
- Los **parámetros rest** son como meter todo lo que no se usa directamente en una caja (agrupando elementos).

Diferencias Clave

Característica	Spread Operator	Parámetros Rest
Uso	Expande un iterable en elementos individuales	Agrupa múltiples argumentos en un array
Lugar de uso	Fuera de funciones o en objetos	Dentro de la definición de una función
Posición	Puede estar en cualquier parte del código	Debe ser el último parámetro en una función
Función	Clonar, combinar arrays u objetos, pasar argumentos a funciones	Manejar un número indefinido de argumentos

7.4. template strings

Las **template strings** o **template literals** en JavaScript, introducidas en **ES6**, son una forma más potente y flexible de trabajar con cadenas de texto (strings). Estas permiten la interpolación de variables, la creación de cadenas multilínea y la incorporación de expresiones complejas dentro de las cadenas.

Las **template strings** se definen utilizando el **acento grave** o **backticks** (``) en lugar de las comillas simples (' ') o dobles (" "). Esto te permite integrar variables y expresiones directamente en la cadena sin necesidad de concatenar.



Ejemplo:

```
const nombre = "Ana";
const saludo = `Hola, ${nombre}!`;
console.log(saludo);  // "Hola, Ana!"
```

7.4.1. Interpolación de Variables

La interpolación de variables en las template strings se hace utilizando el símbolo ${}. Todo lo que esté dentro de los corchetes {} será evaluado e insertado en la cadena.

Ejemplo de interpolación:

```
const edad = 25;
const mensaje = `Tienes ${edad} años.`;
console.log(mensaje);  // "Tienes 25 años."
```

7.4.2. Interpolación de Expresiones

No solo puedes insertar variables, sino también expresiones complejas (como operaciones matemáticas, funciones, etc.).

Ejemplo con expresiones:

```
const a = 5;
const b = 10;
const resultado = `La suma de 5 + 10 es ${a + b}.`;
console.log(resultado);  // "La suma de 5 + 10 es 15."
```

7.4.3. Cadenas Multilínea

Las template strings permiten crear cadenas que se extienden en múltiples líneas sin necesidad de usar caracteres especiales como \n (nueva línea) en cadenas tradicionales.



Ejemplo de cadena multilínea:

```
const poema = `Este es un poema
que tiene múltiples líneas
sin necesidad de caracteres especiales.`;
console.log(poema);

// Salida:
/*
Este es un poema
que tiene múltiples líneas
sin necesidad de caracteres especiales.
*/
```

7.4.4. Uso con Funciones

Puedes utilizar funciones dentro de template literals para ejecutar lógica más avanzada.

Ejemplo con una función:

```
function getPrecioConIVA(precio) {
return precio * 1.21;
}

const producto = "ordenador";
const precio = 1000;
const mensaje = `El precio de un ${producto} con IVA es ${getPrecioConIVA(precio)} euros.`;
console.log(mensaje);  // "El precio de un ordenador con IVA es 1210 euros."
```

7.4.5. Interpolación de Objetos

Las template strings también permiten insertar propiedades de objetos directamente.



Ejemplo con objetos:

```
const persona = {
  nombre: "Carlos",
  edad: 30,
};

const presentacion = `Hola, me llamo ${persona.nombre} y tengo ${persona.edad} años.`;
console.log(presentacion);  // "Hola, me llamo Carlos y tengo 30 años."
```

7.4.6. Escapar los Backticks

Si necesitas incluir un backtick ` dentro de una template string, debes escapar utilizando una barra invertida \.

Ejemplo de escape de backticks:

```
const mensaje = `El carácter backtick (\`) es muy útil.`;
console.log(mensaje);  // "El carácter backtick (`) es muy útil."
```

7.4.7. Tagged Template Literals

Los tagged template literals te permiten utilizar una función especial para procesar una template string antes de que se interpolen las variables y las expresiones.

Ejemplo con tagged templates:

```
function etiqueta(cadenas, ...valores) {
  console.log(cadenas);  // ["Hola, ", " tienes ", " años."]
  console.log(valores);  // ["Ana", 25]
  return `${valores[0]}, tu edad es: ${valores[1]}`;
}

const nombre = "Ana";
const edad = 25;
const mensaje = etiqueta`Hola, ${nombre} tienes ${edad} años.`;
console.log(mensaje);  // "Ana, tu edad es: 25"
```



Ventajas de Usar Template Strings

- Legibilidad: Son mucho más fáciles de leer que las concatenaciones con el símbolo +.
- Código más limpio: Se evita el uso excesivo de concatenaciones, lo que simplifica el código.
- Multilínea sin esfuerzo: Crear cadenas de texto en varias líneas es más fácil y natural.
- Interpolación avanzada: Permiten incluir variables, expresiones, objetos y funciones de manera directa y legible.

Conclusión

Las template strings o template literals en ES6 proporcionan una manera más poderosa y flexible de trabajar con cadenas en JavaScript. Son fáciles de usar y muy útiles para la interpolación de variables, la creación de cadenas multilínea y el manejo de expresiones complejas, lo que mejora significativamente la legibilidad y mantenimiento del código.

8. Eventos en JavaScript

Los eventos en JavaScript son acciones que ocurren en el navegador, como hacer clic en un botón, pasar el ratón sobre un elemento, escribir en un campo de texto, etc. Puedes "escuchar" estos eventos y ejecutar código en respuesta a ellos.

¿Cómo funcionan los eventos?

Imagina que tienes un botón en una página web. Cuando haces clic en él, quieres que suceda algo. Esto lo logras asociando una función (lo que se llama un "manejador de eventos") al evento del clic.

Ejemplo de Evento:

- Clic en un botón: Cuando el usuario hace clic en un botón, eso genera un evento de clic.
- Envío de un formulario: Cuando el usuario envía un formulario, se genera un evento de envío.

8.1 Tipos de Eventos

JavaScript proporciona una amplia variedad de eventos que puedes usar en tus páginas web. Aquí hay algunos ejemplos comunes:



- Eventos del Mouse:
  - click: Se activa cuando se hace clic en un elemento.
  - dblclick: Se activa cuando se hace doble clic en un elemento.
  - mouseover: Se activa cuando el ratón se mueve sobre un elemento.
  - mouseout: Se activa cuando el ratón se mueve fuera de un elemento.

Ejemplo click:

```
<button id="miBoton">Haz clic aquí</button>

<script>
    document.getElementById("miBoton").addEventListener("click", function() {
        alert("¡Hiciste clic en el botón!");
    });
</script>
```

Ejemplo mouseover y mouseout:

```
<p id="miParrafo">Pasa el ratón por aquí</p>
<script>
    const parrafo = document.getElementById('miParrafo');

    parrafo.addEventListener('mouseover', function() {
        parrafo.style.color = 'red';
    });

    parrafo.addEventListener('mouseout', function() {
        parrafo.style.color = 'black';
    });
</script>
```

- Eventos del Teclado:
  - keydown: Se activa cuando se presiona una tecla.
  - keyup: Se activa cuando se suelta una tecla.
  - keypress: Se activa cuando se presiona una tecla que genera un carácter.



Ejemplo:

```
<input type="text" id="miInput" placeholder="Escribe algo...">
<script>
    document.getElementById('miInput').addEventListener('keydown', function(event) {
        console.log('Tecla presionada: ' + event.key);
    });
</script>
```

- Eventos de Formulario:
  - submit: Se activa cuando se envía un formulario.
  - change: Se activa cuando el valor de un elemento de formulario cambia.
  - focus: Se activa cuando un elemento de formulario recibe el foco.

Ejemplo submit:

```
<form id="miFormulario">
    <input type="text" id="nombre" placeholder="Nombre">
    <button type="submit">Enviar</button>
</form>

<script>
    document.getElementById("miFormulario").addEventListener("submit", function(event) {
        alert("Formulario enviado.");
    });
</script>
```



Ejemplo change y focus:

```
<input type="text" id="miCampo">
<script>
    const campo = document.getElementById('miCampo');

    campo.addEventListener('change', function() {
        console.log('Campo cambiado: ' + campo.value);
    });

    campo.addEventListener('focus', function() {
        console.log('Campo en foco');
    });
</script>
```

- Eventos de Carga:
  - load: Se activa cuando una página o un recurso se ha cargado completamente.
  - unload: Se activa cuando el usuario abandona la página.
  - beforeunload: Se activa justo antes de que el usuario abandone una página web o cierre la pestaña/navegador.
  - DOMContentLoaded: Se activa cuando el documento HTML ha sido completamente cargado y parseado, sin esperar a que se carguen los estilos o imágenes.

Ejemplo load:

```
<script>
    window.addEventListener('load', function() {
        console.log('Página completamente cargada');
    });
</script>
```



Ejemplo DOMContentLoaded:

```
<script>
    document.addEventListener('DOMContentLoaded', function() {
        console.log('DOM completamente cargado y parseado');
    });
</script>
```

Diferencias entre load, DOMContentLoaded, unload, beforeunload

Las propiedades unload y beforeunload son dos eventos en JavaScript que están relacionados con la navegación y la salida de una página web. Aunque su función parece similar, se utilizan en situaciones diferentes y tienen comportamientos distintos

Características del evento unload

- No interrumpe la navegación: No ofrece la posibilidad de detener o cancelar la acción de salir de la página.
- Uso típico: Generalmente, se utiliza para realizar tareas finales como cerrar conexiones o limpiar recursos, aunque su uso ha disminuido por razones de seguridad.
- Limitaciones: No puedes mostrar mensajes personalizados o interactuar con el usuario antes de que se cierre la página. No se garantiza que todas las acciones programadas se completen correctamente, ya que el navegador podría descargar rápidamente la página sin esperar a que el código termine de ejecutarse.

Características del evento beforeunload

- Interacción con el usuario: Puedes mostrar un cuadro de diálogo de confirmación que le pregunta al usuario si realmente quiere abandonar la página. Sin embargo, los navegadores modernos no permiten mostrar mensajes personalizados en este cuadro de diálogo por razones de seguridad.
- Detiene la navegación: El usuario tiene la opción de cancelar la acción de abandonar la página.
- Uso típico: Se utiliza cuando es importante advertir al usuario de que está a punto de perder datos no guardados (por ejemplo, si está completando un formulario o editando un documento).



Diferencias clave entre unload y beforeunload

Característica	unload	beforeunload
Momento en que se dispara	Cuando la página está a punto de descargarse.	Justo antes de que la página comience a descargarse.
Interacción con el usuario	No permite interacción con el usuario.	Permite advertir al usuario antes de salir de la página.
Posibilidad de detener la salida	No. No puede detener la salida de la página.	Sí. Permite mostrar un cuadro de diálogo de confirmación.
Mensajes personalizados	No es posible.	No es posible en navegadores modernos (cuadro estándar).
Propósito	Limpieza de recursos al abandonar la página.	Advertir al usuario sobre la salida (posible pérdida de datos).

¿Cuándo usar cada uno?

- unload: Si necesitas realizar tareas finales o limpieza cuando el usuario abandona la página (como cerrar conexiones, detener procesos o guardar en almacenamiento local).
- beforeunload: Cuando quieres advertir al usuario antes de que abandone la página, especialmente si hay datos no guardados o acciones que deberían completarse antes de salir.

Los eventos load y DOMContentLoaded son dos eventos importantes en JavaScript que indican que la página web ha sido cargada, pero difieren en cuándo y qué partes de la página han sido completamente procesadas

Características de DOMContentLoaded

- Momento: Se activa tan pronto como el DOM (Document Object Model) ha sido completamente construido, es decir, cuando se ha terminado de parsear el HTML.
- No espera recursos externos: No espera a que los recursos como imágenes, hojas de estilo o scripts externos hayan terminado de cargar.
- Uso típico: Se utiliza cuando necesitas manipular el DOM lo más pronto posible, sin depender de que otros recursos como imágenes o estilos hayan sido descargados.

Características de load

- Momento: Se activa después de que toda la página, incluidos todos los recursos externos, haya sido completamente descargada y cargada.



- Espera recursos externos: Espera a que las imágenes, archivos CSS, scripts y otros elementos multimedia terminen de cargarse antes de dispararse.
- Uso típico: Se utiliza cuando es importante asegurarse de que todos los recursos, incluidos los medios (imágenes, videos, etc.), estén completamente cargados antes de ejecutar el código.

Diferencias clave entre DOMContentLoaded y load

Característica	DOMContentLoaded	load
Momento de ejecución	Cuando el HTML ha sido completamente cargado y parseado.	Cuando todo el contenido de la página (incluidos recursos externos) ha sido cargado.
Espera por recursos externos	No. No espera a que las imágenes, hojas de estilo, etc., terminen de cargar.	Sí. Espera hasta que todos los recursos externos terminen de cargar.
Uso típico	Manipulación temprana del DOM (estructuras HTML).	Cuando necesitas asegurarte de que imágenes, estilos y otros recursos estén completamente cargados.
Velocidad de ejecución	Se ejecuta más rápido, ya que no espera por los recursos externos.	Se ejecuta más tarde, una vez que todos los recursos estén listos.
Aplicación	Ideal para manipular el DOM sin preocuparse por imágenes u otros archivos externos.	Ideal para interactuar con elementos multimedia que deben estar completamente cargados.

¿Cuándo usar cada uno?

- DOMContentLoaded: Si necesitas realizar tareas finales o limpieza cuando el usuario abandona la página (como cerrar conexiones, detener procesos o guardar en almacenamiento local).
- load: Cuando quieres advertir al usuario antes de que abandone la página, especialmente si hay datos no guardados o acciones que deberían completarse antes de salir.

- Eventos Personalizados

Puedes crear tus propios eventos personalizados usando el constructor CustomEvent.



Ejemplo:

```
<button id="miBoton">Crear Evento</button>
<script>
    const boton = document.getElementById('miBoton');

    boton.addEventListener('click', function() {
        const eventoPersonalizado = new CustomEvent('miEvento', { detail: 'Información adicional' });
        document.dispatchEvent(eventoPersonalizado);
    });

    document.addEventListener('miEvento', function(event) {
        console.log('Evento personalizado recibido con detalle:', event.detail);
    });
</script>
```

Los eventos en JavaScript son una parte fundamental para hacer páginas web interactivas y dinámicas. Entender cómo manejarlos te permitirá responder a las acciones del usuario y a cambios en el estado de la página de manera efectiva.

8.2 Manejo de Eventos

Para manejar eventos en JavaScript, puedes usar dos métodos principales:

- Atributos de Evento en HTML
  Puedes agregar atributos de evento directamente en el HTML. Estos atributos especifican qué función de JavaScript se debe ejecutar cuando ocurre el evento.



Ejemplo:

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Eventos en HTML</title>
</head>
<body>
    <button onclick="mostrarMensaje()">Haz clic aquí</button>

    <script>
        function mostrarMensaje() {
            alert('¡Hola! Has hecho clic en el botón.');
        }
    </script>
</body>
</html>
```

Ventajas:

- Simplicidad: Es fácil de implementar cuando tienes un sitio pequeño o solo necesitas añadir un evento rápidamente.
- Legibilidad en HTML: Es posible ver rápidamente qué eventos están asociados con cada elemento al mirar el código HTML.

Desventajas:

- Separación de responsabilidades: Combina el comportamiento (JavaScript) con la estructura (HTML), lo que rompe el principio de mantener el código HTML, CSS y JavaScript separados.
- No es escalable: A medida que el proyecto crece, tener muchos eventos declarados en HTML puede hacer que el código sea más difícil de mantener.
- Solo permite un evento por tipo: Si intentas añadir más de un evento al mismo elemento con el mismo tipo de evento (como dos eventos onclick), el último sobrescribirá a los anteriores.



- Métodos de Eventos en JavaScript
  También puedes añadir manejadores de eventos en JavaScript usando el método addEventListener(). Este método permite agregar múltiples eventos a un solo elemento y proporciona más flexibilidad.

  Ejemplo:

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Eventos con addEventListener</title>
</head>
<body>
    <button id="miBoton">Haz clic aquí</button>

    <script>
        const boton = document.getElementById('miBoton');

        boton.addEventListener('click', function() {
            alert('¡Hola! Has hecho clic en el botón.');
        });
    </script>
</body>
</html>
```

  Ventajas:
  - Separación de responsabilidades: El HTML se mantiene limpio y sin lógica JavaScript, lo que facilita la organización y mantenimiento del código.
  - Escalabilidad: Permite agregar múltiples eventos al mismo elemento sin sobrescribir los anteriores. Además, es más fácil gestionar el código cuando hay muchos eventos distribuidos en distintos elementos.
  - Compatibilidad con múltiples eventos: Puedes asociar más de un evento del mismo tipo a un solo elemento, y todos se ejecutarán.



Desventajas:

- Curva de aprendizaje ligeramente más alta: Para alguien que apenas comienza a aprender JavaScript, puede parecer más complicado al principio.
- No es visible desde el HTML: No puedes ver inmediatamente qué eventos están asociados a los elementos al revisar solo el HTML, ya que la lógica está en los archivos JavaScript.

Diferencias Clave

Aspecto	Eventos en atributos HTML (onclick)	Eventos con addEventListener()
Separación de responsabilidades	Mezcla el JavaScript con el HTML, dificultando la legibilidad.	Separa HTML y JavaScript, siguiendo buenas prácticas.
Múltiples eventos	No permite agregar varios eventos del mismo tipo a un solo elemento.	Permite agregar múltiples eventos del mismo tipo sin sobrescribir.
Escalabilidad	Dificultad para manejar eventos en grandes proyectos.	Escalable y más fácil de gestionar en proyectos grandes.
Uso de eventos dinámicos	Menos flexible para añadir o eliminar eventos dinámicamente.	Ideal para eventos dinámicos que pueden agregarse o removerse.

Casos en los que conviene usar addEventListener()

- Proyectos medianos o grandes: Si estás trabajando en una aplicación que puede crecer, addEventListener() te permite mantener una mejor organización del código.
- Manejo de múltiples eventos: Si necesitas que un elemento tenga más de un evento asociado, addEventListener() es la opción ideal.
- Aplicaciones interactivas: Cuando la página necesita escuchar muchos eventos, como formularios dinámicos, galerías de imágenes o interfaces de usuario complejas.

Casos en los que conviene usar eventos en atributos HTML (onclick, onmouseover, etc.)

- Prototipos rápidos: Si estás creando una pequeña demostración o un prototipo, usar eventos en los atributos HTML puede ser más rápido y fácil.



- Proyectos pequeños: En sitios pequeños con pocas interacciones, puede ser suficiente.

Conclusión
Aunque declarar eventos como atributos HTML puede ser conveniente en ciertos casos sencillos, el uso de addEventListener() es generalmente una mejor práctica, especialmente en proyectos que crecen en complejidad. Este método te brinda mayor flexibilidad, escalabilidad y una mejor separación de responsabilidades entre el HTML y el JavaScript.

8.3. Prevención de eventos con preventDefault()

El método preventDefault() se utiliza para prevenir el comportamiento predeterminado de un evento. Por ejemplo, puedes evitar que un formulario se envíe o que un enlace redirija a una nueva página.

Ejemplo: Imaginemos que tienes un formulario que no deseas enviar si el usuario no ha completado un campo obligatorio.

```
<form id="miFormulario">
    <input type="text" id="nombre" placeholder="Nombre">
    <button type="submit">Enviar</button>
</form>

<script>
    document.getElementById("miFormulario").addEventListener("submit", function(event) {
        let nombre = document.getElementById("nombre").value;
        if (nombre === "") {
            alert("El nombre es obligatorio.");
            event.preventDefault(); // Prevenir el envío del formulario
        }
    });
</script>
```

En este ejemplo, si el campo de nombre está vacío, el formulario no se enviará y se mostrará una alerta.



9. DOM (DOCUMENT OBJECT MODEL)

El DOM (Document Object Model) es un concepto fundamental en el desarrollo web que te permite interactuar con los documentos HTML o XML a través de JavaScript. En esta guía vamos a cubrir en profundidad qué es el DOM, cómo funciona, para qué sirve, y cómo puedes manipularlo usando JavaScript, incluyendo los métodos más utilizados para agregar, modificar y eliminar elementos.

El DOM es una representación estructurada del documento HTML o XML que se carga en el navegador. Puedes imaginarlo como un árbol en el que cada nodo representa una parte del documento. Estos nodos pueden ser elementos (como un párrafo <p>, una lista <ul>, etc.), atributos (como class, id), o contenido textual.

El navegador crea este modelo en memoria cuando carga la página web, lo que permite a JavaScript interactuar con los elementos, modificarlos o crear nuevos.

9.1 ¿Cómo Funciona el DOM?

El DOM organiza el contenido del HTML como una estructura jerárquica (árbol), donde cada parte del HTML es un nodo en ese árbol. Hay varios tipos de nodos, pero los más importantes son:

- Nodo de Elemento: Cada etiqueta HTML se representa como un nodo de elemento. Ejemplo: , <p>, <ul>, etc.
- Nodo de Texto: Cada fragmento de texto dentro de un elemento es un nodo de texto.
- Nodo de Atributo: Los atributos (como class, id) son considerados nodos en el DOM, pero no se muestran como parte del árbol principal.



Ejemplo de un Documento HTML y su Estructura DOM:

```html
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Ejemplo DOM</title>
</head>
<body>
    <h1>Título de la Página</h1>
    <p>Este es un párrafo de ejemplo.</p>
</body>
</html>
```

¿Para Qué Sirve el DOM?

El DOM es útil porque permite que JavaScript manipule el contenido de una página web sin tener que recargarla. Con el DOM, puedes hacer cosas como:

- Agregar nuevos elementos a la página.
- Modificar o eliminar elementos existentes.
- Cambiar el contenido o los estilos de los elementos.
- Responder a eventos del usuario (clics, teclas presionadas, etc.).

9.2 Interacción con el DOM usando JavaScript

JavaScript ofrece varios métodos y propiedades para interactuar con el DOM. Aquí cubriremos los métodos más usados para seleccionar, agregar, modificar, y eliminar elementos.

9.2.1. Seleccionar Elementos en el DOM

1. document.getElementById()

Este método selecciona un elemento basado en su id.



```
<h1 id="titulo">Hola Mundo</h1>
<script>
    const elemento = document.getElementById('titulo');
    console.log(elemento); // <h1 id="titulo">Hola Mundo</h1>
</script>
```

2. document.getElementsByClassName()

Selecciona todos los elementos que tienen una clase específica. Devuelve un *HTMLCollection* (una colección de elementos).

```
<p class="texto">Primer párrafo</p>
<p class="texto">Segundo párrafo</p>
<script>
    const elementos = document.getElementsByClassName('texto');
    console.log(elementos); // HTMLCollection con los dos párrafos
</script>
```

3. document.querySelector()

Selecciona el primer elemento que coincide con un selector CSS. Muy versátil porque puedes seleccionar por id, clase, etiqueta, etc.

```
<p class="texto">Hola</p>
<p class="texto">Mundo</p>
<script>
    const elemento = document.querySelector('.texto');
    console.log(elemento); // <p class="texto">Hola</p>
</script>
```

4. document.querySelectorAll()

Selecciona todos los elementos que coinciden con un selector CSS y devuelve un NodeList, que es como un array.



```
<p class="texto">Hola</p>
<p class="texto">Mundo</p>
<script>
    const elementos = document.querySelectorAll('.texto');
    console.log(elementos); // NodeList con ambos párrafos
</script>
```

9.2.2. Agregar Elementos al DOM

Puedes agregar nuevos elementos a tu documento HTML dinámicamente usando JavaScript y el DOM.

1. createElement()

Este método crea un nuevo elemento HTML que puedes agregar a la página.

```
<div id="contenedor"></div>
<script>
    const nuevoElemento = document.createElement('p');
    nuevoElemento.textContent = 'Este es un nuevo párrafo.';
    document.getElementById('contenedor').appendChild(nuevoElemento);
</script>
```

Aquí se crea un nuevo párrafo y se añade al div con el id="contenedor".

2. insertBefore()

Puedes insertar un elemento antes de otro dentro del DOM.



```
<div id="contenedor">
    <p id="parrafoExistente">Párrafo existente</p>
</div>
<script>
    const nuevoElemento = document.createElement('p');
    nuevoElemento.textContent = 'Párrafo antes del existente';
    const contenedor = document.getElementById('contenedor');
    const parrafoExistente = document.getElementById('parrafoExistente');
    contenedor.insertBefore(nuevoElemento, parrafoExistente);
</script>
```

9.2.3. Modificar Elementos en el DOM

1. Cambiar el Texto con textContent

Puedes cambiar el texto de un elemento usando la propiedad textContent.

```
<h1 id="titulo">Texto original</h1>
<script>
    const titulo = document.getElementById('titulo');
    titulo.textContent = 'Texto cambiado';
</script>
```

2. Modificar Atributos con setAttribute()

Puedes modificar o agregar atributos a un elemento, como src de una imagen o href de un enlace.

```
<img id="miImagen" src="imagen1.jpg" alt="Imagen">
<script>
    const imagen = document.getElementById('miImagen');
    imagen.setAttribute('src', 'imagen2.jpg'); // Cambia la fuente de la imagen
</script>
```



9.2.4. Eliminar Elementos del DOM

1. removeChild()

Puedes eliminar un elemento hijo de su contenedor padre.

```
<div id="contenedor">
    <p id="parrafo">Este es un párrafo</p>
</div>
<script>
    const contenedor = document.getElementById('contenedor');
    const parrafo = document.getElementById('parrafo');
    contenedor.removeChild(parrafo); // Elimina el párrafo
</script>
```

9.2.5. Métodos Útiles del DOM

1. innerHTML

Permite obtener o establecer el HTML dentro de un elemento. Se usa para modificar contenido completo (texto y etiquetas).

```
<div id="contenedor"></div>
<script>
    const contenedor = document.getElementById('contenedor');
    contenedor.innerHTML = '<p>Nuevo contenido con <strong>HTML</strong></p>';
</script>
```

Advertencia: innerHTML puede ser inseguro si se usa con datos no confiables, ya que puede abrir la puerta a ataques XSS (Cross-site Scripting).

2. classList

Permite agregar, quitar y verificar clases de un elemento de manera eficiente.



```
<div id="caja" class="rojo"></div>
<script>
    const caja = document.getElementById('caja');
    caja.classList.add('azul');    // Agrega una clase
    caja.classList.remove('rojo'); // Elimina una clase
    caja.classList.toggle('verde'); // Alterna una clase
</script>
```

Conclusión

El DOM es una herramienta poderosa que te permite manipular los documentos HTML dinámicamente. Al dominar su uso, puedes crear interactividad, cambiar contenido sobre la marcha y responder a los eventos del usuario sin necesidad de recargar la página.

10. JSON (JavaScript Object Notation)

JSON es un formato de texto ligero para intercambiar datos. Es ampliamente utilizado en aplicaciones web para enviar y recibir información entre el servidor y el cliente. JSON está basado en la sintaxis de objetos de JavaScript, pero puede ser usado por muchos lenguajes de programación.

Estructura de JSON

Un archivo JSON se organiza en pares de clave-valor. Aquí tienes un ejemplo básico de cómo se ve:

```
{
  "nombre": "Juan",
  "edad": 30,
  "hobbies": ["leer", "jugar fútbol", "viajar"],
  "casado": true
}
```

- "nombre" es una clave y "Juan" es su valor.
- "edad" tiene un valor numérico de 30.
- "hobbies" es un array de tres elementos.



- "casado" es un valor booleano (true).

En un archivo JSON, los tipos de datos permitidos son limitados a aquellos que se pueden representar de forma universal y que son compatibles con diferentes lenguajes de programación.

Datos permitidos:

- string
- number
- boolean
- null
- array
- object

Datos no permitidos:

- Funciones
- Objetos del DOM
- Valores indefinidos
- Symbol

10.1. Convertir de Objeto a JSON con JSON.stringify()

La función JSON.stringify() se utiliza para convertir un objeto de JavaScript en una cadena de texto en formato JSON (JavaScript Object Notation). Esto es útil cuando necesitas enviar datos a través de una red o almacenarlos en una base de datos, ya que el formato JSON es legible tanto para humanos como para máquinas.

```
const persona = {
    nombre: "Ana",
    edad: 28,
    ocupacion: "Ingeniera"
};

const jsonPersona = JSON.stringify(persona);
console.log(jsonPersona);
// Resultado: '{"nombre":"Ana","edad":28,"ocupacion":"Ingeniera"}'
```

Aquí hemos convertido el objeto persona en una cadena de texto JSON que puede ser enviada o almacenada fácilmente.



Uso de JSON.stringify() con Arrays
También puedes usar JSON.stringify() con arrays para convertirlos a formato JSON.

```
const numeros = [1, 2, 3, 4, 5];
const jsonNumeros = JSON.stringify(numeros);
console.log(jsonNumeros);
// Resultado: '[1,2,3,4,5]'
```

Convertir un objeto con propiedades anidadas

```
const empresa = {
    nombre: "TechCorp",
    empleados: [
        { nombre: "Carlos", puesto: "Desarrollador" },
        { nombre: "Lucía", puesto: "Diseñadora" }
    ]
};

const jsonEmpresa = JSON.stringify(empresa);
console.log(jsonEmpresa);
// Resultado:
// '{"nombre":"TechCorp","empleados":[{"nombre":"Carlos","puesto":"Desarrollador"},{"nombre":"Lucía","puesto":"Diseñadora"}]}'
```

10.2. Convertir de JSON a Objeto con JSON.parse()

La función JSON.parse() se utiliza para convertir una cadena de texto en formato JSON de vuelta a un objeto de JavaScript. Esto es útil cuando recibes datos JSON de una API o de una base de datos y necesitas manipularlos en tu código.

```
const jsonPersona = '{"nombre":"Ana","edad":28,"ocupacion":"Ingeniera"}';
const personaObjeto = JSON.parse(jsonPersona);
console.log(personaObjeto);
// Resultado: { nombre: "Ana", edad: 28, ocupacion: "Ingeniera" }
```

En este ejemplo, la cadena de texto JSON ha sido convertida de nuevo a un objeto de JavaScript para su manipulación.

Uso de JSON.parse() con Arrays
También puedes convertir arrays en formato JSON de nuevo a arrays de JavaScript.



```
const jsonNumeros = '[1,2,3,4,5]';
const numerosArray = JSON.parse(jsonNumeros);
console.log(numerosArray);
// Resultado: [1, 2, 3, 4, 5]
```

Convertir una cadena JSON con objetos anidados:

```
const jsonEmpresa = '{"nombre":"TechCorp","empleados":[{"nombre":"Carlos","puesto":"Desarrollador"},{"nombre":"Lucía","puesto":"Diseñadora"}]}';
const empresaObjeto = JSON.parse(jsonEmpresa);
console.log(empresaObjeto);
// Resultado: { nombre: "TechCorp", empleados: [ { nombre: "Carlos", puesto: "Desarrollador" }, { nombre: "Lucía", puesto: "Diseñadora" } ] }
```

Diferencias clave entre JSON.stringify() y JSON.parse()

Función	Acción	Ejemplo de uso
JSON.stringify()	Convierte un objeto a una cadena JSON	Convertir un objeto para enviarlo a un servidor o almacenarlo en una base de datos.
JSON.parse()	Convierte una cadena JSON a un objeto	Convertir datos JSON recibidos de una API en un objeto para manipularlos en JavaScript.

10.3. Diferencia entre JSON y XML

JSON y XML son formatos populares para el intercambio de datos, pero tienen diferencias significativas:

Característica	JSON	XML
Sintaxis	Basado en la sintaxis de objetos de JavaScript.	Basado en etiquetas (similar a HTML).
Tamaño	Más ligero, ocupa menos espacio.	Más pesado debido a las etiquetas.
Legibilidad	Más fácil de leer y entender.	Más difícil de leer por su estructura compleja.
Flexibilidad de Tipos	Soporta arrays, objetos y tipos de datos simples.	Todo es representado como texto.
Comentarios	No permite comentarios.	Permite comentarios.



Uso común	APIs, aplicaciones web modernas, bases de datos NoSQL.	Documentos y configuraciones estructuradas.
Compatibilidad	Específico para el intercambio de datos.	Puede ser utilizado para estructurar documentos completos.

Ejemplo de JSON:

```
{
   "nombre": "Ana",
   "edad": 28,
   "ocupacion": "Ingeniera"
}
```

Ejemplo de XML:

```
<persona>
   <nombre>Ana</nombre>
   <edad>28</edad>
   <ocupacion>Ingeniera</ocupacion>
</persona>
```

10.4. Agregar, Eliminar y Modificar Datos de un JSON

Una vez que tienes un objeto JavaScript basado en JSON, puedes agregar, eliminar o modificar datos de manera dinámica. Para esto, se usan operaciones estándar de objetos en JavaScript.

Agregar un nuevo valor a un JSON

- Agregar una nueva clave-valor:
  - Simplemente se asigna una nueva propiedad al objeto.

```
const persona = { nombre: "Ana", edad: 28 };
persona.ocupacion = "Ingeniera";
console.log(persona);
// Resultado: { nombre: "Ana", edad: 28, ocupacion: "Ingeniera" }
```



- Agregar un nuevo elemento a un array:
  - Usamos el método .push() para agregar elementos.

```
const persona = { nombre: "Ana", hobbies: ["leer"] };
persona.hobbies.push("correr");
console.log(persona);
// Resultado: { nombre: "Ana", hobbies: ["leer", "correr"] }
```

Eliminar un valor de un JSON

- Eliminar una propiedad:
  - Usamos la palabra clave delete.

```
const persona = { nombre: "Ana", edad: 28 };
delete persona.edad;
console.log(persona);
// Resultado: { nombre: "Ana" }
```

- Eliminar un elemento de un array:
  - Usamos el método .splice() para eliminar elementos del array.

```
const persona = { nombre: "Ana", hobbies: ["leer", "correr"] };
persona.hobbies.splice(1, 1);
console.log(persona);
// Resultado: { nombre: "Ana", hobbies: ["leer"] }
```

Modificar un valor en un JSON

- Modificar una clave-valor:
  - Asignamos un nuevo valor a la propiedad existente.

```
const persona = { nombre: "Ana", edad: 28 };
persona.edad = 30;
console.log(persona);
// Resultado: { nombre: "Ana", edad: 30 }
```



- Modificar un elemento de un array:
  - Modificamos el valor directamente por su índice.

```
const persona = { nombre: "Ana", hobbies: ["leer", "correr"] };
persona.hobbies[1] = "nadar";
console.log(persona);
// Resultado: { nombre: "Ana", hobbies: ["leer", "nadar"] }
```

Con estas técnicas puedes manipular fácilmente los datos de un JSON en JavaScript.

10.5. JSON Anidados

Los JSON anidados son estructuras en las que un objeto o un array dentro de un archivo JSON contiene otros objetos o arrays. Esto permite representar datos complejos de una forma más organizada y estructurada.

¿Cómo funcionan los JSON anidados?

- Objeto dentro de otro objeto: Un objeto puede contener como valor otro objeto, lo que permite organizar datos relacionados.

  Ejemplo:

```
{
   "nombre": "Ana",
   "edad": 28,
   "direccion": {
      "calle": "Calle 123",
      "ciudad": "Bogotá",
      "pais": "Colombia"
   }
}
```

  - Aquí, el valor de la propiedad direccion es otro objeto JSON con las claves calle, ciudad y pais.



- Array dentro de un objeto: Un objeto también puede contener arrays como valores. Estos arrays pueden tener valores primitivos o incluso otros objetos.

  Ejemplo:

```
{
   "nombre": "Ana",
   "edad": 28,
   "hobbies": ["leer", "correr", "cocinar"]
}
```

  - En este caso, hobbies es un array de cadenas.

- Array de objetos: Los arrays pueden contener objetos, lo que es útil para representar listas de elementos más complejos.

  Ejemplo:

```
{
   "nombre": "Ana",
   "edad": 28,
   "amigos": [
      {
         "nombre": "Luis",
         "edad": 29
      },
      {
         "nombre": "Carla",
         "edad": 27
      }
   ]
}
```

  - Aquí, amigos es un array que contiene dos objetos, cada uno con las propiedades nombre y edad.



Acceso a datos en JSON anidados

Para acceder a los datos en un JSON anidado usando JavaScript, simplemente se accede a las propiedades como lo harías en un objeto común, utilizando la notación de punto (.) o los corchetes ([]).

Ejemplos en JavaScript

- Acceder a un objeto anidado:

```
const persona = {
   nombre: "Ana",
   direccion: {
      calle: "Calle 123",
      ciudad: "Bogotá"
   }
};

console.log(persona.direccion.ciudad); // Resultado: "Bogotá"
```

- Acceder a un array dentro de un objeto:

```
const persona = {
   nombre: "Ana",
   hobbies: ["leer", "correr", "cocinar"]
};

console.log(persona.hobbies[1]); // Resultado: "correr"
```



- Acceder a un array de objetos:

```
const persona = {
   nombre: "Ana",
   amigos: [
      { nombre: "Luis", edad: 29 },
      { nombre: "Carla", edad: 27 }
   ]
};

console.log(persona.amigos[0].nombre); // Resultado: "Luis"
```

Uso práctico de JSON anidados

Los JSON anidados se utilizan frecuentemente en aplicaciones que manejan datos complejos, como las respuestas de una API que pueden incluir detalles de usuarios, productos, ubicaciones, etc.



Ejemplo de un JSON anidado en una API de comercio electrónico:

```json
{
   "producto": {
      "id": 12345,
      "nombre": "Laptop",
      "caracteristicas": {
         "marca": "TechBrand",
         "procesador": "Intel i7",
         "ram": "16GB"
      },
      "precios": [
         {
            "moneda": "USD",
            "valor": 999.99
         },
         {
            "moneda": "EUR",
            "valor": 899.99
         }
      ]
   }
}
```

- En este ejemplo, se tiene un objeto producto que contiene propiedades anidadas como características (otro objeto) y precios (un array de objetos).



Manipulación de JSON anidado en JavaScript

- Modificar un valor en un objeto anidado:

```
const persona = {
   nombre: "Ana",
   direccion: {
      calle: "Calle 123",
      ciudad: "Bogotá"
   }
};

persona.direccion.ciudad = "Medellín";
console.log(persona.direccion.ciudad); // Resultado: "Medellín"
```

- Agregar un nuevo objeto a un array de objetos:

```
const persona = {
   nombre: "Ana",
   amigos: [
      { nombre: "Luis", edad: 29 }
   ]
};

persona.amigos.push({ nombre: "Carla", edad: 27 });
console.log(persona.amigos);
// Resultado: [{ nombre: "Luis", edad: 29 }, { nombre: "Carla", edad: 27 }]
```

Los JSON anidados son esenciales para manejar datos jerárquicos y complejos en JavaScript, facilitando la organización y manipulación eficiente de la información.

10.6. Ventajas y desventajas de JSON

- Ventajas:
  - Ligero y fácil de leer/escribir.
  - Formato universalmente aceptado para intercambio de datos.



- Basado en la sintaxis de objetos de JavaScript, por lo que es fácil de usar en aplicaciones web.}

- Desventajas:
  - No soporta funciones ni métodos, solo datos.
  - No soporta comentarios dentro del archivo JSON (a diferencia de otros formatos como XML).

11. Expresiones Regulares

Una **expresión regular** es una secuencia de caracteres que forma un patrón de búsqueda. Este patrón puede ser utilizado para buscar, reemplazar o validar datos en cadenas de texto.

Ejemplo:
Supongamos que tienes una lista de correos electrónicos y deseas validar si tienen el formato correcto, como "usuario@dominio.com". En lugar de verificar cada carácter manualmente, puedes usar una expresión regular que verifique si el correo sigue el patrón adecuado de un email.

11.1. ¿Cómo se construye una expresión regular?

Una expresión regular se define entre dos **barras inclinadas** (/) en JavaScript. Además, puede contener una serie de caracteres especiales que permiten definir el patrón de búsqueda.

```
let regex = /patrón/;
```

Componentes principales de una RegEx:

- **Literales**: Son caracteres normales que deben aparecer en la cadena. Ejemplo: /hola/ busca la palabra "hola".
- **Metacaracteres**: Son caracteres con un significado especial. Ejemplo: . coincide con cualquier carácter.
- **Modificadores**: Se añaden después de la barra inclinada (/) para modificar el comportamiento de la RegEx. Ejemplo: /hola/i (la i indica que la búsqueda es insensible a mayúsculas/minúsculas).



11.2. Funciones para trabajar con expresiones regulares en JavaScript

JavaScript proporciona varios métodos para usar expresiones regulares:

- test()
  Este método verifica si una cadena coincide con una expresión regular. Devuelve true o false.

  Ejemplo:

```
let regex = /hola/;
console.log(regex.test("hola mundo")); // true
console.log(regex.test("adiós mundo")); // false
```

- exec()
  Este método busca una coincidencia y devuelve un array con los resultados, o null si no encuentra coincidencias.

  Ejemplo:

```
let regex = /hola/;
console.log(regex.exec("hola mundo")); // ["hola"]
console.log(regex.exec("adiós mundo")); // null
```

- match()
  El método match() es utilizado en las cadenas de texto para buscar coincidencias.

  Ejemplo:

```
let texto = "La temperatura es de 25 grados.";
let regex = /\d+/; // Busca uno o más dígitos
console.log(texto.match(regex)); // ["25"]
```

- replace()
  Este método se utiliza para reemplazar texto que coincida con una expresión regular.



Ejemplo:

```
let texto = "hola mundo";
let regex = /mundo/;
console.log(texto.replace(regex, "amigo")); // "hola amigo"
```

- search()
  El método search() busca la posición donde comienza la primera coincidencia.

  Ejemplo:

```
let texto = "hola mundo";
let regex = /mundo/;
console.log(texto.search(regex)); // 5 (posición donde comienza "mundo")
```

11.3. Caracteres especiales (metacaracteres)

Los caracteres especiales en las expresiones regulares tienen funciones específicas:

- .: Coincide con cualquier carácter excepto una nueva línea (\n).
- ^: Marca el inicio de una cadena.
- $: Marca el final de una cadena.
- \d: Coincide con cualquier dígito (0-9).
- \w: Coincide con cualquier carácter alfanumérico (letras, números y guion bajo).
- \s: Coincide con cualquier espacio en blanco (espacio, tabulación, salto de línea).
- \b: Coincide con un límite de palabra.
- []: Define una clase de caracteres. Ejemplo: [aeiou] busca cualquier vocal.
- |: Funciona como un "o" lógico. Ejemplo: hola|adios busca "hola" o "adios".

Ejemplo con metacaracteres:

```
let regex = /\d{3}-\d{2}-\d{4}/; // Coincide con un formato de número de teléfono
console.log(regex.test("123-45-6789")); // true
```



11.4. Cuantificadores

Los cuantificadores permiten definir cuántas veces un carácter puede aparecer.

- *: Cero o más veces.
- +: Una o más veces.
- ¿: Cero o una vez.
- {n}: Exactamente n veces.
- {n,}: Al menos n veces.
- {n,m}: Entre n y m veces.

Ejemplo de cuantificadores:

```
let regex = /\d{3,5}/; // Coincide con entre 3 y 5 dígitos
console.log(regex.test("12345")); // true
console.log(regex.test("12")); // false
```

11.5. Grupos de captura

Los **grupos de captura** permiten almacenar partes específicas de una coincidencia.

- Se definen usando **paréntesis** () alrededor del texto que deseas capturar.

Ejemplo:

```
let regex = /(\d{3})-(\d{2})-(\d{4})/;
let resultado = regex.exec("123-45-6789");

console.log(resultado[1]); // "123"
console.log(resultado[2]); // "45"
console.log(resultado[3]); // "6789"
```

11.6. Expresiones regulares avanzadas: Anclas y Lookahead

Anclas ^ y $

- ^: Coincide con el inicio de la cadena.
- $: Coincide con el final de la cadena.



Ejemplo:

```
let regex = /^hola/; // Coincide con "hola" solo si está al inicio
console.log(regex.test("hola mundo")); // true
console.log(regex.test("mundo hola")); // false
```

Lookahead positivo y negativo

- Un lookahead positivo busca si un patrón está seguido por otro patrón específico.
- Un lookahead negativo busca si un patrón no está seguido por otro patrón.

Ejemplo:

```
let regexPositivo = /hola(?= mundo)/;
console.log(regexPositivo.test("hola mundo")); // true
console.log(regexPositivo.test("hola amigo")); // false

let regexNegativo = /hola(?! mundo)/;
console.log(regexNegativo.test("hola mundo")); // false
console.log(regexNegativo.test("hola amigo")); // true
```

Aplicación práctica: Validación de correo electrónico

Ejemplo:
Validar que un correo electrónico tenga el formato adecuado (por ejemplo, "usuario@dominio.com").

Expresión regular:

```
let emailRegex = /^[\w-\.]+@([\w-]+\.)+[\w-]{2,4}$/;
console.log(emailRegex.test("usuario@dominio.com")); // true
console.log(emailRegex.test("usuario@dominio")); // false
```



12. Almacenamiento en JavaScript

El almacenamiento en JavaScript se refiere a las diferentes formas de guardar datos en el navegador del usuario, permitiendo que una aplicación web mantenga información entre diferentes sesiones, páginas o incluso después de cerrar el navegador. Esto es útil para almacenar configuraciones, preferencias del usuario, carritos de compras, entre otros.

En JavaScript, existen tres mecanismos principales para el almacenamiento de datos en el navegador:

- Cookies
- localStorage
- sessionStorage

12.1. Cookies

Una cookie es un pequeño archivo de texto que se almacena en el navegador del usuario. Se utiliza para recordar información entre sesiones, como datos de inicio de sesión, preferencias del usuario, o el estado del carrito de compras.

- Analogía de la vida diaria: Las cookies son como un cuaderno de notas que llevas a todas partes. Cuando visitas un sitio web (como una tienda), este puede anotar en tu cuaderno detalles como tus preferencias. La próxima vez que visites, la tienda puede revisar tu cuaderno y mostrarte contenido personalizado.

Características principales de las cookies

- Duración: Puedes definir cuánto tiempo debe durar la cookie antes de expirar.
- Tamaño limitado: Las cookies suelen tener un tamaño máximo de 4KB.
- Alcance: Las cookies pueden ser específicas de un dominio y una ruta en particular.

Cómo crear, leer y eliminar cookies con JavaScript

- Crear una cookie:

```
document.cookie = "usuario=Juan; expires=Thu, 31 Dec 2024 12:00:00 UTC; path=/";
```

  - usuario=Juan: El valor de la cookie.
  - expires: Fecha de expiración. Si no se especifica, la cookie se eliminará cuando se cierre el navegador.



- path=/: Especifica en qué rutas del sitio la cookie es válida.

- Leer una cookie:

```
let cookies = document.cookie;
console.log(cookies);  // Muestra todas las cookies almacenadas
```

Las cookies se devuelven como una cadena de texto donde cada par clave=valor está separado por punto y coma.

- Eliminar una cookie: Para eliminar una cookie, puedes establecer su fecha de expiración en el pasado.

```
document.cookie = "usuario=; expires=Thu, 01 Jan 1970 00:00:00 UTC; path=/;";
```

Ventajas y desventajas de las cookies

- Ventajas:
  - Son útiles para almacenar pequeñas cantidades de información persistente en el navegador.
  - Ayudan a recordar preferencias de usuario y sesiones entre visitas.

- Desventajas:
  - Tienen limitaciones de tamaño (4KB por cookie).
  - No son seguras para almacenar información sensible como contraseñas (mejor usar localStorage o sessionStorage para datos temporales).
  - Cada cookie se envía al servidor en cada solicitud HTTP, lo que puede afectar el rendimiento si hay muchas cookies.

Cuando usar cookies

- Se usan para almacenar información que debe persistir entre sesiones o entre visitas a un sitio web, como credenciales de inicio de sesión (con precauciones de seguridad) o preferencias del usuario (tema claro/oscuro, idioma preferido, etc.).



12.2. LocalStorage

LocalStorage es una característica de HTML5 que permite almacenar datos de manera local en el navegador del usuario, sin necesidad de enviarlos a un servidor. Los datos almacenados en LocalStorage persisten incluso después de que el usuario cierra el navegador o apaga el equipo, a menos que sean eliminados explícitamente.

Características clave de LocalStorage

- Almacenamiento persistente: Los datos no se eliminan cuando se cierra el navegador.
- Tamaño limitado: Generalmente, el límite es de 5MB por dominio.
- Datos clave-valor: Los datos se almacenan como pares de clave-valor. Tanto las claves como los valores son cadenas de texto.
- Accesible desde cualquier pestaña: Los datos son compartidos entre diferentes pestañas y ventanas abiertas del mismo navegador y dominio.

Usos comunes de LocalStorage

- Guardar preferencias del usuario, como temas de color o configuraciones de idioma.
- Almacenar temporalmente datos para una aplicación web, como el contenido de un carrito de compras.
- Mantener el estado de una sesión o datos no sensibles.

12.2.1. Métodos de LocalStorage

LocalStorage tiene varios métodos que permiten la manipulación de los datos almacenados. Aquí están los más importantes:

- setItem(key, value)
  Este método se utiliza para almacenar un nuevo par clave-valor en LocalStorage. Si ya existe un valor con la misma clave, se sobrescribirá.

```
localStorage.setItem('clave', 'valor');
```

Ejemplo:

```
// Almacena el nombre de usuario
localStorage.setItem('nombre', 'Juan');
```



- Clave: La cadena que identifica al dato.
- Valor: El dato asociado con la clave (se guarda como una cadena de texto).

- getItem(key)
  Este método devuelve el valor almacenado asociado con la clave proporcionada. Si la clave no existe, devuelve null.

```
// Recupera el nombre de usuario
let nombre = localStorage.getItem('nombre');
console.log(nombre); // "Juan"
```

- removeItem(key)
  Elimina el valor almacenado asociado con la clave especificada.

```
// Elimina el nombre de usuario almacenado
localStorage.removeItem('nombre');
```

- clear()
  Elimina todos los datos almacenados en LocalStorage para el dominio actual.

```
// Elimina todos los elementos de LocalStorage
localStorage.clear();
```

- length
  Devuelve el número de pares clave-valor almacenados en LocalStorage.

```
let cantidad = localStorage.length;
```

- key(index)
  Devuelve la clave almacenada en una posición específica.

```
// Recupera la primera clave almacenada
let primeraClave = localStorage.key(0);
console.log(primeraClave);
```



12.3. SessionStorage

SessionStorage es similar a LocalStorage, pero con una diferencia importante: los datos almacenados en SessionStorage solo persisten durante la sesión actual del navegador. Una vez que el usuario cierra la pestaña o ventana, los datos se eliminan automáticamente.

Características clave de SessionStorage

- Datos temporales: Los datos se mantienen mientras la pestaña esté abierta. Al cerrar la pestaña, los datos se eliminan.
- Tamaño limitado: Similar a LocalStorage, el límite es generalmente de 5MB.
- Datos clave-valor: Al igual que LocalStorage, SessionStorage también usa pares clave-valor.

Usos comunes de SessionStorage

- Almacenar datos temporales que no deben persistir más allá de la sesión actual, como el progreso de un formulario.
- Mantener el estado de la sesión del usuario en una sola pestaña.

12.4. Diferencias entre Cookies, LocalStorage y SessionStorage

Característica	LocalStorage	SessionStorage	Cookies
Persistencia	Persiste hasta que se borra manualmente o con clear()	Persiste solo durante la sesión del navegador	Se puede configurar para persistir o expirar en un tiempo determinado
Tamaño máximo	Generalmente 5MB	Generalmente 5MB	Aproximadamente 4KB por cookie
Ámbito (Scope)	Accesible desde todas las pestañas/ventanas del mismo dominio	Accesible solo en la pestaña donde se creó	Accesibles en el navegador y servidor del dominio (enviadas con cada solicitud)
Intercambio con el servidor	Solo accesible en el navegador	Solo accesible en el navegador	Se envían automáticamente al servidor con cada solicitud HTTP
Seguridad	No se envía al servidor (localmente en el navegador)	No se envía al servidor	Vulnerable si no se configuran correctamente con HTTPS y HttpOnly



Vida útil	Persiste incluso al cerrar el navegador	Se elimina al cerrar la pestaña	Depende de la configuración de expiración (puede ser hasta de sesión o persistente)
Accesibilidad	JavaScript (client-side)	JavaScript (client-side)	JavaScript y Servidor (client-side y server-side)
Uso recomendado	Almacenar datos persistentes del usuario (ej. configuración, tokens no sensibles)	Datos temporales de la sesión (ej. formularios temporales)	Almacenar información pequeña para autenticación y seguimiento de usuario

Conclusiones

- LocalStorage: Ideal para almacenar datos que deben persistir indefinidamente en el navegador, como preferencias del usuario o tokens no sensibles. No se comparte con el servidor, por lo que es más seguro para datos sensibles que no necesitan ser accesibles desde el servidor.
- SessionStorage: Adecuado para almacenar datos temporales que solo son necesarios mientras la pestaña está abierta. Al cerrarse la pestaña o ventana, los datos desaparecen. Es útil para estados temporales o progreso del usuario dentro de la sesión actual.
- Cookies: Son más apropiadas para gestionar información que necesita ser compartida entre el navegador y el servidor, como tokens de autenticación o seguimiento de sesiones. Sin embargo, tienen limitaciones de tamaño y pueden ser menos seguras si no se configuran correctamente (ej. con HttpOnly y Secure).

13. Fundamentos de POO (Programación Orientada a Objetos) en JavaScript

La Programación Orientada a Objetos (POO) es un paradigma de programación que modela la realidad mediante objetos que contienen datos y métodos que manipulan esos datos. Aunque JavaScript no es un lenguaje puramente orientado a objetos como Java o C++, con la introducción de ES6 y las clases, se ha vuelto mucho más amigable para trabajar con este paradigma.



Esta guía te explicará paso a paso los conceptos fundamentales de la POO en JavaScript, profundizando en cómo funciona y proporcionando ejemplos de la vida cotidiana para ayudar a comprender los conceptos clave.

13.1. Conceptos Básicos de la POO

13.1.1 Objetos

Los objetos son las "entidades" en la POO. Un objeto en el mundo real podría ser cualquier cosa con propiedades y comportamientos. Por ejemplo, un carro tiene propiedades como "color", "marca" y "modelo", y comportamientos como "acelerar" o "frenar".

Analogía: Piensa en un objeto como una caja. En esta caja, puedes poner diferentes elementos que representen las propiedades y comportamientos del objeto. Un carro (objeto) puede tener el color rojo, ser de la marca Toyota, y tener la capacidad de acelerar.

Ejemplo en JavaScript:

```
let carro = {
  marca: "Toyota",
  color: "Rojo",
  acelerar: function() {
    console.log("El carro está acelerando.");
  }
};

console.log(carro.marca); // "Toyota"
carro.acelerar(); // "El carro está acelerando."
```

13.1.2 Clases

Una clase es un molde o plantilla a partir del cual se crean objetos. Cada objeto es una instancia de una clase.

Analogía: Si pensamos en "carro" como un concepto general (una clase), entonces cada carro específico (un Toyota rojo, un Honda azul) es una instancia de esa clase.



Ejemplo de una clase en JavaScript:

```
class Carro {
  constructor(marca, color) {
    this.marca = marca;
    this.color = color;
  }

  acelerar() {
    console.log(`El ${this.marca} está acelerando.`);
  }
}

let miCarro = new Carro("Toyota", "Rojo");
miCarro.acelerar(); // "El Toyota está acelerando."
```

13.1.3 constructor

En el paradigma de Programación Orientada a Objetos (POO), un constructor es un método especial que se ejecuta cuando se crea una nueva instancia de una clase. Su propósito principal es inicializar los objetos, es decir, asignarles propiedades y valores.

El constructor se define dentro de una clase usando la palabra clave constructor(). Este método se ejecuta automáticamente cuando se crea una instancia de la clase usando la palabra clave new.



Ejemplo de uso del constructor en una clase:

```
class Carro {
  constructor(marca, color) {
    // Inicializamos las propiedades
    this.marca = marca;
    this.color = color;
  }

  // Método para mostrar detalles del carro
  mostrarDetalles() {
    console.log(`Este carro es un ${this.marca} de color ${this.color}.`);
  }
}

// Crear una nueva instancia de la clase Carro
let miCarro = new Carro("Toyota", "Rojo");
miCarro.mostrarDetalles(); // "Este carro es un Toyota de color Rojo."
```

Detalles importantes

- Obligatoriedad del constructor: Si no defines un constructor en una clase, JavaScript usará uno por defecto. Este constructor por defecto no toma ningún parámetro y no hace nada más allá de crear la instancia.
- Acceso a propiedades con this: Dentro del constructor, this hace referencia a la nueva instancia del objeto que está siendo creada, lo que permite acceder y asignar propiedades.
- Llamar al constructor: Para crear una nueva instancia de una clase, debes usar la palabra clave new, lo que invoca automáticamente el constructor.

Ejemplo sin constructor definido:
Si no se define un constructor explícitamente, JavaScript crea uno implícito.



```
class Perro {
  ladrar() {
    console.log("¡Guau guau!");
  }
}

let miPerro = new Perro(); // El constructor predeterminado es invocado
miPerro.ladrar(); // "¡Guau guau!"
```

13.1.4 Instancias

Una instancia es un objeto creado a partir de una clase. Es como un "ejemplar" único de esa clase.

Ejemplo:

```
let carro1 = new Carro("Honda", "Azul");
let carro2 = new Carro("Ford", "Negro");

console.log(carro1.marca); // "Honda"
console.log(carro2.color); // "Negro"
```

Cada objeto (carro1 y carro2) es una instancia de la clase Carro.

13.1.5. Funciones y Métodos

En el contexto de la Programación Orientada a Objetos (POO), tanto las funciones como los métodos son bloques de código que realizan tareas específicas, pero tienen diferencias en cómo se utilizan y dónde se definen.

- Función
  Una función es un bloque de código independiente que realiza una tarea específica y puede ser llamado desde cualquier lugar del programa. Las funciones no están asociadas a ningún objeto o clase en particular.



- Método
  Un método es una función que está asociada a un objeto o una clase. En otras palabras, es una función que forma parte de la "estructura" del objeto o clase, y actúa sobre los datos de ese objeto (es decir, sus propiedades).

  Ejemplo Método en una clase:

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

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

const persona1 = new Persona('Ana');
persona1.saludar(); // Llama al método y muestra 'Hola, me llamo Ana'
```

  Aquí, saludar() es un **método** de la clase Persona. Está definido dentro de la clase y actúa sobre las propiedades de las instancias creadas a partir de esa clase.

  Diferencias clave entre Método y Función

Característica	Función	Método
Ubicación	Independiente, no está dentro de un objeto o clase	Definido dentro de un objeto o clase
Asociación	No está asociada a ningún objeto o clase	Asociado a un objeto o clase, accede a sus propiedades
Acceso a datos	No tiene acceso a las propiedades de objetos específicos	Usa this para acceder a las propiedades del objeto o clase
Forma de llamada	Se llama directamente por su nombre	Se llama a través de un objeto o instancia



Conclusión:

- Funciones son bloques de código independientes, no están asociadas a objetos o clases, y se llaman por su nombre.
- Métodos son funciones que pertenecen a un objeto o una clase y pueden acceder y manipular las propiedades de ese objeto o instancia usando this.

En POO, los métodos permiten definir comportamientos específicos para los objetos, mientras que las funciones se usan para tareas generales que no necesitan estar vinculadas a una entidad específica

13.2. Principios de la Programación Orientada a Objetos

13.2.1 Encapsulamiento

El encapsulamiento consiste en ocultar los detalles internos de un objeto y exponer solo lo necesario. Las propiedades y métodos que no deben ser accesibles desde fuera se mantienen privados, mientras que los necesarios se hacen públicos.

Analogía: Piensa en una televisión. Sabes cómo encenderla y cambiar el canal (interfaz pública), pero no necesitas saber cómo funcionan sus circuitos internos (detalles privados).

Ejemplo en JavaScript:



```
class CuentaBancaria {
  constructor(saldoInicial) {
    let saldo = saldoInicial;  // Propiedad privada

    this.depositar = function(cantidad) {
      saldo += cantidad;
    }

    this.obtenerSaldo = function() {
      return saldo;
    }
  }
}

let miCuenta = new CuentaBancaria(1000);
miCuenta.depositar(500);
console.log(miCuenta.obtenerSaldo()); // 1500
```

Aquí, saldo es privado y solo puede modificarse mediante los métodos depositar y obtenerSaldo.

13.2.2 Herencia

La herencia permite que una clase "hija" herede las propiedades y métodos de una clase "padre". Esto promueve la reutilización de código.

Analogía: Piensa en una clase "Animal" que tiene características generales como "comer" y "respirar". Una clase "Perro" puede heredar de "Animal" y agregar características específicas como "ladrar".



Ejemplo en JavaScript:

```
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 miPerro = new Perro("Fido");
miPerro.hacerSonido(); // "Fido ladra."
```

La clase Perro hereda de la clase Animal, pero puede sobreescribir el método hacerSonido.

13.2.3 Polimorfismo

El polimorfismo permite que una clase o método adopte múltiples formas. Las clases derivadas pueden modificar el comportamiento de los métodos de la clase base.

Analogía: Un empleado puede tener diferentes responsabilidades según su puesto. Todos son empleados, pero un gerente tiene tareas diferentes a un asistente.



Ejemplo en JavaScript:

```
class Empleado {
  trabajar() {
    console.log("El empleado está trabajando.");
  }
}

class Gerente extends Empleado {
  trabajar() {
    console.log("El gerente está supervisando.");
  }
}

let empleado = new Empleado();
let gerente = new Gerente();

empleado.trabajar(); // "El empleado está trabajando."
gerente.trabajar();  // "El gerente está supervisando."
```

Ambos son empleados, pero el método trabajar varía según el tipo de empleado.

13.2.4 Abstracción

La abstracción consiste en mostrar solo los detalles esenciales y ocultar la implementación. Es un concepto donde solo los aspectos más relevantes de un objeto se revelan al mundo exterior.

Analogía: En un automóvil, solo ves los pedales y el volante para conducir. No ves los detalles del motor y cómo funcionan internamente.

Conclusión

La Programación Orientada a Objetos (POO) en JavaScript es una poderosa herramienta que te permite organizar y estructurar tu código de manera eficiente, promoviendo la reutilización y facilitando el mantenimiento a medida que tus aplicaciones crecen en complejidad.



14. Introducción a la Asincronía en JavaScript

JavaScript es un lenguaje de programación que maneja operaciones de forma asíncrona mediante el uso de callbacks, promesas, y async/await. Esto es crucial para manejar tareas que requieren tiempo, como solicitudes a servidores o lectura de archivos.

Analogía: Imagina que estás en un restaurante. Pides una comida (una operación que tomará tiempo). En lugar de esperar sentado sin hacer nada, puedes hacer otras cosas mientras el chef prepara tu comida. De manera similar, JavaScript usa técnicas asíncronas para manejar tareas mientras sigue ejecutando otras operaciones.

14.1. Callback

Un callback es una función pasada como argumento a otra función, que luego se ejecuta después de que se completa una tarea.

Analogía: Pide una pizza por teléfono y te dicen que te llamarán cuando esté lista. El teléfono que te llama es el callback; ejecutará una función (llamarte) después de que la pizza esté lista.

Ejemplo de Callback:

```
function fetchData(callback) {
  setTimeout(() => {
    // Simula la obtención de datos después de 2 segundos
    callback('Datos obtenidos');
  }, 2000);
}

fetchData(data => {
  console.log(data); // 'Datos obtenidos'
});
```

Ventajas y Desventajas de los Callbacks

- Ventajas:
    - Simples y directos para tareas simples.



- Desventajas:
    - Puede llevar a lo que se llama "callback hell" o "pyramid of doom", donde el código se vuelve difícil de leer y mantener.

14.2. Promise

Una promesa es un objeto que representa el eventual resultado o error de una operación asíncrona. Permite manejar resultados y errores de manera más organizada que los callbacks.

Analogía: Imagina que compras una película en preventa. Te dan un vale (la promesa) que podrás cambiar por la película en el futuro. Cuando vas a recogerla (resolución de la promesa), puedes usar el vale para obtener la película.

Ejemplo de Promesa:

```
let promise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('Datos obtenidos'); // O reject('Error') si algo sale mal
  }, 2000);
});

promise.then(data => {
  console.log(data); // 'Datos obtenidos'
}).catch(error => {
  console.error(error);
});
```

Encadenamiento de Promesas

Puedes encadenar varias promesas para realizar tareas en secuencia.



```
fetchData()
  .then(data => processData(data))
  .then(result => console.log(result))
  .catch(error => console.error('Error:', error));
```

14.3. Async/Await

async/await es una forma más moderna y legible de manejar operaciones asíncronas que utiliza promesas bajo el capó.

Analogía: En lugar de hacer cola para el nuevo disco que quieres, ahora puedes hacer una reserva y continuar con otras cosas mientras esperas. Cuando el disco esté disponible, puedes recogerlo sin hacer una cola complicada.

Sintaxis

- async: Marca una función como asíncrona, lo que permite usar await dentro de ella.
- await: Pausa la ejecución de la función hasta que la promesa se resuelva.

```
async function getData() {
  try {
    let response = await fetch('https://jsonplaceholder.typicode.com/posts/1');
    if (!response.ok) {
      throw new Error('Error en la solicitud');
    }
    let data = await response.json();
    console.log(data);
  } catch (error) {
    console.error('Error:', error);
  }
}

getData();
```

Ventajas de async/await

- Código más limpio y legible: Evita el anidamiento de promesas y callbacks.



- Manejo de errores simplificado: Utiliza try/catch para manejar errores en operaciones asíncronas.

14.4. fetch()

fetch() es una función nativa de JavaScript utilizada para hacer solicitudes HTTP de forma asíncrona. Permite interactuar con recursos a través de la web (APIs) para obtener, enviar, modificar o eliminar datos de un servidor sin necesidad de recargar la página.

Analogía: Piensa en fetch() como en un mensajero. Tú le das un encargo (la solicitud HTTP) y, mientras el mensajero va a recoger la información, tú puedes seguir haciendo otras tareas. Cuando el mensajero regresa, puedes revisar los datos que trajo (la respuesta HTTP).

¿Para qué se usa fetch()?

- Obtener datos de un servidor o una API.
- Enviar datos a un servidor (por ejemplo, enviar información de un formulario).
- Manipular datos en un servidor mediante solicitudes HTTP como GET, POST, PUT, DELETE.

Sintaxis

La función fetch() toma como argumento una URL y devuelve una promesa que se resuelve en la respuesta de la solicitud. La sintaxis básica es:

```
fetch(url)
  .then(response => {
    // Procesar la respuesta
  })
  .catch(error => {
    // Manejar el error
  });
```



```
fetch('https://jsonplaceholder.typicode.com/posts/1')
  .then(response => response.json()) // Convertir la respuesta en formato JSON
  .then(data => console.log(data)) // Imprimir los datos en la consola
  .catch(error => console.error('Error:', error)); // Manejar errores
```

¿Cómo funciona fetch()?

Fases de una solicitud con fetch():

- Realizar la solicitud: Llamas a fetch() con la URL del recurso.
- Esperar la respuesta: fetch() devuelve una promesa que se resuelve cuando la solicitud completa el proceso.
- Leer los datos: Puedes convertir la respuesta en varios formatos, como JSON, texto, o Blob.
- Manejar errores: Si la solicitud falla (problemas de red, servidor no disponible), se captura el error.

14.4.1 Métodos HTTP

Los métodos HTTP son un conjunto de acciones que permiten la comunicación entre un cliente (por ejemplo, un navegador web) y un servidor web. Estos métodos indican qué tipo de operación se desea realizar en un recurso determinado (como obtener información, enviar datos, actualizar contenido, etc.). Cada método tiene un propósito específico y es fundamental para interactuar con APIs y servidores web.

Cuando haces una solicitud HTTP, puedes usar diferentes métodos, como:

- GET: Para obtener datos.
- POST: Para enviar datos.
- PUT: Para actualizar datos.
- DELETE: Para eliminar datos.
- PATCH: Actualizar parcialmente un recurso.

Los métodos HTTP permiten la implementación de una arquitectura basada en recursos, donde cada recurso en un servidor tiene su propia URL. Los diferentes métodos permiten manipular estos recursos de manera segura, eficiente y controlada.

Por ejemplo, cuando visitas una página web, el navegador envía una solicitud GET al servidor para recuperar la información de la página. Si te registras en una aplicación,



tu navegador probablemente utilizará POST para enviar tus datos de registro al servidor.

Ejemplo de solicitud GET

```
fetch('https://jsonplaceholder.typicode.com/posts')
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error('Error:', error));
```

Ejemplo de solicitud POST

```
fetch('https://jsonplaceholder.typicode.com/posts', {
  method: 'POST', // Método POST
  headers: {
    'Content-Type': 'application/json' // Cabecera para enviar datos en formato JSON
  },
  body: JSON.stringify({
    title: 'Nuevo Post',
    body: 'Este es el contenido del nuevo post',
    userId: 1
  })
})
.then(response => response.json())
.then(data => console.log(data))
.catch(error => console.error('Error:', error));
```

Explicación del ejemplo:

- Usamos POST para enviar un nuevo post al servidor.
- **headers**: Definimos las cabeceras de la solicitud, indicando que estamos enviando datos en formato JSON.
- **body**: Contiene los datos a enviar en el cuerpo de la solicitud.

14.4.2 Convertir la respuesta

fetch() devuelve una promesa que contiene un objeto Response. Puedes convertir la respuesta al formato que necesites, como JSON, texto o Blob.

- **response.json()**: Convierte la respuesta en un objeto JavaScript.
- **response.text()**: Convierte la respuesta en texto.



- response.blob(): Convierte la respuesta en un objeto binario (por ejemplo, para manejar archivos).

Ejemplo con JSON:

```
fetch('https://jsonplaceholder.typicode.com/posts/1')
  .then(response => response.json()) // Convertimos la respuesta a JSON
  .then(data => console.log(data)) // Imprimimos los datos
  .catch(error => console.error('Error:', error));
```

fetch() no lanza un error automáticamente si la respuesta HTTP tiene un código de error (por ejemplo, 404 o 500). Aun así, debes comprobar el estado de la respuesta y manejar los errores manualmente.

Manejo de errores con fetch()

```
fetch('https://jsonplaceholder.typicode.com/invalid-url')
.then(response => {
    if (!response.ok) {
    throw new Error('Error en la solicitud: ' + response.status); // Lanza un error si la respuesta no es correcta
    }
    return response.json();
})
.then(data => console.log(data))
.catch(error => console.error('Error capturado:', error)); // Captura y maneja el error
```

Explicación:

- response.ok: Esta propiedad indica si la solicitud fue exitosa (código de estado HTTP entre 200-299).
- Si ok es false, lanzamos un error manualmente.

Ventajas de usar fetch()

- Basado en Promesas: Es fácil de usar junto con async/await para escribir código asíncrono más legible.
- Soporte nativo: No requiere bibliotecas externas como jQuery.
- Manejo sencillo de respuestas JSON: Es ideal para trabajar con APIs RESTful que devuelven datos en formato JSON.

Desventajas de fetch()

- No maneja errores de red automáticamente: Aunque fetch() detecta errores de red, no lanza un error por códigos de respuesta HTTP no exitosos (como 404 o 500). Debes manejarlos manualmente.



- Soporte limitado en navegadores antiguos: En algunos navegadores antiguos, es posible que necesites usar un *polyflll* (como whatwg-fetch) para garantizar la compatibilidad.

14.4.3 Uso de async/await con fetch()

Para hacer el código más legible, puedes usar async/await en lugar de .then().

Ejemplo con async/await

```
async function obtenerDatos() {
  try {
    let respuesta = await fetch('https://jsonplaceholder.typicode.com/posts/1');

    if (!respuesta.ok) {
      throw new Error('Error en la solicitud');
    }

    let datos = await respuesta.json();
    console.log(datos);
  } catch (error) {
    console.error('Error:', error);
  }
}

obtenerDatos();
```

Explicación:

- Usamos await para esperar a que la promesa de fetch() se resuelva.
- En caso de error, lanzamos una excepción con throw y la capturamos en el bloque catch.

fetch() es una herramienta poderosa y flexible para hacer solicitudes HTTP en JavaScript. Junto con promesas y async/await, facilita el manejo de datos asíncronos, especialmente cuando trabajamos con APIs. Sin embargo, es importante tener en cuenta sus limitaciones, como el manejo de errores manual, para evitar comportamientos inesperados.



14.5 Códigos de estado HTTP

Los **códigos de estado HTTP** son respuestas que los servidores web envían al cliente (normalmente el navegador o una aplicación) cuando éste hace una solicitud a un recurso. Estos códigos indican el estado de la solicitud: si fue exitosa, si hubo un error o si se necesita alguna acción adicional.

Los códigos de estado HTTP están divididos en **cinco clases principales** que van del 1xx al 5xx, donde cada clase representa un tipo diferente de respuesta.

14.5.1 Respuestas informativas (1xx)

Estos códigos indican que la solicitud ha sido recibida y el proceso continúa.

- **100 Continue**: El servidor ha recibido la solicitud inicial y el cliente puede continuar con la misma.
- **101 Switching Protocols**: El servidor acepta cambiar el protocolo, solicitado por el cliente.

14.5.2. Respuestas satisfactorias (2xx)

Estos códigos indican que la solicitud fue recibida, entendida y aceptada correctamente.

- **200 OK**: La solicitud ha sido exitosa y el servidor ha enviado los datos solicitados.
- **201 Created**: La solicitud fue exitosa y se ha creado un nuevo recurso como resultado.
- **202 Accepted**: La solicitud ha sido aceptada, pero el procesamiento no se ha completado aún. Esto es común en operaciones asíncronas.
- **204 No Content**: La solicitud fue exitosa, pero no hay contenido que devolver.

14.5.3 Redirecciones (3xx)

Estos códigos indican que es necesario tomar más acciones para completar la solicitud, normalmente una redirección a otra URL.

- **301 Moved Permanently**: La URL solicitada ha sido cambiada permanentemente. El cliente debe usar la nueva URL en el futuro.
- **302 Found**: La URL solicitada ha sido temporalmente movida a una nueva ubicación.
- **304 Not Modified**: El recurso no ha cambiado desde la última solicitud del cliente, por lo que puede usar una copia almacenada en caché.



14.5.4. Errores del cliente (4xx)

Estos códigos indican que hubo un error en la solicitud enviada por el cliente.

- 400 Bad Request: El servidor no pudo entender la solicitud debido a una sintaxis incorrecta.
- 401 Unauthorized: La solicitud requiere autenticación y el cliente no ha proporcionado credenciales válidas.
- 403 Forbidden: El servidor entiende la solicitud, pero se niega a autorizarla.
- 404 Not Found: El servidor no pudo encontrar el recurso solicitado.
- 405 Method Not Allowed: El método HTTP utilizado no está permitido para el recurso solicitado.
- 408 Request Timeout: El cliente tardó demasiado en enviar la solicitud completa y el servidor cortó la conexión.

14.5.5. Errores del servidor (5xx)

Estos códigos indican que el servidor falló al procesar una solicitud válida.

- 500 Internal Server Error: El servidor encontró un error inesperado que le impidió cumplir con la solicitud.
- 501 Not Implemented: El servidor no tiene la funcionalidad necesaria para cumplir con la solicitud.
- 502 Bad Gateway: El servidor, actuando como un gateway o proxy, recibió una respuesta inválida del servidor aguas arriba.
- 503 Service Unavailable: El servidor no está disponible temporalmente, generalmente debido a sobrecarga o mantenimiento.
- 504 Gateway Timeout: El servidor no recibió una respuesta oportuna del servidor aguas arriba.

Resumen

- 1xx: Respuestas informativas (proceso en marcha).
- 2xx: Respuestas exitosas (la solicitud fue procesada correctamente).
- 3xx: Redirecciones (se requiere acción adicional por parte del cliente).
- 4xx: Errores del cliente (problemas con la solicitud).
- 5xx: Errores del servidor (el servidor no pudo completar la solicitud).

¿Por qué son importantes los Códigos de Estado HTTP?

- Depuración: Ayudan a desarrolladores y administradores de sistemas a identificar problemas con las solicitudes HTTP. Por ejemplo, un error 500 Internal Server Error puede indicar un fallo en el servidor, mientras que un 404 Not Found podría significar que la URL solicitada está mal escrita.



- SEO: En el contexto de optimización para motores de búsqueda, redirecciones adecuadas (301 Moved Permanently) pueden asegurar que los motores de búsqueda indexen correctamente tu contenido.
- Experiencia de usuario: Los códigos de estado permiten a los desarrolladores manejar errores de manera apropiada, mostrando mensajes de error claros o redirigiendo a los usuarios cuando sea necesario.

15. Manejo de Errores en JavaScript

El manejo de errores en JavaScript es esencial para crear aplicaciones robustas que puedan recuperarse de situaciones inesperadas sin fallar. JavaScript proporciona mecanismos para detectar, capturar y manejar errores mediante las estructuras try...catch...finally y el uso de la palabra clave throw.

15.1 Estructura try...catch...finally

La estructura try...catch...finally se utiliza para manejar errores que pueden ocurrir durante la ejecución de un bloque de código. El bloque try contiene el código que puede producir un error, el bloque catch maneja el error si ocurre, y el bloque finally contiene el código que siempre se ejecuta, sin importar si hubo un error o no.

Sintaxis:

```
try {
  // Bloque de código que puede generar errores
} catch (error) {
  // Bloque de código que maneja el error
} finally {
  // Código que siempre se ejecuta, ocurra o no el error
}
```

Componentes:

- try: Aquí colocas el código que deseas monitorear en busca de errores.
- catch: Si ocurre un error dentro del bloque try, este código se ejecuta para manejarlo. El parámetro error puede contener detalles sobre el error.
- finally: Este bloque se ejecuta siempre, sin importar si hubo un error o no, lo que es útil para limpiar recursos o finalizar tareas.



Ejemplo:

```
try {
  let resultado = 10 / 0; // Esto no genera error, pero cualquier otra operación ilegal
  console.log("Resultado:", resultado);
} catch (error) {
  console.log("Ha ocurrido un error:", error.message);
} finally {
  console.log("Este mensaje se mostrará siempre.");
}
```

Funcionamiento:

- El bloque try intenta ejecutar el código.
- Si no ocurre un error, se salta al bloque finally (si existe).
- Si ocurre un error, el flujo se transfiere al bloque catch, donde puedes manejar el error.
- El bloque finally se ejecuta en cualquier caso (haya o no error).

Ejemplo:
Imagina que tienes una aplicación de compras en línea que intenta procesar un pago. Si ocurre un error (como una conexión fallida), el bloque catch podría notificar al usuario y el bloque finally podría limpiar datos temporales.

```
try {
  // Intentamos procesar el pago
  procesarPago();
} catch (error) {
  console.log("Error procesando el pago:", error.message);
  notificarUsuario("Hubo un problema con tu pago, intenta de nuevo.");
} finally {
  limpiarDatosTemporales();
}
```



15.2 Lanzamiento de Errores con throw

El operador throw se utiliza para lanzar errores de manera manual. Esto es útil cuando deseas que tu código genere un error específico si se encuentra con una condición inválida o inesperada.

Sintaxis:

```
throw new Error("Mensaje de error personalizado");
```

Puedes lanzar **cualquier tipo de valor** (número, cadena, objeto), pero es recomendable lanzar instancias de Error, ya que proveen información adicional sobre el problema.

Ejemplo con throw:

```
function verificarEdad(edad) {
  if (edad < 18) {
    throw new Error("Debes ser mayor de 18 años.");
  } else {
    console.log("Acceso permitido.");
  }
}

try {
  verificarEdad(16);  // Esto lanzará un error
} catch (error) {
  console.log(error.message);  // "Debes ser mayor de 18 años."
}
```

Funcionamiento:

- Si el código dentro de la función encuentra una situación inesperada, como una edad menor a 18, se "lanza" un error con throw.
- Este error interrumpe la ejecución normal del programa y pasa el control al bloque catch, donde puede ser manejado.



Tipos de errores personalizados

En lugar de solo lanzar instancias de Error, también puedes crear errores más específicos, como RangeError, ReferenceError, o cualquier otro que ayude a describir mejor el problema.

```
function dividir(a, b) {
  if (b === 0) {
    throw new RangeError("No se puede dividir por cero.");
  }
  return a / b;
}

try {
  dividir(10, 0);
} catch (error) {
  console.log(error.name + ": " + error.message);  // RangeError: No se puede dividir por
}
```

Combinación de try...catch y throw

Una práctica común es utilizar throw para lanzar errores dentro del bloque try, y luego manejar esos errores en el bloque catch.

Ejemplo:

```
function validarNombre(nombre) {
  if (!nombre) {
    throw new Error("El nombre no puede estar vacío.");
  }
}

try {
  validarNombre("");  // Lanzará un error
} catch (error) {
  console.log("Error:", error.message);  // "El nombre no puede estar vacío."
}
```

Conclusiones:

- try...catch...finally te permite capturar y manejar errores de manera elegante.
- throw te permite lanzar tus propios errores personalizados cuando ocurre una condición inválida.



- El uso adecuado de estas estructuras mejora la **robustez** y **fiabilidad** de tu aplicación, permitiendo que maneje errores de forma controlada en lugar de fallar inesperadamente.

Este es un aspecto fundamental del desarrollo en JavaScript, especialmente cuando trabajas con APIs, procesamiento de datos o cualquier situación en la que los errores son comunes.

16. Interactuando con APIs (Ejemplo con la API de Rick y Morty)

En este punto detallaremos cómo interactuar con APIs utilizando JavaScript, haciendo énfasis en las herramientas modernas como fetch(), promesas (promises), async/await y manejo de errores.

¿Qué es una API?

Una API (Application Programming Interface) es una interfaz que permite la comunicación entre diferentes aplicaciones. En el contexto de una API REST, estas son puntos de acceso (endpoints) a los que puedes hacer solicitudes HTTP (GET, POST, PUT, DELETE) para interactuar con datos o servicios.

API de Rick y Morty

Usaremos la API de Rick y Morty como ejemplo. Esta API es pública y nos permitirá obtener datos sobre los personajes de la serie.

El endpoint básico para obtener personajes es:
https://rickandmortyapi.com/api/character

16.1. Uso de fetch() para realizar solicitudes

La función fetch() es la forma moderna de realizar solicitudes HTTP en JavaScript. Es una función asíncrona que devuelve una promesa.



```
fetch('https://rickandmortyapi.com/api/character')
.then(response => {
    // Verifica si la respuesta es exitosa
    if (!response.ok) {
    throw new Error('Error en la solicitud');
    }
    return response.json();  // Convierte la respuesta a JSON
})
.then(data => {
    console.log('Datos recibidos:', data);
    // Trabajamos con los datos recibidos, como una lista de personajes
    const characters = data.results;
    characters.forEach(character => {
    console.log(`${character.name} - ${character.species}`);
    });
})
.catch(error => {
    console.error('Error:', error);
});
```

¿Qué hace este código?

- fetch realiza una solicitud a la API de Rick y Morty.
- response.json() convierte la respuesta a formato JSON.
- data.results contiene una lista de personajes.
- En caso de error, lo capturamos con catch.

16.2. Uso de async/await para manejo de promesas

async/await es una forma más moderna y legible de manejar promesas. Convierte el código basado en promesas en un estilo más secuencial y fácil de entender.



```javascript
async function obtenerPersonajes() {
    try {
        const response = await fetch('https://rickandmortyapi.com/api/character');

        // Verificamos si la respuesta es exitosa
        if (!response.ok) {
        throw new Error('Error en la solicitud');
        }

        const data = await response.json();  // Convertimos la respuesta a JSON
        console.log('Datos recibidos:', data);

        // Mostramos los nombres de los personajes
        data.results.forEach(character => {
        console.log(`${character.name} - ${character.species}`);
        });
    } catch (error) {
        console.error('Error:', error);
    }
}

obtenerPersonajes();
```

¿Qué hace este código?

- La función obtenerPersonajes() es declarada como **async**.
- **await** espera la respuesta de fetch().
- Si hay un error en la solicitud, lo manejamos con un bloque try/catch.

16.3. Manejo de Errores en fetch

El manejo de errores es esencial para evitar que tu aplicación falle de forma inesperada. En fetch(), podemos verificar si el servidor ha respondido correctamente usando response.ok.



```
async function obtenerPersonajePorId(id) {
    try {
        const response = await fetch(`https://rickandmortyapi.com/api/character/${id}`);

        if (!response.ok) {
        throw new Error(`Error: ${response.status} - ${response.statusText}`);
        }

        const data = await response.json();
        console.log(`${data.name} - ${data.species}`);
    } catch (error) {
        console.error('Error al obtener el personaje:', error);
    }
}

obtenerPersonajePorId(1);  // Llamada a la función con un ID
```

¿Qué hace este código?

- Busca un personaje por su ID en la API.
- Si la solicitud no es exitosa, lanza un error con el código de estado HTTP.

16.4. Gestión de Respuestas en JSON

Al consumir APIs, la respuesta suele venir en formato JSON, que es una estructura de datos que facilita la lectura y escritura de datos estructurados.

- JSON.stringify() convierte un objeto JavaScript en una cadena JSON.
- JSON.parse() convierte una cadena JSON en un objeto JavaScript.

Ejemplo de conversión de JSON:

```
const personaje = {
nombre: "Rick Sanchez",
especie: "Humano"
};

// Convertir el objeto a formato JSON
const jsonPersonaje = JSON.stringify(personaje);
console.log(jsonPersonaje);  // {"nombre":"Rick Sanchez","especie":"Humano"}

// Convertir la cadena JSON de nuevo a objeto
const objetoPersonaje = JSON.parse(jsonPersonaje);
console.log(objetoPersonaje.nombre);  // Rick Sanchez
```



16.5. Mostrar Personajes en una Página HTML

Aquí un ejemplo práctico de cómo obtener los personajes y mostrarlos en una lista en HTML.

HTML básico:

```html
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Personajes de Rick y Morty</title>
</head>
<body>
  <h1>Lista de Personajes</h1>
  <ul id="listaPersonajes"></ul>

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

JavaScript (app.js):

```javascript
async function mostrarPersonajes() {
    try {
        const response = await fetch('https://rickandmortyapi.com/api/character');
        const data = await response.json();

        const lista = document.getElementById('listaPersonajes');
        data.results.forEach(character => {
        const li = document.createElement('li');
        li.textContent = `${character.name} - ${character.species}`;
        lista.appendChild(li);
        });
    } catch (error) {
        console.error('Error:', error);
    }
}

mostrarPersonajes();
```

Conclusión

El consumo de APIs es una parte fundamental del desarrollo web moderno, y JavaScript ofrece herramientas como fetch(), promesas y async/await para hacer este proceso sencillo y manejable.