



JavaScript **es un lenguaje de programación ampliamente utilizado que se ejecuta principalmente en navegadores web**, aunque también se puede encontrar en otros entornos como servidores (Node.js) o aplicaciones móviles. Fue creado por Brendan Eich en 1995 y ha evolucionado significativamente desde entonces.

## SITIOS WEB Y APLICACIONES WEB

Mientras que los **sitios web buscan brindar información estática**, las web apps permiten a los usuarios realizar múltiples tareas.

**Las aplicaciones web son plataformas dinámicas e interactivas y sus funcionalidades están en constante mantenimiento y mejora.**

Plataformas como Mercado Libre, Youtube, Gmail, Facebook, Coder-House son web apps por la cantidad de funcionalidades que ofrecen.

En la actualidad, el desarrollo desde cero de sitios web estáticos, es decir, aquellos cuya información no cambia en respuesta a las acciones del usuario, es poco frecuente.

Lo que se busca construir en el ámbito laboral, son plataformas que ofrezcan un alto nivel de interactividad, y un variado número de funcionalidades. Ya no hablamos de sitios, sino de aplicaciones web que permiten realizar tareas importantes a sus usuarios.

Por funcionalidades entendemos diversas tareas que los usuarios y clientes pueden realizar y son típicamente demandadas hoy.

La interactividad en la página es una de ellas. Por ejemplo, implementar animaciones y transiciones complejas, respuestas a ciertos eventos de los usuarios (como clickear un botón), o capturar y enviar datos mediante formularios.

Es normal también consumir alguna API o servicio de backend y/o base de datos, con la cual podemos cargar y administrar la información de la página.

## CARACTERÍSTICAS CLAVE DE JAVASCRIPT:

1. **Lenguaje de Scripting para la Web:** Originalmente, JavaScript fue diseñado para hacer páginas web más interactivas y dinámicas. Se ejecuta en el lado del cliente (en el navegador del usuario) y puede manipular el contenido, la estructura y el estilo de una página web.
2. **Event-Driven y Asíncrono:** JavaScript es un lenguaje orientado a eventos, lo que significa que puede responder a acciones del usuario como clics, entradas del teclado, etc. Además, admite operaciones asíncronas, lo que lo hace muy útil para desarrollar

aplicaciones web que necesitan realizar tareas como cargar datos sin recargar la página completa.

3. **Basado en Objetos:** Aunque JavaScript no es un lenguaje orientado a objetos en el sentido tradicional, utiliza conceptos de objetos. Los objetos en JavaScript pueden ser muy flexibles y se utilizan para almacenar colecciones de datos y unidades de funcionalidad más complejas.
4. **Interpretado y Dinámico:** JavaScript es un lenguaje interpretado, lo que significa que el código se ejecuta línea por línea y se ejecuta por medio de un programa intérprete (en este caso los navegadores son los programas que se encargan de interpretar y ejecutar el código JavaScript que creemos). Es un lenguaje de tipo dinámico, lo que implica que las variables no necesitan tener tipos definidos previamente y pueden cambiar de tipo.
5. **Compatibilidad con Navegadores:** Todos los navegadores web modernos tienen un motor de JavaScript integrado para ejecutarlo. Esto significa que los scripts de JavaScript escritos para una página web generalmente funcionarán en todos los navegadores, aunque pueden existir diferencias en cómo se manejan ciertas características.
6. **ECMAScript:** El estándar que define el lenguaje JavaScript se llama ECMAScript. Ha habido varias versiones de ECMAScript a lo largo de los años, añadiendo nuevas características y mejoras al lenguaje.
7. **Uso en el Lado del Servidor:** Con la introducción de Node.js, JavaScript se ha expandido más allá de los navegadores y se puede usar en el lado del servidor para construir aplicaciones y servicios backend.
8. **Comunidad y Ecosistema:** JavaScript tiene una comunidad muy activa y un ecosistema en constante crecimiento. Existen numerosas bibliotecas y frameworks (como React, Angular y Vue) que facilitan el desarrollo de aplicaciones complejas.

En resumen, JavaScript es un lenguaje de programación esencial para el desarrollo web y ha crecido para abarcar desarrollo de servidores, aplicaciones móviles y más, convirtiéndose en uno de los lenguajes de programación más populares y versátiles en el mundo de la tecnología.

## FRONT-END Y BACK-END

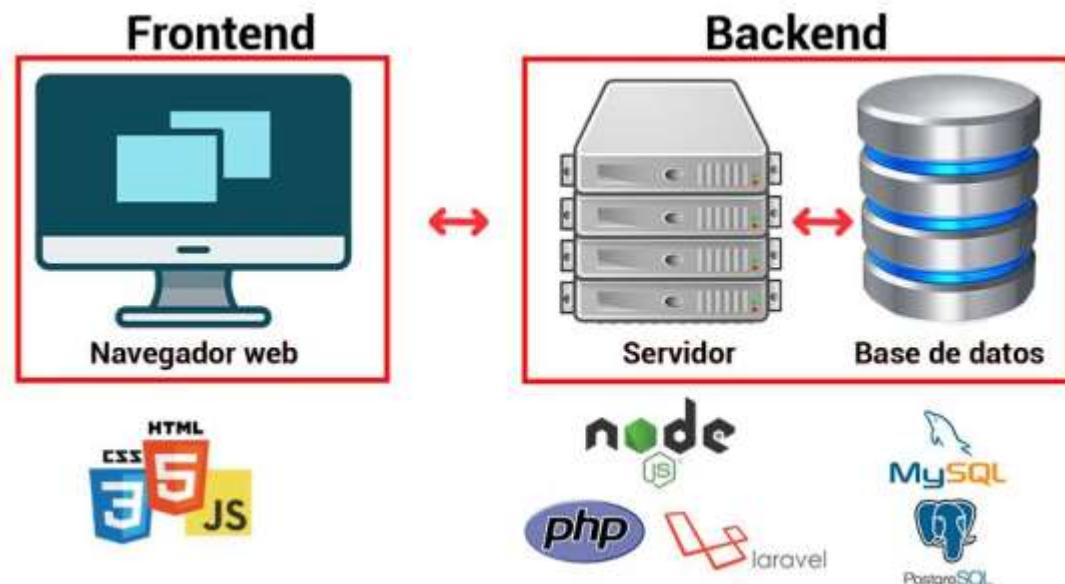
Javascript se utiliza tanto para construir aplicaciones de Frontend como de Backend.

Por Frontend entendemos a la parte de la aplicación que corre en el navegador y con la cual interactúan los usuarios.

Como tal, estaremos creando aplicaciones con JavaScript, HTML y CSS; vinculando los tres lenguajes en el desarrollo un único producto.

Nuestra aplicación de Frontend también consume datos y servicios ofrecidos por algún Backend.

JavaScript será la herramienta que nos permitirá comunicarnos e intercambiar información con APIs u otras aplicaciones.



## CÓDIGO JAVASCRIPT

Nuestro código JavaScript se asocia al archivo HTML que se carga en el navegador. **Tenemos dos maneras de escribir código JavaScript en nuestras aplicaciones web.**

a. **Dentro de un archivo html, entre medio de las etiquetas <script>**

```
<script>
    // Aquí se escribe el código JS
</script>
```

b. **En un archivo individual con extensión .js**

Ejemplo: mi-archivo.js

Se vincula con la etiqueta <script> y el atributo “src”

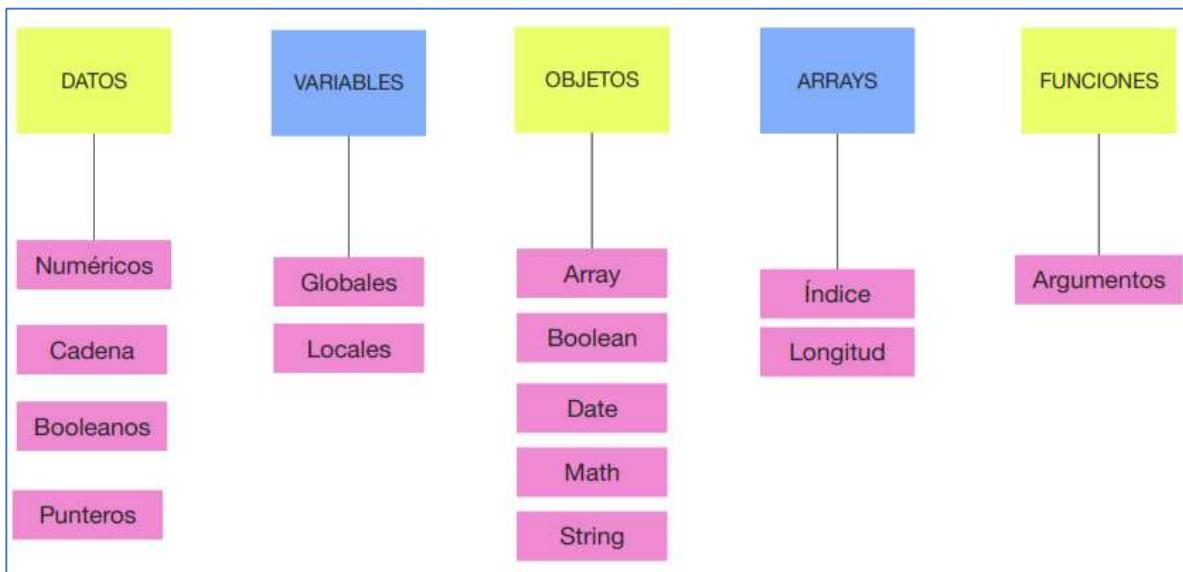
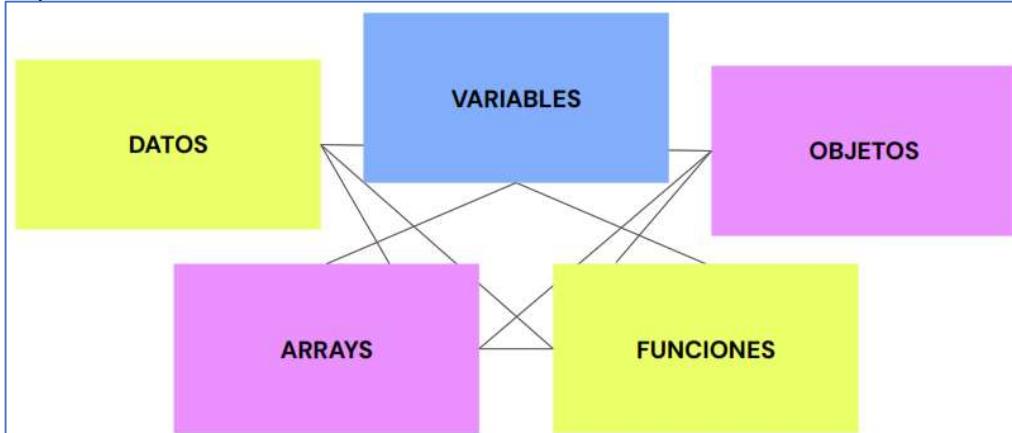
Recuerda no utilizar espacios ni mayúsculas en los nombres de archivo.

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

A este archivo lo tengo que relacionar dentro del código html (en la etiqueta head) y esto lo hacemos con la propia etiqueta script utilizando también el atributo defer para que el navegador primero lea (cargue) todo el código html del documento y luego lea (cargue) el código JS. Luego se pone el atributo src (source) el cual indica cual es la fuente del código que queremos cargar.

## ELEMENTOS BÁSICOS EN JAVASCRIPT

Para trabajar en y con Javascript existen ciertos elementos básicos e imprescindibles.



## IDENTIFICADORES EN JAVASCRIPT

Los identificadores en JavaScript **son nombres utilizados para identificar variables, funciones, clases, y otros elementos en el código**. Son fundamentales en la programación porque permiten referenciar y manipular datos. Aquí hay algunos puntos clave sobre los identificadores en JavaScript:

### Reglas de Nomenclatura

**Deben empezar con una letra (A-Z, a-z), un guión bajo (\_) o un signo de dólar (\$).**

**Los caracteres siguientes** pueden ser letras, números (0-9), guiones bajos o signos de dólar.

**No pueden empezar con un número.**

**No pueden ser palabras reservadas** en JavaScript

```

A: abstract
B: boolean, break, byte
C: case, catch, char, class, const, continue
D: debugger, default, delete, do, double
E: else, enum, export, extends
F: false, final, finally, float, for, function
G: goto
I: if, implements, import, in, instanceof, int, interface
L: let, long
N: native, new, null
P: package, private, protected, public
R: return
S: short, static, super, switch, synchronized
T: this, throw, throws, transient, true, try, typeof
V: var, volatile, void
W: while, with

```

**Sensibilidad a Mayúsculas y Minúsculas: JavaScript distingue entre mayúsculas y minúsculas**, lo que significa que miVariable, MiVariable y mivariable serían considerados identificadores diferentes.

## CONVENCIONES DE ESTILO (NO SON OBLIGATORIAS → BUENAS PRÁCTICAS)

### **camelCase:**

- Variables

```
let contadorDeItems = 0;
```

- Funciones

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

- Objetos

```
let miCarro = {
    marca: 'Toyota',
    modelo: 'Corolla',
    año: 2021
};
```

- Instancias de clases

```
let miCarro = new Carro('Toyota', 'Corolla', 2021);
```

**PascalCase:**

- clases

```
class CarroControl {
    constructor(marca, modelo, color) {
        this.marca = marca;
        this.modelo = modelo;
        this.color = color;
        this.encendido = false;
    }
}
```

**UPPER\_CASE:**

- constantes

```
const MAX_USUARIOS = 100;
const URL_API = "https://miapi.com";
const PI = 3.14159;
```

**snake\_case:**

- Nombre de archivos

```
mi_archivo_javascript.js
```

**Los identificadores deben ser descriptivos para que el código sea fácil de leer y mantener.**

Evitar nombres genéricos como x o data a menos que su uso sea muy claro en el contexto específico.

## Comentarios

```
<script>
    // Comentario simple: una línea
    /* Comentario de más de una línea I
       Comentario de más de una línea II */
</script>
```

# ORDENAMIENTO DEL CÓDIGO

1. **Importaciones de Módulos Externos:** Comienza importando módulos de bibliotecas externas o de terceros. Estos generalmente proporcionan funcionalidades base que tu código necesita.

```
import React from 'react';
import { useState } from 'react'
import express from 'express';
```

2. **Importaciones de Módulos Internos:** A continuación, importa los módulos que son parte de tu aplicación o proyecto pero que están definidos en otros archivos.

```
import './estilos/main.css';
import logo from './imagenes/logo.png';
```

3. **Importaciones de Estilos y Assets:** Luego, si estás trabajando en un proyecto que incluye frontend, como una aplicación React, importa hojas de estilo y otros recursos como imágenes.

```
import './estilos/main.css';
import logo from './imagenes/logo.png';
```

4. **Declaración de Variables y Constantes:** Despues de las importaciones, declara cualquier variable o constante que tu archivo necesite.

```
const app = express();
let contador = 0;
```

5. **Definición de Funciones y Clases:** Posteriormente, define las funciones y clases que serán utilizadas en el archivo.

6. **Ejecución del Código:** Finalmente, coloca el código que se ejecutará, como llamadas a funciones, manipulación del DOM, etc.

# TIPOS DE DATOS

En JavaScript, los tipos de datos se dividen en dos categorías principales: **primitivos y compuestos (u objetos)**:

## Tipos Primitivos

Los tipos de datos primitivos **son inmutables** y se almacenan directamente en la ubicación de la variable, es decir que se accede directamente al valor.

Cuando realizas una **operación en un valor primitivo, como modificar una cadena de texto, lo que realmente sucede es la creación de un nuevo valor primitivo**. El valor original no cambia. Por ejemplo:

```
let cadena = "Hola";
cadena = cadena + " Mundo";
```

En este caso, la cadena original "Hola" no se modifica. En su lugar, se crea una nueva cadena "Hola Mundo" y se asigna a la variable cadena. La cadena original permanece inalterada en cualquier otro lugar donde se haya referenciado previamente.

Los datos primitivos **no tienen métodos**. Los principales tipos primitivos en JavaScript son:

**Number:** Representa tanto enteros como flotantes.

```
let edad = 25; // Entero
let precio = 99.99; // Flotante
```

**String:** Representa secuencias de caracteres y se utiliza para trabajar con texto.

```
let nombre = "Alice";
```

**Boolean:** Representa un valor verdadero o falso.

```
let esVisible = true;
```

**Undefined:** Indica una variable no inicializada o un valor ausente.

```
let indefinido;
```

**Null:** Representa intencionalmente un valor "nulo" o "vacío".

```
let vacio = null;
```

**Symbol (introducido en ES6):** Representa un identificador único.

```
let simbolo = Symbol("id");
console.log(simbolo)
console.log(typeof(simbolo))
```

Symbol(id)
symbol

**BigInt (introducido en ES11):** Permite representar enteros muy grandes.

```
let numeroGrande = BigInt(123121321231321212312312)
console.log(numeroGrande)
console.log(typeof(numeroGrande))
```

123121321231321212312312
bigint

## Tipos Compuestos (Objetos)

Los tipos compuestos u objetos son colecciones de propiedades **y se almacenan como referencias en la variable**. Cuando asignas un objeto a una variable en JavaScript, **lo que realmente estás almacenando en esa variable es una referencia al objeto, no el objeto en sí**. Esta referencia actúa de manera similar a un puntero, en el sentido de que apunta a un lugar en la memoria donde se almacena el objeto.

Los datos compuestos en JavaScript **son mutables**. Esto significa que sus contenidos o propiedades pueden ser modificados después de que han sido creados.

**Object:** Los objetos son colecciones de pares clave-valor. Cada clave accede a un valor.

```
let person = {
    name: "Alice",
    age: 25,
    isStudent: true
};
```

**Array:** son listas ordenadas de valores que pueden ser de cualquier tipo.

```
let numbers = [1, 2, 3, 4, 5];
```

**Function:** Las funciones son bloques de código que pueden ser definidos y luego invocados.

```
function greet(name) {
    return "Hello " + name + "!";
}
```

**Date:** Los objetos Date se utilizan para trabajar con fechas y horas.

```
let now = new Date();
```

**RegExp:** Representan expresiones regulares, que son patrones utilizados para hacer coincidir combinaciones de caracteres en cadenas.

```
let re = /ab+c/;
```

**Clases y Constructores:** Definiciones para crear instancias de objetos.

## Notas Adicionales

**Tipos Primitivos vs Compuestos:** Una diferencia clave entre primitivos y compuestos es cómo se maneja la asignación y copia. Los primitivos se copian por su valor, mientras que los compuestos se copian por su referencia.

**Tipado Dinámico:** JavaScript es un lenguaje de tipado dinámico, lo que significa que no necesitas especificar el tipo de dato al declarar una variable. El tipo de una variable puede cambiar durante la ejecución del programa.

**Wrapper Objects:** En JavaScript, existen objetos envoltorios para los tipos primitivos (String, Number, Boolean, Symbol, BigInt). Estos objetos raramente se utilizan directamente, pero son importantes en el funcionamiento interno del lenguaje.

En JavaScript, los objetos envoltorios (Wrapper Objects) son objetos especiales que proporcionan una forma de trabajar con los tipos de datos primitivos (como strings, numbers y booleans) como si fueran objetos. Esto es importante porque, en JavaScript, los tipos primitivos no son objetos y, por lo tanto, no pueden tener propiedades o métodos. Sin embargo, para proporcionar funcionalidades adicionales, JavaScript utiliza objetos envoltorios.

## VARIABLES

En JavaScript, existen principalmente tres formas de declarar variables (declarar una variable significa crearla), cada una con sus propias características y comportamientos: var, let, y const. Estas diferencias son importantes para entender cómo manejar los valores y el alcance de las variables en tu código.

**Asignación.** En una variable podemos asignar distintos tipos de valores mediante el operador de asignación, que es el símbolo **=**

**Iniciar variables** significa asignarles un valor inicial en el momento de su declaración.

```
let numero = 5; // Inicializa una variable 'numero' con el valor 5
```

## 1. var

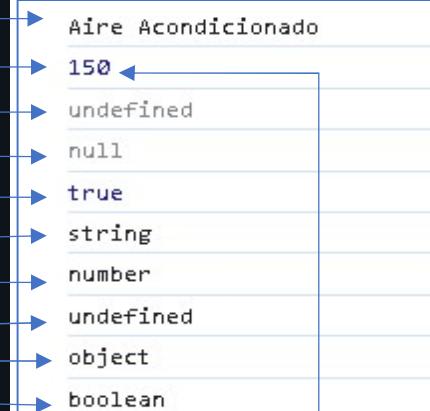
**Alcance de Función:** Las variables declaradas con var tienen un alcance de función, lo que significa que son accesibles dentro de toda la función en la que fueron declaradas. Si se declaran fuera de una función, tienen un alcance global.

**Elevación de Variable (Hoisting):** Las declaraciones var son elevadas al inicio de su contexto de ejecución. Esto significa que pueden ser referenciadas antes de su declaración en el código, aunque su inicialización no se eleva.

**Re-declaración Permitida:** Puedes re-declarar la misma variable varias veces usando var, lo cual puede llevar a errores sutiles.

**Inicialización Opcional:** Las variables var pueden ser declaradas sin inicializar, en cuyo caso se les asigna el valor **undefined**.

```
var producto = "Aire Acondicionado"
var precio = 150
var cantidad;
var valorNulo = null
var valorBooleano = true
console.log(producto)
console.log(precio)
console.log(cantidad)
console.log(valorNulo)
console.log(valorBooleano)
console.log(typeof(producto))
console.log(typeof(precio))
console.log(typeof(cantidad))
console.log(typeof(valorNulo))
console.log(typeof(valorBooleano))
```



### Re-Declaración:

En la consola los datos de tipo number se ven azules

```
var producto = "Aire Acondicionado"
var producto= "Monitor"
console.log(producto)
```

Monitor

### Reasignación

```
var producto = "Aire Acondicionado"
producto= "Monitor"
console.log(producto)
```

Monitor

## 2. let

**Alcance de Bloque:** Las variables let tienen un alcance de bloque, lo que significa que solo son accesibles dentro del bloque `({})` en el que fueron declaradas.

**No Hay Elevación de Variable (Hoisting):** Aunque técnicamente se elevan, no se puede acceder a ellas antes de su declaración en el bloque. Intentar hacerlo resultará en un error de referencia.

**Re-declaración No Permitida:** No se pueden re-declarar variables let dentro del mismo alcance, evitando así sobreescrituras accidentales.

**Inicialización Opcional:** Al igual que con var, puedes declarar una variable let sin inicializarla.

### Re-Declaración:

```
let producto = "Aire Acondicionado"
let producto= "Monitor"
console.log(producto)
```



✖️ Uncaught SyntaxError: Identifier 'producto' has already been declared (at [tempo\\_temporal.html:33](#))

### Reasignación

```
let producto1 = "Aire Acondicionado"
producto1= "Monitor"
console.log(producto1)
```

Monitor



## 3. const

**Alcance de Bloque:** Al igual que let, const tiene un alcance de bloque. `({})`

**Valores Inmutables:** Las variables declaradas con const deben ser inicializadas en el momento de la declaración y no pueden ser reasignadas después. Sin embargo, si la variable const es un objeto o un array, sus propiedades o elementos pueden ser modificados.

**No Hay Elevación de Variable (Hoisting):** Igual que con let, no se puede acceder a ellas antes de su declaración en el bloque.

**Re-declaración No Permitida:** No se puede re-declarar una variable const dentro del mismo alcance.

```
const PI=3.1416
console.log(PI)
```

3.1416

```
const ARREGLO = [25,30,['hola','juan']]
console.log(ARREGLO)
console.log(typeof(ARREGLO))
```

```
▼ (3) [25, 30, Array(2)] ⓘ
  0: 25
  1: 30
  ▶ 2: (2) ['hola', 'juan']
    length: 3
    ► [[Prototype]]: Array(0)

object
```

```
const OBJETO = {nombre:"juan", apellido:"Ríos"}
console.log(OBJETO)
console.log(typeof(OBJETO))
```

```
▶ {nombre: 'juan', apellido: 'Ríos'}
object
```

**Deben ser inicializadas (asignarles un valor) al momento de la creación.**

```
const PI;
```



✖ Uncaught SyntaxError: Missing initializer in const declaration (at [temporal.htm temporal.html:32 1:32:15](#))

### Re-declaración

```
const PI=3.1416
const PI=3.1416154
console.log(PI)
```



✖ Uncaught SyntaxError: Identifier 'PI' has already been declared (at [temporal.htm temporal.html:33 m1:33:15](#))

### Reasignación:

```
const PI=3.1416
PI=3.1416154
console.log(PI)
```



✖ ▶ Uncaught TypeError: Assignment to constant variable.
 at [temporal.html:33:11](#)



**Mutabilidad de Objetos y Arrays:** Aunque una variable const no puede ser reasignada, si apunta a un objeto o un array, los contenidos del objeto o array pueden ser modificados.

## Consideraciones Adicionales

**Preferencia por let y const:** Es una práctica común y recomendada usar let y const para la declaración de variables en lugar de var, debido a su manejo más predecible del alcance y a la prevención de errores comunes.

**La elección entre let y const** generalmente depende de si se espera que el valor de la variable cambie (let) o no (const).

**Variables Globales:** Las variables declaradas fuera de cualquier función o bloque son globales y accesibles desde cualquier parte del código. **El uso excesivo de variables globales generalmente se considera una mala práctica,** ya que pueden llevar a un código difícil de mantener y a conflictos de nombres.

## STRINGS

En JavaScript, las **cadenas de texto, conocidas como "strings"**, son una **secuencia de caracteres utilizados para representar y manejar texto**. Aquí hay varios aspectos importantes sobre las strings en JavaScript:

### Creación de Strings:

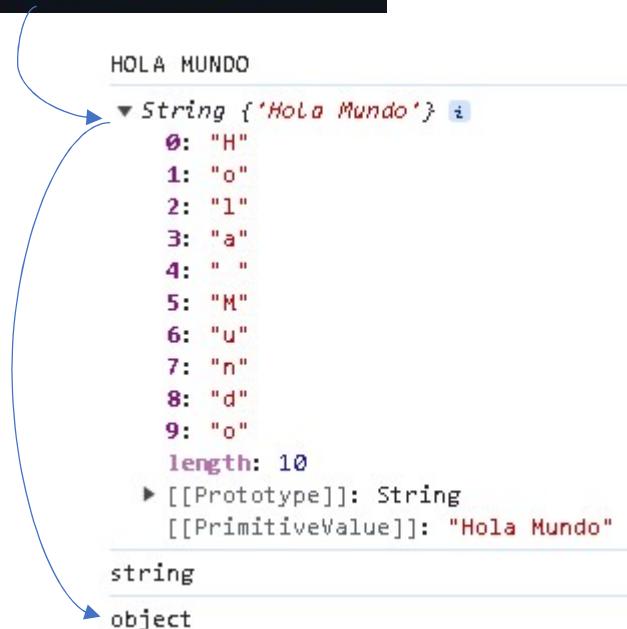
Puedes crear strings usando **comillas simples, dobles o acentos graves (backticks)**:

```
let string1 = 'Hola';
let string2 = "Mundo";
let string3 = `¡Hola Mundo!`;
```

Hola
Mundo
¡Hola Mundo!

**Otras dos formas** de crear Strings (**la segunda crea un objeto tipo string**)

```
const producto= String("HOLA MUNDO");
const producto1 = new String("Hola Mundo");
```



## Caracteres especiales:

- Las strings pueden incluir caracteres especiales como nuevas líneas (\n), tabuladores (\t), etc.
- Para incluir una comilla dentro de una string delimitada por el mismo tipo de comilla, debes escaparla con una barra invertida (\):

```
let frase = 'Él dijo, "Hola"';  
let otraFrase = "Y ella respondió, 'Hola'";  
let fraseConEscape = 'Él dijo, \"Hola\"';
```

Él dijo, "Hola"
Y ella respondió, 'Hola'
Él dijo, "Hola"

## Concatenación de Strings:

Las strings pueden ser **unidas usando el operador +:**

```
let saludo = string1 + ' ' + string2; // "Hola Mundo"
```

**El método concat()** en JavaScript se utiliza para unir dos o más strings. Este método devuelve una nueva string que es la unión de las strings pasadas como argumentos, sin modificar las strings originales.

```
let parte1 = "JavaScript ";  
let parte2 = "es ";  
let parte3 = "genial";  
let frase = parte1.concat(parte2, parte3); // "JavaScript es genial"
```



Aunque concat() es un método válido para unir strings, en la práctica moderna de JavaScript, es más común utilizar el operador + o las plantillas literales para concatenar strings debido a su sintaxis más simple y legibilidad

También se pueden **concatenar** usando **la sintaxis de plantillas literales con backticks:**

```
let saludo = `${string1} ${string2}`; // "Hola Mundo"
```



**Las plantillas literales (template literals)** en JavaScript son una forma de definir strings que facilita la incorporación de expresiones y la creación de cadenas de texto multilínea.

#### Sintaxis:

- Las plantillas literales se delimitan con acentos graves (backticks)

#### Interpolación de Expresiones:

- Puedes insertar expresiones dentro de una plantilla literal utilizando la sintaxis  `${expresión}`.
- Las expresiones dentro de  `${}` son evaluadas y el resultado se convierte en parte de la string.

#### Cadenas Multilínea:

- Las plantillas literales pueden abarcar múltiples líneas, lo que facilita la creación de strings multilínea sin necesidad de concatenar varias strings o utilizar caracteres de escape para los saltos de línea.

```
let dirección = `Calle Falsa 123  
Ciudad, País`;
```

```
Calle Falsa 123  
Ciudad, País
```

#### Expresiones Complejas:

- No solo puedes insertar variables en las plantillas literales, sino también expresiones más complejas, como operaciones matemáticas, llamadas a funciones, etc.

```
resultado = `La suma de 3 y 4 es ${3 + 4}`;
```

```
La suma de 3 y 4 es 7
```

#### Uso con Funciones:

- Las plantillas literales también pueden ser usadas con etiquetas, que son funciones que permiten analizar y modificar el contenido de la plantilla literal antes de que esta sea procesada en una string.



## Propiedades y Métodos:

- Las strings tienen una **propiedad .length** que devuelve su longitud.
- Hay varios **métodos** para manipular strings, como `.toUpperCase()`, `.toLowerCase()`, `.slice()`, `.split()`, `.indexOf()`, `.replace()`, entre otros.

```
let texto = "Hola Mundo";
console.log(texto.length); // 10
console.log(texto.toUpperCase()); // "HOLA MUNDO"
```

**El método `slice()`** en JavaScript es utilizado para extraer una sección de una string (corta una porción) y devuelve una nueva string, sin modificar la original.

```
str.slice(start, end)
```

- **start** (obligatorio): Índice en el que comienza la extracción. Si es negativo, se cuenta desde el final de la string.
- **end** (opcional): Índice antes del cual terminar la extracción. Si se omite, `slice()` extrae hasta el final de la string. Si es negativo, se refiere a la posición desde el final de la string.

Extraer los primeros 5 caracteres de una string:

```
let str = "Hola Mundo";
let subStr = str.slice(0, 5); // "Hola "
```

Extraer los últimos 5 caracteres de una string:

```
let subStr = str.slice(-5); // "Mundo"
```

Extraer caracteres entre dos índices:

```
let subStr = str.slice(2, 7); // "la Mu"
```

**El método `split()`** en JavaScript es utilizado para dividir una string en un array de substrings, basándose en un separador especificado. Este método es muy útil para descomponer una string en partes más pequeñas. Aquí te explico cómo funciona:

### 1. Sintaxis:

```
str.split([separador[, limite]])
```

- **separador (opcional): Especifica el carácter o la expresión regular que se usa para dividir la string.** Si se omite el separador, el array resultante contendrá un solo elemento con la string original completa.
- **limite (opcional): Un valor entero que especifica un límite en el número de divisiones a realizar.** El array resultante contendrá como máximo este número de elementos.

### 2. Comportamiento:

- La string original no se modifica; **.split()** devuelve un nuevo array.
- Si el **separador** es una cadena vacía (""), la string se divide entre cada carácter.
- Si el **separador** no se encuentra en la string, el array resultante tendrá un solo elemento, que será la string completa.

Dividir una string por espacios:

```
let frase = "Hola Mundo";
let palabras = frase.split(' '); // ['Hola', 'Mundo']

let caracteres = frase.split('');
// ['H', 'o', 'l', 'a', ' ', 'M', 'u', 'n', 'd', 'o']
```

Dividir una string usando un límite:

```
let parte = frase.split(' ', 1); // ['Hola']
```

**El método substring()** es una función de los objetos de cadena (string) que se utiliza para extraer una subcadena de una cadena de texto.

### Diferencias con slice()

```
let texto = "Hola Mundo";

// Extraer "Hola"
let subcadena1 = texto.substring(0, 4);

// Extraer "Mundo"
let subcadena2 = texto.substring(5);

// Extraer "Mun"
let subcadena3 = texto.substring(5, 8);

// Intercambio de indices (extraerá "Hola")
let subcadena4 = texto.substring(4, 0);
```

Aunque `substring()` y `slice()` pueden parecer similares, tienen algunas diferencias. Una diferencia clave es que `slice()` puede aceptar índices negativos, interpretándolos como posiciones desde el final de la cadena, mientras que `substring()` trata cualquier índice negativo como 0.

En la mayoría de los casos, `slice()` puede ser una mejor opción debido a su

flexibilidad con índices negativos y un comportamiento más predecible cuando se manejan índices fuera de rango. Sin embargo, `substring()` es útil cuando se necesita garantizar que todos los índices sean tratados como no negativos.

**El método indexOf()** en JavaScript es utilizado para **buscar la posición de la primera aparición de un valor especificado dentro de una string**. Este método devuelve el índice de la primera ocurrencia del valor buscado.

```
let frase = "Hola Mundo";
let posicion = frase.indexOf("Mundo"); // Retorna 5
```

**El método replace()** en JavaScript se utiliza para buscar una subcadena o una expresión regular dentro de una string y reemplazarla con una nueva subcadena. Este método es muy útil para modificar strings de manera dinámica

```
let frase = "Hola Mundo";
let nuevaFrase = frase.replace("Mundo", "a todos"); // "Hola a todos"
```

**El método trimStart()** en JavaScript es utilizado para eliminar los espacios en blanco del principio de una string. Este método no afecta a la string original; en su lugar, devuelve una nueva string con los espacios en blanco eliminados del inicio.

```
let ejemplo = "    Hola Mundo";
let resultado = ejemplo.trimStart(); // "Hola Mundo"
```

**trimStart()** es un método relativamente reciente y puede no estar disponible en versiones antiguas de algunos navegadores. En navegadores más antiguos, podrías necesitar usar **trimLeft()**, que es un alias de trimStart() y tiene la misma funcionalidad

**El método trimEnd()** en JavaScript es utilizado para eliminar los espacios en blanco del final de una string. Este método devuelve una nueva string sin los espacios en blanco al final, sin modificar la string original.

```
let ejemplo = "Hola Mundo    ";
let resultado = ejemplo.trimEnd(); // "Hola Mundo"
```



**Aplicar más de un método a un objeto en programación se conoce como "encadenamiento de métodos" o "method chaining".** Esta técnica permite llamar múltiples métodos en una sola línea de código, enlazando los llamados de los métodos uno tras otro.

Eliminar en las dos direcciones los espacios en blanco:

```
const producto= String("      HOLÀ MUNDO      ");
console.log(producto.trimStart().trimEnd())
HOLA MUNDO
```

**El método repeat()** se utiliza para crear una nueva cadena que consiste en la cadena original repetida un número específico de veces. Este método es útil cuando necesitas duplicar o multiplicar el contenido de una cadena.

```
let texto = "abc";

// Repetir "abc" 3 veces: "abcabcabc"
let repetido1 = texto.repeat(3);
```

El método repeat() es útil en varios escenarios, como la creación de patrones, el relleno de espacios para alinear texto en consolas o interfaces, o simplemente cuando se necesita duplicar una cadena un número determinado de veces sin necesidad de utilizar bucles.



## Distinción entre propiedades y métodos

En JavaScript, así como en otros lenguajes de programación orientados a objetos, la **distinción entre propiedades y métodos de una variable (o más precisamente, de un objeto)** es fundamental:

### 1. Propiedades:

- **Las propiedades son valores asociados con un objeto.** Pueden ser básicamente cualquier tipo de dato válido en JavaScript: números, cadenas de texto, booleanos, funciones, otros objetos, etc.
- **Accedes a las propiedades de un objeto usando la notación de punto o la notación de corchetes.** Por ejemplo, si tienes un objeto **persona**, puedes acceder a su propiedad **nombre** usando **persona.nombre** o **persona['nombre']**.
- Las propiedades pueden ser estáticas (asociadas con la clase/constructor) o de instancia (asociadas con una instancia específica del objeto).

### 2. Métodos:

- **Los métodos son funciones asociadas con un objeto.** Son un tipo especial de propiedad cuyos valores son funciones.
- Al igual que las propiedades, **los métodos se acceden usando la notación de punto o de corchetes, pero se invocan (se ejecutan) utilizando paréntesis**, por ejemplo, **persona.saludar()**.
- Los métodos pueden realizar acciones que a menudo afectan los datos del objeto o realizan cálculos basados en esos datos.
- Al igual que las propiedades, los métodos pueden ser estáticos o de instancia.

Por ejemplo, considera el siguiente objeto **persona**:

```
let persona = {
    nombre: 'Ana',
    edad: 30,
    saludar: function() {
        console.log(`Hola, mi nombre es ${this.nombre}`);
    }
};
```

En este ejemplo:

- **nombre** y **edad** son propiedades del objeto **persona**.
- **saludar** es un método del objeto **persona**.

La principal diferencia es que **las propiedades almacenan datos, mientras que los métodos son funciones** que realizan alguna acción con esos datos.



Aunque JavaScript maneja los datos primitivos de una manera que a veces puede parecer que son objetos (debido a la conversión automática a objetos para acceder a métodos y propiedades - Wrapper Objects), hay una distinción clara entre tipos primitivos y objetos en JavaScript. Esta distinción es diferente a la aproximación de Python, donde casi todo es un objeto.



## Comparación de Strings:

Las strings se pueden comparar usando operadores de igualdad (==, ===). La comparación es sensible a mayúsculas y minúsculas.

## Inmutabilidad:

Las strings en JavaScript son inmutables, lo que significa que una vez creadas, no pueden ser modificadas. Los métodos que parecen modificar una string en realidad retornan una nueva string.

# NUMEROS (NUMBER)

## Tipos de Números

**1. Números Enteros:** Incluyen tanto números positivos como negativos, así como el 0. Ejemplo: 5, -3, 0.

**2. Números Decimales (Flotantes):** Números con decimales. Ejemplo: 3.14, -0.5.

**3. Infinity y -Infinity:** Representan el infinito matemático positivo y negativo.

**4. NaN (Not a Number):** Indica un valor que no es un número, a menudo como resultado de una operación matemática indefinida o errónea.

**5. BigInt:** En versiones recientes de JavaScript, **el tipo BigInt** fue introducido para representar **números enteros muy grandes**.

## Creación de Números

En JavaScript, hay varias formas de crear números, dependiendo del contexto y del tipo de número que necesites

**Uso de Literales Numéricos:** Es la forma más directa y común de crear números. Simplemente escribes el número en tu código.

```
let num1 = 100;           // Entero
let num2 = 3.14;          // Decimal
let num3 = -45;           // Negativo
let num4 = 2.99e8;         // Notación científica (equivale a 299000000)
```

**Convertir de Cadena a Número:** Puedes convertir cadenas en números **usando parseInt(), parseFloat()**, o el **constructor Number()**.

```
let num5 = parseInt("123");      // Convierte la cadena en entero
let num6 = parseFloat("3.14");    // Convierte la cadena en decimal
```

```
let num7 = Number("456");
                    // Convierte la cadena en número (entero o decimal)
```

**Operadores Aritméticos:** Los números también pueden ser creados como resultado de operaciones aritméticas.

```
let num8 = 10 + 20;    // Resultado de una suma
let num9 = 50 / 2;     // Resultado de una división
```

**El Objeto Number:** Se puede utilizar el **constructor Number** para **crear un objeto número**. Aunque **no es común en la práctica**, ya que los literales numéricos son más simples y directos.

```
let num10 = new Number(123); // Crea un objeto número
```

**Notación Literal de BigInt:** Para crear un número BigInt (números enteros muy grandes), se añade n al final del número.

```
let bigNum = 12345678901234567890n; // Crea un BigInt
```

**Uso del Objeto Math:** El objeto Math proporciona varias constantes y métodos para trabajar con números, como Math.PI para obtener el valor de  $\pi$ , o Math.random() para obtener un número aleatorio.

```
let pi = Math.PI; // Valor de  $\pi$ 
let randomNum = Math.random(); // Número aleatorio entre 0 y 1
```

**Hexadecimales, Binarios y Octales:** JavaScript permite crear números en bases distintas a la decimal (base 10) utilizando prefijos específicos (0x para hexadecimal, 0b para binario, 0o para octal).

```
let hexNum = 0xFF; // Hexadecimal para 255
let binNum = 0b1010; // Binario para 10
let octNum = 0o744; // Octal para 484
```

## Operaciones con Números

**1. Operaciones Aritméticas Básicas:** Suma (+), resta (-), multiplicación (\*), y división (/).

**2. Operador de Módulo (%):** Devuelve el resto de una división.

**3. Incremento (++) y Decremento (--):** Aumentan o disminuyen un número en una unidad.

**4. Math Object:** En Javascript existe un objeto completo (en la ventana global – por eso se puede acceder hasta desde la consola) que se llama Math y proporciona métodos para operaciones matemáticas más complejas, como **Math.pow()**, **Math.sqrt()**, **Math.round()**, **Math.random()**, entre otros.

```
> Math
< ▾ Math {abs: f, acos: f, acosh: f, asin: f, asinh: f, ...} ⓘ
  E: 2.718281828459045
  LN2: 0.6931471805599453
  LN10: 2.302585092994046
  LOG2E: 1.4426950408889634
  LOG10E: 0.4342944819032518
  PI: 3.141592653589793
  SQRT1_2: 0.7071067811865476
  SQRT2: 1.4142135623730951
  ► abs: f abs()
  ► acos: f acos()
  ► acosh: f acosh()
  ► asin: f asin()
  ► asinh: f asinh()
  ► atan: f atan()
  ► atan2: f atan2()
  ► atanh: f atanh()
  ► cbrt: f cbrt()
  ► ceil: f ceil()
  ► clz32: f clz32()
  ► cos: f cos()
  ► cosh: f cosh()
  ► exp: f exp()
  ► expm1: f expm1()
  ► floor: f floor()
  ► fround: f fround()
  ► hypot: f hypot()
  ► imul: f imul()
  ► log: f log()
  ► log1p: f log1p()
  ► log2: f log2()
  ► log10: f log10()
  ► max: f max()
  ► min: f min()
  ► pow: f pow()
  ► random: f random()
  ► round: f round()
  ► sign: f sign()
  ► sin: f sin()
  ► sinh: f sinh()
  ► sqrt: f sqrt()
  ► tan: f tan()
  ► tanh: f tanh()
  ► trunc: f trunc()
  ► Symbol(Symbol.toStringTag): "Math"
  ► [[Prototype]]: Object
```

## Precisión y Problemas

Los números de punto flotante pueden tener problemas de precisión. Por ejemplo,  $0.1 + 0.2$  no es exactamente  $0.3$  debido a cómo los números son representados en memoria.

Para **operaciones que requieren alta precisión**, como las financieras, se recomienda usar bibliotecas especializadas o **BigInt** para números enteros muy grandes.

 Los números en JavaScript son versátiles y adecuados para la mayoría de las operaciones matemáticas comunes en el desarrollo web y de aplicaciones. Sin embargo, para operaciones muy específicas o con requerimientos de precisión alta, siempre es bueno considerar las limitaciones y buscar alternativas como BigInt o librerías matemáticas externas.

## Orden de las operaciones

En JavaScript, el orden de las operaciones sigue las reglas estándar de la aritmética, conocidas como la jerarquía de operaciones o la regla PEMDAS/BODMAS. Esto determina el orden en que se evalúan las operaciones en una expresión. Aquí está el orden, de mayor a menor prioridad:

### 1. Paréntesis:

- Las operaciones dentro de los paréntesis se realizan primero. Esto se aplica a los paréntesis `()`, corchetes `[]` y llaves `{}`.

### 2. Exponentes:

- Operaciones de exponenciación (`\*\*`). Se evalúan de derecha a izquierda.

### 3. Multiplicación y División:

- Estas operaciones tienen la misma prioridad. Se evalúan de izquierda a derecha.

### 4. Suma y Resta:

- Al igual que la multiplicación y la división, la suma y la resta tienen la misma prioridad y se evalúan de izquierda a derecha.

### 5. Operadores de Asignación:

- Como `=`, `+=`, `-=`. Estos generalmente tienen la menor prioridad.

### 6. Operadores Lógicos:

- Operadores como `&&` (y lógico), `||` (o lógico), etc., se evalúan después de las operaciones aritméticas.

## **Consideraciones Adicionales**

**Asociatividad:** En **operaciones con la misma prioridad**, la asociatividad (izquierda a derecha o derecha a izquierda) determina el orden de operación.

La mayoría de las operaciones en JavaScript tienen asociatividad de izquierda a derecha. Esto significa que, si dos operaciones comparten la misma prioridad, la operación que está más a la izquierda se realiza primero. Por ejemplo:

Suma y Resta:  $3 + 4 - 2$  se evalúa como  $(3 + 4) - 2$ .

**Operadores Unarios:** Operadores como `++`, `--`, `!` (negación lógica), etc., también tienen su propia prioridad y generalmente se aplican directamente al operando que les sigue.

## **Operadores Unarios**

En JavaScript, los operadores unarios son operadores que **actúan sobre un solo operando (un valor o una expresión)**.

**Operador de Negación (!):** Se utiliza para negar una expresión booleana. Si la expresión es verdadera, la negación la hace falsa, y viceversa.

```
let verdadero = true;
let falso = !verdadero; // falso
```

**Operador de Incremento (++) y Decremento (--):** Estos operadores se utilizan para aumentar o disminuir el valor de una variable numérica en 1 unidad, respectivamente.

```
let contador = 5;
contador++; // Incremento en 1: contador ahora es 6
contador--; // Decremento en 1: contador ahora es 5 nuevamente
```

**Operador de Negación Numérica (-):** Este operador se utiliza para convertir una expresión en un valor numérico negativo.

```
let positivo = 10;
let negativo = -positivo; // -10
```

**Operador de Tipo (typeof):** Se utiliza para obtener el tipo de datos de una variable o expresión.

```
let numero = 42;
let tipo = typeof numero; // "number"
```

## Operadores Binarios

Los operadores binarios en JavaScript son operadores que **actúan sobre dos operandos, es decir, dos valores o expresiones.**

### Operadores Aritméticos:

**+ (suma):** Suma dos valores numéricos.

**- (resta):** Resta el segundo valor del primero.

**\* (multiplicación):** Multiplica dos valores numéricos.

**/ (división):** Divide el primer valor por el segundo.

**% (módulo):** Devuelve el resto de la división del primer valor por el segundo.

```
let resultado = 10 + 5; // resultado es 15
```

### Operadores de Comparación:

**== (igual a):** Compara si dos valores son iguales en valor (no en tipo).

**!= (diferente de):** Compara si dos valores no son iguales en valor (no en tipo).

**===(estrictamente igual a):** Compara si dos valores son iguales en valor y tipo.

**!==(estrictamente diferente de):** Compara si dos valores no son iguales en valor o tipo.

**> (mayor que):** Compara si el primer valor es mayor que el segundo.

**< (menor que):** Compara si el primer valor es menor que el segundo.

**>= (mayor o igual que):** Compara si el primer valor es mayor o igual que el segundo.

**<= (menor o igual que):** Compara si el primer valor es menor o igual que el segundo.

```
let resultado = 10 > 5; // resultado es true
```

### Operadores Lógicos:

**&& (y lógico):** Retorna true si ambas condiciones son verdaderas.

**|| (o lógico):** Retorna true si al menos una de las condiciones es verdadera.

**! (no lógico):** Invierte el valor de verdad de una condición.

```
let resultado = (10 > 5) && (5 < 3); // resultado es false
```

## **Operadores de Asignación:**

**= (asignación):** Asigna un valor a una variable.

**+=, -=, \*=, /=, %=:** Realizan una operación y asignan el resultado a la variable.

Por ejemplo, **+=, -=**, estos operadores se utilizan para aumentar o disminuir el valor de una variable numérica en tantas unidades como se asignen, en el caso siguiente aumentará de tres en tres.

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

```
let numero = 5;
numero += 3;
numero += 3;
numero += 4;
console.log(numero);
```

15

## **Operadores Ternarios**

Los operadores ternarios en JavaScript **son operadores condicionales que permiten tomar decisiones basadas en una condición**. Tienen la siguiente sintaxis:

```
condicion ? valorSiVerdadero : valorSiFalso
```

Aquí tienes una explicación detallada de cómo funcionan:

**condicion:** Una expresión que se evalúa como verdadera o falsa.

**valorSiVerdadero:** El valor que se devuelve si la condición es verdadera.

**valorSiFalso:** El valor que se devuelve si la condición es falsa.

El operador ternario evalúa la condición primero. Si la condición es verdadera, devuelve valorSiVerdadero; de lo contrario, devuelve valorSiFalso.

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

```
let esDiaLaboral = true;
let actividad = esDiaLaboral ? "Trabajo" : "Descanso";
console.log(actividad); // "Trabajo"
```

```
let numero = 5;
let resultado = (numero % 2 === 0) ? "Par" : "Impar";
console.log(resultado); // "Impar"
```

## Propiedades y Métodos

**parseInt()** Convierte un string a un número entero.

```
let cadena = "123";
let numero = parseInt(cadena);

console.log(numero); // Esto imprimirá 123 como un número entero.
```

**parseFloat()** Convierte un string a un número decimal.

```
let cadena = "3.14";
let numero = parseFloat(cadena);
console.log(numero); // Esto imprimirá 3.14 como un número decimal.
```

Convertir una cadena en un número, ya sea entero o decimal. Aquí tienes un ejemplo:

```
let cadena = "42";
let numero = Number(cadena);
console.log(numero);

let cadena1 = "42.7";
let numero1 = Number(cadena1);
console.log(numero1);
```

42
42.7



Es importante tener en cuenta que, si la cadena no se puede analizar como un número válido, la conversión dará como resultado NaN

```
let cadena = "cuarenta y dos";
let numero = Number(cadena);
console.log(numero);
console.log(typeof(numero));
```

NaN
number

**toString()**: Convierte un número a una cadena de texto (string).

```
let num = 123;
let str = num.toString(); // "123"
```

**toFixed(n):** Redondea el número a un número específico de decimales.

```
let num = 2.345;
let fixed = num.toFixed(2); // "2.35"
```

**toPrecision(n):** Formatea un número a una longitud específica.

```
let num = 7.1234;
let precise = num.toPrecision(3); // "7.12"
```

## Dar formato a un número como pesos argentinos

Puedes utilizar el método **toLocaleString()**. Este método permite formatear un número como una cadena de texto según una localidad específica, en este caso, **la localidad de Argentina ('es-AR')**, y también puedes especificar el estilo de moneda.

```
let numero = 123456.78;
let formatoPesosArgentinos = numero.toLocaleString('es-AR', {
  style: 'currency',
  currency: 'ARS'
});

console.log(formatoPesosArgentinos); // Ejemplo de salida: "$ 123.456,78"
```

'**es-AR**' especifica la localidad para Argentina.

**style: 'currency'** indica que el formato debe ser de tipo moneda.

**currency: 'ARS'** especifica la moneda como el peso argentino (ARS).

**Esto dará como resultado un número formateado como moneda, incluyendo el símbolo del peso argentino y la estructura numérica acorde al estándar argentino (puntos para los miles y coma para los decimales).**

## Comprobaciones

**isNaN()** para verificar si un valor no es un número.

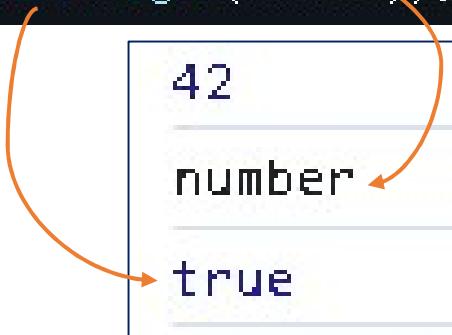
**isFinite()** para verificar si un valor es un número finito.

```
let cadena = "42";
let numero = Number.parseInt(cadena);
console.log(Number.isNaN(numero));
console.log(Number.isFinite(numero));
```

false
true

**isInteger()** Para saber si un número es entero:

```
let cadena = "42";
let numero = Number.parseInt(cadena);
console.log(numero);
console.log(typeof(numero));
console.log(Number.isInteger(numero));
```



## BOOLEAN

Los booleanos en JavaScript son un tipo de dato **que solo puede tener dos valores: true (verdadero) o false (falso)**. Son fundamentales en la programación, ya que **se utilizan para realizar comprobaciones lógicas y controlar el flujo del programa.**

### Conceptos Clave

**Valores Literales:** Los valores true y false son literales booleanos en JavaScript.

```
let verdadero = true;
let falso = false;
```

**Operadores Lógicos:** Se utilizan para realizar operaciones lógicas, como **AND (&&)**, **OR (||)** y **NOT (!)**.

```
let resultado1 = true && false; // false
let resultado2 = true || false; // true
let resultado3 = !true; // false
```

**Operadores de Comparación:** Se usan para comparar valores, como igualdad (==, ===), desigualdad (!=, !==), mayor que (>), menor que (<), etc.

```
let esIgual = 5 === 5; // true
let esMayor = 10 > 5; // true
```



**Conversión a Booleano:** JavaScript convierte automáticamente valores a booleanos en contextos lógicos. **Algunos valores siempre se convierten en false (falsos), como 0, NaN, "" (cadena vacía), null, undefined, y por supuesto false. Todos los demás valores se convierten en true.**

```
let miValor = "Hola";
if (miValor) {
  console.log("Es verdadero"); // Se ejecuta porque "Hola" es verdadero (true)
}
```

**Declaraciones Condicionales y Bucles:** Los booleanos son comúnmente usados en declaraciones condicionales como **if, else if, else**, y en bucles como **while y for**.

```
let edad = 20;
if (edad >= 18) {
  console.log("Es adulto"); // Se ejecuta si la condición es verdadera
}
```

## Ejemplos Prácticos

**Validación de Formularios:** Comprobar si el usuario ha ingresado todos los datos necesarios.

**Control de Fluxos:** Decidir qué código ejecutar basado en ciertas condiciones.

**Bucles:** Continuar o detener un bucle basado en una condición booleana.

**Los booleanos son una parte esencial del control de flujo y la lógica en JavaScript, permitiendo a los desarrolladores escribir código que reaccione de manera diferente bajo diferentes circunstancias.**

## OBJETOS

Los objetos en JavaScript son estructuras de datos fundamentales que permiten almacenar colecciones de pares clave-valor. En un objeto, **los valores pueden ser de cualquier tipo**, incluyendo números, cadenas, arrays, funciones e incluso otros objetos.

**Los objetos en JavaScript son mutables.**



En JavaScript, cuando hablamos de "propiedades" de un objeto, nos referimos tanto a la clave (también conocida como "nombre de la propiedad" o "key") como al valor asociado a esa clave. Juntos, la clave y su valor correspondiente forman una "propiedad" del objeto.

En este ejemplo el objeto persona tiene tres propiedades:

1. **La propiedad nombre**, cuya clave es nombre y cuyo valor es "Juan".
2. **La propiedad edad**, cuya clave es edad y cuyo valor es 30.
3. **La propiedad esEstudiante**, cuya clave es esEstudiante y cuyo valor es true.

## Creación de Objetos

**Literal de Objeto:** Es la forma más común y directa de crear un objeto.

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

**Constructor new Object():** Una forma menos común de crear objetos usando el constructor Object.

```
let persona = new Object();
persona.nombre = "Juan";
persona.edad = 30;
persona.esEstudiante = true;
```

## Object Constructor → Constructor de Objetos

El constructor de objetos se refiere a **una función especial que se utiliza para inicializar nuevos objetos**. El término "constructor de objetos" puede referirse a dos cosas diferentes, dependiendo del contexto:

**Función Constructora Genérica de Objetos:** JavaScript tiene un constructor de objetos genérico incorporado llamado Object(). Este constructor crea un objeto vacío o convierte valores a objetos.

```
let obj1 = new Object();
let obj2 = new Object({a: 1, b: 2}); // Crea un objeto con propiedades
```

**Funciones Constructoras Personalizadas:** En JavaScript, puedes crear tus propias funciones constructoras para definir objetos personalizados. Las funciones constructoras son útiles cuando necesitas crear múltiples objetos que comparten las mismas propiedades y métodos.

```
function Persona(nombre, edad) {
    this.nombre = nombre;
    this.edad = edad;
    this.saludar = function() {
        console.log('Hola, mi nombre es ' + this.nombre);
    };
}

let personal = new Persona("Alberto", 30);
personal.saludar(); // "Hola, mi nombre es Alberto"
```

## Uso de this en Funciones Constructoras

El uso de this en una función constructora es fundamental. Se refiere al objeto recién creado y permite asignarle propiedades y métodos. Cuando se llama a una función con el nuevo operador new, JavaScript crea automáticamente un nuevo objeto, y this dentro de la función se refiere a ese nuevo objeto.

### Nota sobre la Nomenclatura

Es una convención común en JavaScript **nombrar las funciones constructoras con una letra mayúscula inicial**. Esto ayuda a distinguirlas de las funciones normales.

### Prototipos

En JavaScript, cada función constructora tiene una propiedad prototype que es compartida por todas las instancias creadas a partir de esa función constructora. Esto es útil para agregar métodos que deben estar disponibles en todas las instancias:

```
Persona.prototype.despedirse = function() {
    console.log('Adiós, me llamo ' + this.nombre);
};

personal.despedirse(); // "Adiós, me llamo Alberto"
```

Agregar métodos al prototipo en lugar de directamente en el constructor es más eficiente en términos de memoria, ya que este método se

almacena una sola vez en memoria, en lugar de en cada instancia del objeto.

## Acceso a Propiedades

Puedes acceder y modificar las propiedades de un objeto usando la **notación de punto** o la **notación de corchetes**.

**Notación de Punto:** Más usada para propiedades con nombres que son identificadores válidos.

```
let persona = {  
    nombre: "Carlos",  
    edad: 25  
};  
  
console.log(persona.nombre);  
console.log(persona.edad);
```

Carlos
25

**Notación de Corchetes:** Útil cuando el nombre de la propiedad es una cadena con caracteres especiales o cuando el nombre de la propiedad está en una variable.

```
let persona = {  
    nombre: "Carlos",  
    edad: 25  
};  
  
console.log(persona["nombre"]);  
console.log(persona["edad"]);
```

Carlos
25

## Modificación de Propiedades

```
let persona = {  
    nombre: "Carlos",  
    edad: 25  
};  
persona.nombre="Roberto"  
persona.edad=37  
console.log(persona.nombre);  
console.log(persona.edad);
```

Roberto
37

## Agregar Propiedades

```
let persona = {  
    nombre: "Carlos",  
    edad: 25  
};  
persona.nombre="Roberto"  
persona.edad=37  
persona.estadoCivil="Soltero"  
console.log(persona.nombre);  
console.log(persona.edad);  
console.log(persona.estadoCivil);
```

Roberto
37
Soltero

## Eliminar Propiedades

```
let persona = {
  nombre: "Carlos",
  edad: 25
};
persona.nombre="Roberto";
delete persona.edad;
persona.estadoCivil="Soltero";
console.log(persona);
```

```
▼ {nombre: 'Roberto', estadoCivil: 'Soltero'} i
  estadoCivil: "Soltero"
  nombre: "Roberto"
▶ [[Prototype]]: Object
```

## Métodos de Objetos

**Los objetos también pueden tener funciones como propiedades**, conocidas como métodos.

```
let persona = {
  nombre: "Juan",
  saludar: function() {
    console.log("Hola, soy " + this.nombre);
  }
};

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



**Función:** Un bloque de código reutilizable que realiza una tarea específica. Puede existir por sí sola. **sumar()**

**Método:** Una función asociada con un objeto. Depende de ese objeto para su contexto o datos. **objeto.sumar()**



**DESTRUCTURING DE OBJETOS → extraer propiedades y asignarlas a distintas variables individuales. → DESEMPAQUETAMIENTO**

La desestructuración de objetos en JavaScript es **una técnica que permite extraer propiedades de un objeto (o elementos de un array) y asignarlas a variables individuales.**

Cuando desestructuras un objeto, extraes sus propiedades y las asignas a variables con el mismo nombre de las propiedades, o a nuevas variables con nombres diferentes.

Ejemplo básico:

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

// Desestructuración
let { nombre, edad } = persona;
console.log(nombre);
console.log(edad);
```

Carlos
30

## Renombrar Variables en la Desestructuración

Puedes asignar los valores de las propiedades a variables con diferentes nombres.

```
let { nombre: nombreUsuario, edad: edadUsuario } = persona;

console.log(nombreUsuario); // Carlos
console.log(edadUsuario); // 30
```

En este ejemplo, las propiedades nombre y edad se asignan a las nuevas variables nombreUsuario y edadUsuario.

## Destructuring o desestructuración Anidada

**Si un objeto contiene otros objetos como propiedades**, también puedes desestructurar las propiedades del objeto anidado

```
let persona = {
    nombre: "Carlos",
    edad: 30,
    dirección: {
        calle: "Principal",
        ciudad: "Madrid"
    }
};

// Desestructuración anidada
let { nombre, dirección: { calle, ciudad } } = persona;

console.log(calle); // Principal
console.log(ciudad); // Madrid
```

En este caso, calle y ciudad se extraen del objeto dirección que está anidado dentro de persona.

## Objetos dentro de objetos

Para **acceder a propiedades de un objeto que está contenido dentro de otro objeto** en JavaScript, puedes hacerlo utilizando **la notación de puntos o la notación de corchetes**. Aquí tienes un ejemplo para ilustrar ambos métodos:

```
let persona = {
    nombre: "Alberto",
    edad: 52,
    domicilio: {
        direccion: "Av. Zand 81",
        ciudad: "Laguna Chica"
    }
};
```

### Notación de puntos:

```
let ciudad = persona.domicilio.ciudad;
console.log(ciudad); // Muestra "Laguna Chica"
```

### Notación de corchetes:

```
let ciudad = persona["domicilio"]["ciudad"];
console.log(ciudad); // Muestra "Laguna Chica"
```

## Prototipos y Herencia

En JavaScript, **cada objeto tiene un prototipo, un objeto del que hereda métodos y propiedades**. La herencia prototípica es un aspecto fundamental de la forma en que JavaScript maneja los objetos y sus relaciones.

Los prototipos y la herencia **permiten compartir propiedades y métodos entre objetos**. Estos conceptos son parte del mecanismo de prototipos, que es una forma de herencia basada en objetos más que en clases, como se ve en muchos otros lenguajes de programación orientados a objetos.

### Prototipos

**Cada objeto en JavaScript tiene una propiedad interna y oculta llamada [[Prototype]]** (a menudo accesible a través de \_\_proto\_\_ en los navegadores, aunque esta forma de acceso está obsoleta). Esta propiedad es una referencia a otro objeto, conocido como el "prototipo" del objeto.

**El prototipo de un objeto sirve como una plantilla o un "objeto padre", del cual el objeto "hijo" hereda propiedades y métodos.** Si intentas acceder a una propiedad o método de un objeto que no está definido directamente en él, JavaScript buscará automáticamente esa propiedad o método en el prototipo del objeto, y así sucesivamente en la cadena de prototipos, hasta que lo encuentre o llegue al final de la cadena.

### **Herencia prototípica de Objetos**

En JavaScript, los objetos pueden heredar propiedades y métodos de otros objetos a través de la cadena de prototipos. Esto es conocido como **herencia prototípica**.

## **Objeto this**

Dentro de los métodos de un objeto, **this se refiere al objeto mismo, lo que permite acceder a sus propiedades y métodos**.

La palabra **this** se refiere a los valores que existen dentro del objeto.

```
let persona = {
    nombre: "Juan",
    saludar: function() {
        console.log("Hola, soy " + this.nombre);
    }
};
```

## **Mutabilidad de Objetos (CONST)**

**Cuando se declara un objeto utilizando la palabra clave const, lo que se mantiene constante es la referencia al objeto, no los contenidos del objeto en sí.** Esto significa **que no puedes reasignar la variable a un objeto diferente, pero sí puedes modificar las propiedades** del objeto original.

```
const persona = {
    nombre: "Alberto",
    edad: 52,
    domicilio: {
        direccion: "Av. Zand 81",
        ciudad: "Laguna Chica"
    }
};

// Cambiando la edad de 52 a 32
persona.edad = 32;

console.log(persona.edad); // Muestra 32
```

Aquí te muestro cómo puedes cambiar la edad de la persona de 52 a 32 en el ejemplo anterior, aunque el objeto esté declarado como const:

En este código, aunque persona es una constante, puedes cambiar la propiedad edad de 52 a 32 sin ningún problema. **Lo que no podrías hacer es reasignar persona a un objeto completamente nuevo**, como **persona = { ... }**, ya que eso

intentaría cambiar la referencia del objeto, lo cual no está permitido con variables declaradas con const.

## Congelar objetos para no poder modificarlos

Puedes utilizar el **método Object.freeze()** para congelar un objeto. Una vez congelado, no podrás agregar, eliminar, o modificar las propiedades de ese objeto. Esto es útil cuando quieras asegurarte de que un objeto permanezca inmutable.

```
const persona = {
    nombre: "Alberto",
    edad: 52,
    domicilio: {
        direccion: "Av. Zand 81",
        ciudad: "Laguna Chica"
    }
};

Object.freeze(persona);

// Intentando modificar la propiedad
persona.edad = 32; // Esto no tendrá efecto

console.log(persona.edad); // Muestra 52, ya que el objeto está congelado
```

En este ejemplo, después de aplicar `Object.freeze()` a `persona`, cualquier intento de modificar las propiedades de `persona` será ignorado.

Sin embargo, **hay que tener en cuenta que `Object.freeze()` es superficial. Esto significa que si el objeto contiene otros objetos (como en el caso de `domicilio` dentro de `persona`), estos objetos anidados no se congelan.** Para congelar completamente un objeto incluyendo sus objetos anidados, tendrías que aplicar `Object.freeze()` recursivamente a todos los objetos anidados.

```
function deepFreeze(objeto) {
    Object.keys(objeto).forEach(prop => {
        if (typeof objeto[prop] === 'object' && objeto[prop] !== null) {
            deepFreeze(objeto[prop]);
        }
    });
    return Object.freeze(objeto);
}

deepFreeze(persona);

// Intentando modificar la propiedad anidada
persona.domicilio.ciudad = "Nueva Ciudad"; // Esto no tendrá efecto

console.log(persona.domicilio.ciudad); // Muestra "Laguna Chica"
```

En este último ejemplo, el **método `deepFreeze()`** aplica **`Object.freeze()` de manera recursiva a todas las propiedades de objeto**, incluyendo objetos anidados, asegurando que el objeto completo se vuelva inmutable.

## Métodos aplicables a objetos

En JavaScript, los objetos pueden interactuar con una variedad de métodos incorporados. Estos métodos proporcionan funcionalidades para realizar operaciones comunes con objetos.

**`Object.assign()`:** Copia las propiedades de uno o más objetos fuente a un objeto destino (une dos objetos).

```
const obj1 = { a: 1 };
const obj2 = { b: 2 };
const obj3 = Object.assign(obj1, obj2);
console.log(obj3); // { a: 1, b: 2 }
```

Esto mismo se puede hacer **usando el operador de propagación (spread operator) (...)**.

```
const obj1 = { a: 1, b: 2 };
const obj2 = { b: 3, c: 4 };
const combinado = { ...obj1, ...obj2 }; // { a: 1, b: 3, c: 4 }
```

**`Object.keys()`:** Devuelve un array con los nombres de las propiedades (keys) de un objeto.

```
const objeto = { a: 1, b: 2, c: 3 };
console.log(Object.keys(objeto)); // ["a", "b", "c"]
```

**`Object.values()`:** Devuelve un array con los valores de las propiedades de un objeto.

```
console.log(Object.values(objeto)); // [1, 2, 3]
```

**`Object.entries()`:** Devuelve un array con arrays de [key, value] para cada propiedad en el objeto.

```
console.log(Object.entries(objeto)); // [["a", 1], ["b", 2], ["c", 3]]
```

**Object.freeze():** Congela un objeto. Las propiedades del objeto no pueden ser modificadas.

**Object.isFrozen():** Verifica si un objeto está congelado.

**Object.seal():** Previene la adición o eliminación de propiedades de un objeto, pero permite modificar las propiedades existentes.

**Object.isSealed():** Verifica si un objeto está sellado.

**Object.defineProperty():** Permite definir una nueva propiedad en un objeto o modificar una existente.

```
Object.defineProperty(objeto, 'nuevaPropiedad', {
  value: 4,
  writable: true,
  enumerable: true,
  configurable: true
});
```

**Object.getOwnPropertyDescriptor():** Obtiene la descripción de una propiedad específica de un objeto.

## ARRAYS (ARREGLOS)

Los arrays en JavaScript son **estructuras de datos que te permiten almacenar varios valores en una sola variable (se denominan elementos)**. Son uno de los tipos de datos más útiles y comúnmente usados en JavaScript para **almacenar colecciones de datos**. Aquí tienes una visión general sobre los arrays en JavaScript, incluyendo cómo se crean y manipulan.

Son estructuras de datos versátiles que **pueden contener una amplia variedad de tipos de datos, y son mutables**.

## Creación de Arrays

Puedes crear un array de varias formas:

**Usando corchetes (Array literal):**

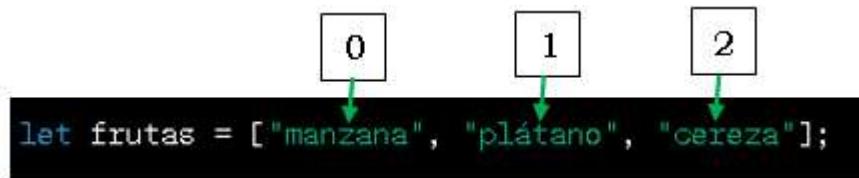
```
let frutas = ["manzana", "plátano", "cereza"];
```

**Usando el constructor new Array():**

```
let numeros = new Array(1, 2, 3, 4, 5);
```

## Acceso a Elementos y Modificación de elementos

El acceso a los elementos de un array se realiza principalmente a través de **índices**. Cada elemento en un array tiene un índice asociado, comenzando con 0 para el primer elemento.



```
let primerFruta = frutas[0]; // "manzana"
frutas[1] = "mango"; // Cambia "plátano" por "mango"
```

## Otra forma de ver en la consola un arreglo .table

```
let frutas = ["pera", "naranja", "manzana", "platano", "cereza"];
console.table(frutas);
```

(índice)	Valor
0	'pera'
1	'naranja'
2	'manzana'
3	'platano'
4	'cereza'

▼ Array(5) [ ]

- 0: "pera"
- 1: "naranja"
- 2: "manzana"
- 3: "platano"
- 4: "cereza"

  length: 5

► [[Prototype]]: Array(0)

## Recorrer un Array →Iterador

```
let frutas = ["pera", "naranja", "manzana", "platano", "cereza"];
for(let i=0; i < frutas.length; i++){
  console.log(i);
}
```

0
1
2
3
4

```
let frutas = ["pera", "naranja", "manzana", "platano", "cereza"];
for(let i=0; i < frutas.length; i++){
  console.log(frutas[i]);
}
```

pera
naranja
manzana
platano
cereza

## Métodos Comunes de Arrays

Los arrays en JavaScript tienen una variedad de métodos útiles:

**push():** Añade uno o más elementos al final del array.

```
let frutas = ["manzana", "platano", "cereza"];
frutas.push("naranja")
```

```
▼ Array(4) i
  0: "manzana"
  1: "platano"
  2: "cereza"
  3: "naranja"
  length: 4
▶ [[Prototype]]: Array(0)
```

**pop():** Elimina el último elemento del array y lo devuelve (si se asigna a una variable).

```
let frutas = ["manzana", "platano", "cereza"];
let ultimaFruta = frutas.pop()
```

```
► (2) ['manzana', 'platano']
cereza
```

**shift():** Elimina el primer elemento del array y lo devuelve (si se asigna a una variable).

```
let frutas = ["manzana", "platano", "cereza"];
let primerFruta = frutas.shift()
```

```
▼ (2) ['platano', 'cereza'] i
  0: "platano"
  1: "cereza"
  length: 2
▶ [[Prototype]]: Array(0)
manzana
```

**unshift():** Añade uno o más elementos al principio del array.

```
let frutas = ["manzana", "platano", "cereza"];
frutas.unshift("pera", "duraznos")
```

```
▼ (5) ['pera', 'duraznos', 'manzana', 'platano', 'cereza'] i
  0: "pera"
  1: "duraznos"
  2: "manzana"
  3: "platano"
  4: "cereza"
  length: 5
▶ [[Prototype]]: Array(0)
```

**slice():** Devuelve una copia de una parte del array.

```
let frutas = ["pera", "naranja", "manzana", "platano", "cereza"];
let frutasSeleccionadas = frutas.slice(0,3)

console.log(frutasSeleccionadas)
```

```
▼ (3) ['pera', 'naranja', 'manzana']
  0: "pera"
  1: "naranja"
  2: "manzana"
  length: 3
  ► [[Prototype]]: Array(0)
```

**splice():** Cambia el contenido de un array eliminando, reemplazando o añadiendo elementos.

```
let frutas = ["pera", "naranja", "manzana", "platano", "cereza"];
```

### Eliminar Elementos

Supongamos que quieres eliminar "manzana" y "platano" del array

```
frutas.splice(2, 2); // Elimina 2 elementos a partir del índice 2
```

El **primer argumento** (2) es el **índice desde donde comenzamos a eliminar** (donde 0 es el primer elemento), y el **segundo argumento** (2) es el **número de elementos a eliminar**.

```
["pera", "naranja", "cereza"]
```

### Añadir Elementos

Ahora, digamos que quieres añadir "melón" y "mango" después de "naranja".

```
frutas.splice(2, 0, "melón", "mango");
```

```
// Añade "melón" y "mango" a partir del índice 2
```

Aquí, el primer argumento (2) es la posición donde empezamos a añadir, el **segundo argumento (0)** indica que no se elimina ningún elemento, y los argumentos restantes son los elementos que queremos añadir.

```
["pera", "naranja", "melón", "mango", "cereza"]
```

## Reemplazar Elementos

Si queremos reemplazar "naranja" por "kiwi", podríamos hacerlo de la siguiente manera:

```
frutas.splice(1, 1, "kiwi");
```

```
// Reemplaza i elemento a partir del índice i con "kiwi"
```

En este caso, **el primer argumento (1) es el índice donde comienza el reemplazo, el segundo argumento (1) es el número de elementos a reemplazar**, y "kiwi" es el elemento que se añade en lugar del eliminado.

```
["pera", "kiwi", "melón", "mango", "cereza"]
```

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

```
let frutas = ["pera", "naranja", "manzana", "platano", "cereza"];
frutas.forEach(function(item) {
  console.log(item + " 1.0");
});
```

pera	1.0
naranja	1.0
manzana	1.0
platano	1.0
cereza	1.0

**map(): Crea un nuevo array** (es la diferencia con forEach) con el resultado de la llamada a una función proporcionada **en cada elemento**.

```
let frutas = ["pera", "naranja", "manzana", "platano", "cereza"];
let longitudes = frutas.map(function(item) {
  return item.length;
});
console.log(longitudes);
```

▼ (5) [4, 7, 7, 7, 6] ↴
0: 4
1: 7
2: 7
3: 7
4: 6
length: 5
▶ [[Prototype]]: Array(0)

**filter():** Crea un nuevo array con todos los elementos que cumplen una condición.

```
let frutas = ["pera", "naranja", "manzana", "platano", "cereza"];
let frutasLargas = frutas.filter(function(item) {
  return item.length > 6;
});
```

▼ (3) ['naranja', 'manzana', 'platano'] ↴
0: "naranja"
1: "manzana"
2: "platano"
length: 3
▶ [[Prototype]]: Array(0)

**find() y findIndex():** Encuentran un elemento y su índice, respectivamente, que cumpla una condición.

```
let frutas = ["pera", "naranja", "manzana", "platano", "cereza"];
let primeraFrutaLarga = frutas.find(function(item) {
    return item.length > 5;
});
let indice = frutas.findIndex(function(item) {
    return item.length > 5;
});

console.log(primeraFrutaLarga);
console.log(indice);
```

naranja  
 1

**sort():** Ordena los elementos de un array.

```
let frutas = ["pera", "naranja", "manzana", "platano", "cereza"];
frutas.sort();
console.log(frutas);
```

```
▼ (5) ['cereza', 'manzana', 'naranja', 'pera', 'platano']
  0: "cereza"
  1: "manzana"
  2: "naranja"
  3: "pera"
  4: "platano"
  length: 5
  ▶ [[Prototype]]: Array(0)
```

**reduce():** Reduce los elementos de un array a un único valor.

```
let numeros = [1, 3, 4, 90, 87];
let suma = numeros.reduce(function(total, valor) {
    return total + valor;
}, 0);
console.log(suma);
```

185

**some():** Este método **prueba si al menos uno de los elementos en el array (u objeto) cumple con una condición especificada en una función de prueba proporcionada.** Retorna true si al menos un elemento del array satisface la condición; de lo contrario, retorna false. No modifica el array original.

La sintaxis básica de .some() es la siguiente:

```
let resultado = array.some(function(elemento, indice, arr) {
    // Condición a comprobar
    return condicion;
});
```

**Donde:**

**array:** Es el array sobre el cual se llama el método .some().

**function(elemento, indice, arr):** Es la función de prueba que se ejecuta en cada elemento del array. Esta función puede recibir hasta tres argumentos:

**elemento:** El elemento actual del array que se está procesando.

**indice (opcional):** El índice del elemento actual en el array.

**arr (opcional):** El array sobre el que se está llamando .some().

El método .some() es especialmente útil para comprobar si un array contiene al menos un elemento que cumple con una condición específica, como se muestra en el siguiente ejemplo:

Ejemplo

```
let numeros = [1, 2, 3, 4, 5];

let hayMayorQueTres = numeros.some(function(numero) {
    return numero > 3;
});

console.log(hayMayorQueTres); // true, porque hay elementos (4 y 5)
```

En este caso, .some() comprueba si hay algún número en el array números que sea mayor que 3. Tan pronto como encuentra un elemento que cumple con la condición (en este caso, 4), retorna true.

**Es importante notar que .some() no continúa procesando los elementos restantes una vez que encuentra un elemento que cumple con la condición.** Esto lo hace eficiente, especialmente para arrays grandes.

**includes()** es utilizado para **determinar si un array contiene un determinado elemento, retornando true si lo encuentra y false si no.** Es una manera simple y directa de comprobar la presencia de un elemento en un array, y es muy útil para condicionales y validaciones.

La sintaxis básica de .includes() es:

```
let resultado = array.includes(elementoABuscar, posicionInicial);
```

**Donde:**

**array:** Es el array sobre el cual se llama el método .includes().

**elementoABuscar:** Es el elemento que deseas buscar en el array.

**posicionInicial (opcional):** Es el índice en el cual empezar la búsqueda dentro del array. Si no se proporciona, la búsqueda empieza desde el inicio del array.

Es importante notar que .includes() hace una comparación de igualdad estricta (==) con los elementos del array.

**Ejemplo**

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

let contieneBanana = frutas.includes('banana');
console.log(contieneBanana); // true

let contieneMango = frutas.includes('mango');
console.log(contieneMango); // false
```

En este ejemplo, .includes() se utiliza para comprobar si el array frutas contiene 'banana' y 'mango'. Retorna true para 'banana', ya que es un elemento del array, y false para 'mango', que no está en el array.

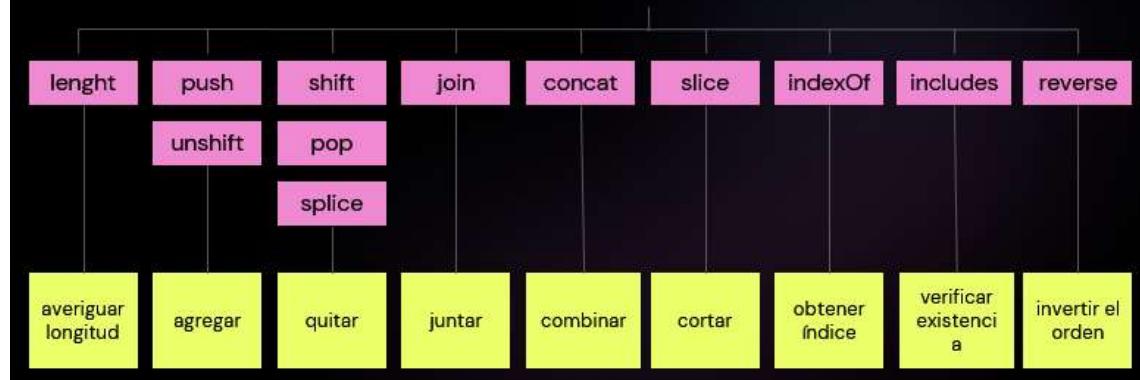
## Propiedades de Arrays

**length:** Devuelve o establece el número de elementos en el array.

```
let frutas = ["pera", "naranja", "manzana", "platano", "cereza"];
let cantidad = frutas.length;
console.log(cantidad)
```

5

## Métodos y propiedades más comunes



## Arrays Multidimensionales

Los arrays pueden contener otros arrays, formando estructuras multidimensionales (como matrices):

```
let matriz = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]
];
```

## Arrays y el Operador Spread

**El operador spread (...)** puede ser útil para copiar y combinar arrays:

```
let frutas = ["pera", "naranja", "manzana", "platano", "cereza"];
let numeros =[50,30,90,100,1200];

let combinado = [...frutas, ...numeros]
console.table(combinado)
```

(índice)	Valor
0	'pera'
1	'naranja'
2	'manzana'
3	'platano'
4	'cereza'
5	50
6	30
7	90
8	100
9	1200
▶ Array(10)	

## El destructuring de arrays (Desempaquetamiento)

El destructuring de arrays en JavaScript es una expresión que te permite **desempaquetar valores de arrays, o propiedades de objetos, en variables distintas**. Esto puede hacer que tu código sea más legible y menos propenso a errores, además de reducir la necesidad de acceder a elementos del array por sus índices.

```
let frutas = ["pera", "naranja", "manzana", "platano", "cereza"];
```

## Ejemplo Básico de Destructuring

Si quieres asignar los primeros dos elementos de este array a variables, puedes hacerlo así:

```
let [primeraFruta, segundaFruta] = frutas;
console.log(primeraFruta); // "pera"
console.log(segundaFruta); // "naranja"
```

## Ignorando Elementos

Si solo te interesan la primera y la tercera fruta, puedes "saltarte" la segunda usando una coma:

```
let [primera, , tercera] = frutas;
console.log(primera); // "pera"
console.log(tercera); // "manzana"
```

## Destructuring con Resto de Elementos

También puedes usar el operador de propagación (...) para agrupar el resto de los elementos en otro array:

```
let [primera, ...resto] = frutas;
console.log(primera); // "pera"
console.log(resto); // ["naranja", "manzana", "platano", "cereza"]
```



El **modo "use strict"** en JavaScript es una forma de optar por una versión más restringida y segura de JavaScript. Al usar "use strict", puedes evitar ciertos comportamientos potencialmente problemáticos y errores comunes en JavaScript, haciendo que tu código sea más robusto y confiable.

Aquí están algunas de las características y efectos de usar "use strict":

**Prevención de Variables Globales No Intencionadas:** Sin "use strict", asignar un valor a una variable no declarada crea automáticamente una variable global. Con "use strict", esto produce un error, ayudando a evitar contaminar el espacio global accidentalmente.

```
"use strict";
x = 3.14; // Esto causará un error porque x no está declarada
```

**Asignaciones a Propiedades no Escriturales y No Existentes:** En modo estricto, se generará un error al intentar asignar valores a propiedades no escribibles o no existentes.

**Uso de Palabras Reservadas:** "use strict" prohíbe utilizar ciertas palabras reservadas como nombres de variables o funciones.

**Eliminación de Variables, Funciones y Argumentos:** En modo estricto, no puedes usar el operador delete en variables, funciones o nombres de argumentos.

**Cambios en la Semántica de this:** En funciones no vinculadas (no-method functions), this ya no se referirá al objeto global. En su lugar, this será undefined, lo que ayuda a prevenir modificaciones accidentales del objeto global.

**Argumentos Duplicados en Funciones:** No se permiten nombres de parámetros duplicados en las declaraciones de funciones.

**Restricciones en eval y arguments:** Palabras clave eval y arguments no pueden ser usadas como variables o asignadas a otro valor. Además, el comportamiento de eval es más restrictivo.

**Para activar el modo estricto**, simplemente coloca la cadena "use strict"; al principio de tu archivo JavaScript o función:

```
"use strict";

function miFuncion() {
    // Código aquí se ejecutará en modo estricto
}
```

El modo estricto es una herramienta poderosa para mejorar la calidad del código JavaScript, y es altamente recomendable para evitar errores sutiles y promover prácticas de codificación seguras.





**El operador de propagación (spread operator), denotado por tres puntos (...),** es una característica muy útil en JavaScript, introducida en ES6 (ECMAScript 2015). **Este operador permite expandir los elementos de un iterable** (como un array o un string) **en lugares donde se esperan múltiples argumentos** (en llamadas a funciones) **o elementos** (en arrays) **o pares clave-valor** (en objetos). Aquí te muestro algunos usos comunes del operador spread:

### En Arrays

#### Combinar Arrays:

```
const arr1 = [1, 2, 3];
const arr2 = [4, 5, 6];
const combinado = [...arr1, ...arr2]; // [1, 2, 3, 4, 5, 6]
```

```
const original = [1, 2, 3];
const copia = [...original]; // [1, 2, 3]
```

#### Convertir un String en un Array:

```
const str = "hola";
const letras = [...str]; // ['h', 'o', 'l', 'a']
```

### En Objetos

#### Combinar Objetos:

```
const obj1 = { a: 1, b: 2 };
const obj2 = { b: 3, c: 4 };
const combinado = { ...obj1, ...obj2 }; // { a: 1, b: 3, c: 4 }
```

Nota: Si hay propiedades duplicadas, el último objeto tiene prioridad.

#### Copiar un Objeto:

```
const original = { a: 1, b: 2 };
const copia = { ...original }; // { a: 1, b: 2 }
```

### En Funciones

El operador spread es útil para pasar un array de argumentos a una función que espera argumentos separados:

```
function sumar(x, y, z) {
    return x + y + z;
}

const numeros = [1, 2, 3];

console.log(sumar(...numeros)); // 6
```



## FUNCTION (FUNCIONES)

**Son bloques de código diseñados para realizar una tarea específica.** Son fundamentales en JavaScript para organizar y reutilizar código. Las funciones pueden ser invocadas (llamadas) tantas veces como sea necesario dentro de un programa.

### Declaración de una Función → Function Declaration → Creación

Una función se declara con la **palabra reservada function**, seguida de un **nombre**, un conjunto de **paréntesis que pueden contener parámetros**, y un **bloque de código encerrado entre llaves {}**.

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

### Expresión de Función → Function expression → Creación

**Una "function expression" es una forma de definir funciones.** A diferencia de una declaración de función, donde se especifica la función y su nombre al comienzo, **una expresión de función asigna la función**

```
const miFuncion = function(parametro1, parametro2) {
    return parametro1 + parametro2;
};
```

**a una variable.** Esto puede ser útil para definir funciones de forma anónima, para funciones de callback, y más. Aquí tienes un ejemplo básico:

Las expresiones de función **son particularmente útiles porque pueden ser anónimas y pueden definirse en lugares donde se necesitan valores, como en asignaciones de variables o como argumentos para otras funciones**. Además, **las expresiones de función tienen un alcance local y pueden ayudar a evitar contaminar el ámbito global**, especialmente cuando se usan dentro de otros bloques de código.

### Invocar (Llamar) una Función → Ejecución

Para **ejecutar una función**, la llamas por su **nombre seguido de paréntesis**.

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

## Parámetros y Argumentos de Función

### Parámetros

Los parámetros son las variables que se enumeran como parte de la definición de una función. Son como placeholders (espacios reservados) que definen los tipos de valores que la función espera recibir cuando se invoca. Los parámetros se definen en la declaración de la función y actúan como variables locales dentro de esa función.

### Argumentos

Los argumentos, por otro lado, son los valores reales que se pasan a la función cuando se invoca. Estos son los valores concretos que se asignan a los parámetros de la función cuando se ejecuta.

Resumen

**Parámetros: Variables** en la definición de una función.

**Argumentos: Valores reales** pasados a la función cuando se llama.



## Parámetros y Argumentos por default

Puedes definir parámetros por defecto en las funciones para los **casos en los que no se proporcionen argumentos específicos al llamar a la función**. Esto es especialmente útil para evitar errores cuando se espera que ciertos argumentos estén presentes y no lo están.

```

function saludar(nombre = "NombrePorDefecto",
apellido = "ApellidoPorDefecto") {
return `Hola, ${nombre} ${apellido}!`;
}

console.log(saludar("Juan", "Pérez"));
console.log(saludar("Laura"));
console.log(saludar()); Hola, Juan Pérez!
                                Hola, Laura ApellidoPorDefecto!
                                Hola, NombrePorDefecto ApellidoPorDefecto!

```

## Retorno de Valores → return

Una función puede **devolver un valor (hacia afuera de la función)** usando la palabra clave `return`. **¿Hacia donde se retorna?, hacia la invocación.**

```

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

let resultado = sumar(5, 3); // resultado es 8

```



**Si quiero utilizar el valor devuelto con `return` fuera de la función, debo asignarlo a una variable.**

## Funciones Anónimas y Expresiones de Función

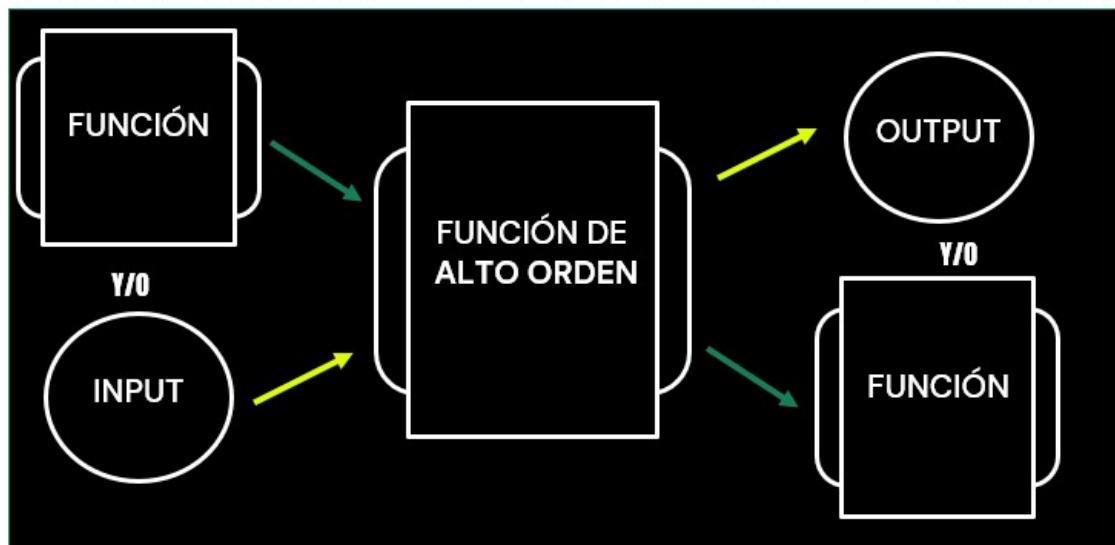
Las funciones **no necesariamente deben tener un nombre**; pueden ser anónimas y asignarse a variables.

```
let multiplicar = function(a, b) {
    return a * b;
};

console.log(multiplicar(3, 4)); // 12
```

## Funciones de Orden Superior

# Funciones de orden superior



**Las funciones pueden aceptar otras funciones como argumentos y pueden ser devueltas por otras funciones.** Esto permite patrones como funciones de orden superior y funciones de callback.

```
// Primero, definimos la función sumar
function sumar(a, b) {
    return a + b;
}

// Luego, definimos la función de orden superior que utiliza la función sumar
function aplicarOperacion(a, b, operacion) {
    return operacion(a, b);
}

// Ahora podemos llamar a aplicarOperacion con sumar como argumento
let resultado = aplicarOperacion(10, 20, sumar);
console.log(resultado); // Debería mostrar 30
```

## Comunicando funciones

La comunicación entre funciones en JavaScript **es un concepto clave en la programación y se puede realizar de varias maneras**. Aquí te muestro algunos métodos comunes para "comunicar" funciones entre sí:

**Esto es muy útil para no crear una función gigante sino pequeñas funciones que luego sean llamadas por otra.**

### Llamando a una Función Dentro de Otra Función:

Puedes llamar a una función desde otra y pasarle datos como argumentos.

Esto es útil cuando una función depende de los resultados o acciones de otra.

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

function muestraSuma(x, y) {
    const resultado = suma(x, y);
    console.log('La suma es:', resultado);
}

muestraSuma(5, 3); // La suma es: 8
```

### Retornando Valores:

Una función puede retornar un valor, que luego puede ser utilizado por otra función.

Este es un enfoque muy común para obtener datos de una función.

```
function obtenerSaludo(nombre) {
    return `Hola, ${nombre}!`;
}

const saludo = obtenerSaludo("Ana");
console.log(saludo); // Hola, Ana!
```

### Callbacks:

Un callback es una función que se pasa a otra función como argumento y que luego se ejecuta dentro de esa función.

Los callbacks son muy utilizados en operaciones asíncronas.

```

function procesoAsincrono(callback) {
    setTimeout(() => {
        console.log("Proceso completado");
        callback();
    }, 1000);
}

procesoAsincrono(() => {
    console.log("Callback ejecutado");
});

```

### **Funciones Anidadas (Closures):**

Las funciones anidadas pueden acceder a las variables de las funciones en las que están contenidas.

Esto permite crear funciones especializadas basadas en ciertos contextos.

```

function crearSaludador(saludo) {
    return function(nombre) {
        console.log(`${saludo}, ${nombre}!`);
    }
}

const saludarHola = crearSaludador("Hola");
saludarHola("Carlos"); // Hola, Carlos!

```

### **Promesas y Async/Await (para operaciones asíncronas):**

Las promesas y async/await permiten manejar respuestas asíncronas de una manera más limpia y estructurada.

```

function obtenerDatosAsincronos() {
    return new Promise((resolve) => {
        setTimeout(() => {
            resolve("Datos recibidos");
        }, 1000);
    });
}

async function manejarDatos() {
    const datos = await obtenerDatosAsincronos();
    console.log(datos);
}

manejarDatos(); // Después de 1 segundo: Datos recibidos

```

## Funciones nativas de JS → Son más de 4000

JavaScript tiene una amplia gama de **funciones nativas, también conocidas como funciones integradas o built-in functions**. Estas funciones están disponibles en el entorno de ejecución de JavaScript y proporcionan funcionalidades esenciales sin la necesidad de definirlas por separado. Aquí te presento algunas de las funciones nativas más comunes y útiles:

### **Funciones de Manipulación de Cadenas:**

charAt(), concat(), includes(), indexOf(), lastIndexOf(), match(), replace(), search(), slice(), split(), substring(), toLowerCase(), toUpperCase(), trim()

### **Funciones de Manipulación de Arrays:**

concat(), every(), filter(), find(), findIndex(), forEach(), indexOf(), join(), map(), pop(), push(), reduce(), reduceRight(), reverse(), shift(), slice(), some(), sort(), splice(), unshift()

### **Funciones de Manipulación de Números y Matemáticas:**

isNaN(), isFinite(), parseFloat(), parseInt()

En el objeto Math: abs(), ceil(), floor(), max(), min(), pow(), random(), round(), sqrt()

### **Funciones de Manejo de Fechas:**

Métodos del objeto Date, como getDate(), getDay(), getMonth(), getFullYear(), setDate(), setMonth(), setFullYear()

### **Funciones de Codificación y Decodificación:**

encodeURI(), encodeURIComponent(), decodeURI(), decodeURIComponent()

### **Funciones de Temporización:**

setTimeout(), setInterval(), clearTimeout(), clearInterval()

### **Funciones de Consola para Depuración:**

console.log(), console.error(), console.warn(), console.info(), console.table(), console.group(), console.groupEnd()

### **Funciones de Manejo de Errores:**

try...catch, throw

### **Funciones JSON:**

JSON.parse(), JSON.stringify()

### **Funciones de Evaluación:**

Eval

## Funciones Flecha (Arrow Functions)

`( ) => {}`

Introducidas en ES6, las arrow functions ofrecen una sintaxis más corta para declarar funciones y this ligado léxicamente.

### Función Regular:

```
function sumarRegular(nro1, nro2) {
    return nro1 + nro2;
}

// Ejemplo de uso
const resultadoRegular = sumarRegular(5, 3);
console.log(resultadoRegular); // 8
```

### Arrow Function:

```
const sumarArrow = (nro1, nro2) => {
    return nro1 + nro2;
};

// Ejemplo de uso
const resultadoArrow = sumarArrow(10, 4);
console.log(resultadoArrow); // 14
```

### O así

```
const sumarArrow = (nro1, nro2) => nro1 + nro2;
```

El return se da por implícito si la función tiene una sola línea

```
// Ejemplo de uso
const resultadoArrow = sumarArrow(10, 4);
console.log(resultadoArrow); // 14
```

## Ventajas de Arrow Functions

Las arrow functions ofrecen varias ventajas y características únicas en comparación con las funciones tradicionales. Aquí te detallo algunas de las ventajas más significativas:

### **Sintaxis más Concisa:**

Las arrow functions tienen una sintaxis más corta y clara, lo que las hace más legibles, especialmente para funciones pequeñas o funciones que se usan como callbacks.

### **No tienen su propio this:**

A diferencia de las funciones tradicionales, las arrow functions no tienen su propio contexto this. En su lugar, capturan el valor de this del ámbito en el que fueron definidas.

Esto es útil en ciertos patrones de programación, como cuando se trabaja con métodos de clase que necesitan acceder al this de la clase dentro de un callback.

### **No tienen su propio arguments:**

Las arrow functions no tienen su propio objeto arguments. Si necesitas acceder a los argumentos pasados a la función, puedes usar parámetros **rest (...args)** para obtener un array de argumentos.

### **Más adecuadas para funciones inline y callbacks:**

Su sintaxis concisa las hace ideales para uso en funciones inline y callbacks, como en métodos de array como map, filter, y reduce.

### **No aptas para métodos de objetos o funciones constructoras:**

Debido a que no tienen su propio this, las arrow functions no son adecuadas como métodos en objetos si necesitas acceder al objeto a través de this.

### **Tampoco pueden ser usadas como funciones constructoras.**

No tienen su propio new.target:

Las arrow functions no tienen new.target, lo que significa que no puedes identificar si fueron llamadas con new.

### **Sintaxis más limpia para el retorno implícito:**

Las arrow functions permiten un retorno implícito de valores cuando se omite el cuerpo de la función con llaves, lo cual es útil para operaciones que requieren una devolución directa de expresiones.

### **Más fáciles de anidar:**

La claridad y simplicidad de la sintaxis de las arrow functions las hacen más fáciles de anidar y entender en situaciones complejas.

En resumen, las arrow functions ofrecen una sintaxis más limpia y una forma más intuitiva de manejar el contexto this, lo que las hace adecuadas para ciertos patrones de programación en JavaScript moderno, especialmente en operaciones que involucran callbacks y métodos de arrays.

## Añadir funciones en un objeto → Métodos

**Los objetos pueden tener propiedades que son funciones.** Estas funciones se conocen como métodos del objeto. Puedes añadir métodos a un objeto de varias maneras.

### Definición Directa en el Objeto:

Puedes definir un método directamente dentro de la literal de un objeto.

```
const miObjeto = {
    nombre: "Ejemplo",
    saludar: function() {
        console.log("Hola! Soy " + this.nombre);
    }
};

miObjeto.saludar(); // Hola! Soy Ejemplo
```

**Se accede a la función (método) mediante el punto.**



**El "hoisting"** es un comportamiento en JavaScript

donde las declaraciones de variables y funciones son movidas al inicio de su ámbito antes de que se ejecute el código. Este comportamiento puede parecer como si estas declaraciones fueran "elevadas" o "izadas" al inicio de su ámbito (de ahí el término "hoisting", que significa "izar" o "elevar" en inglés).

Esto es así porque JS se ejecuta “en dos vueltas” al leer el código lo lee dos veces, **en dos etapas: etapa de creación y etapa de ejecución.**

En JavaScript, la “**etapa de creación**” se refiere a la primera fase del ciclo de vida de ejecución de un script o función, **antes de que el código sea efectivamente ejecutado**. Durante esta etapa, el motor de JavaScript realiza una serie de pasos preparatorios, lo que incluye el hoisting (izar) de declaraciones de funciones y variables.

Tras la etapa de creación, sigue la “**etapa de ejecución**”, donde el código es ejecutado línea por línea. Es importante entender estas etapas, especialmente el hoisting y cómo las variables y funciones son creadas e inicializadas, ya que esto puede afectar significativamente el comportamiento del código y es una fuente común de errores para los programadores menos experimentados en JavaScript.

### Aquí hay algunos puntos clave sobre el hoisting:

**VARIABLES:** En el caso de las variables **declaradas con var**, el hoisting eleva la declaración de la variable al inicio de su ámbito, pero no su inicialización. Si intentas acceder a la variable antes de que se asigne un valor, resultará en undefined.

```
console.log(miVar); // undefined
var miVar = 5;
console.log(miVar); // 5
```

Aquí, la declaración `var miVar` es izada al inicio del ámbito, pero su asignación `miVar = 5` permanece en su lugar original.

**FUNCIONES:** En el caso de las declaraciones de funciones, tanto la declaración como la definición son izadas. Esto significa que puedes llamar a una función antes de que aparezca su definición en el código.

**LET Y CONST:** Las variables declaradas con let y const también son izadas a la parte superior de su ámbito. Sin embargo, no se inicializan con undefined como lo hacen las variables var. En lugar de eso, entran en una "zona temporal muerta" desde el inicio del bloque hasta que se inicializan. Acceder a ellas antes de su declaración resulta en un error ReferenceError.

En JavaScript, el término "ámbito" (o "scope" en inglés) se refiere al contexto en el cual las variables, funciones y expresiones existen y pueden ser accedidas o referenciadas. El ámbito determina la visibilidad y la vida útil de variables y funciones en diferentes partes de tu código. Hay varios tipos de ámbitos en JavaScript:

**Ámbito Global:** Cualquier variable o función definida en el nivel más alto de un documento JavaScript está en el ámbito global. Esto significa que son accesibles desde cualquier otra parte del código, a menos que estén ocultas por una definición en un ámbito más local.

**Ámbito de Función:** significa que cuando defines una variable o función dentro de otra función, esta está accesible únicamente dentro de esa función y no fuera de ella. Este tipo de ámbito es creado por la declaración de funciones y por variables declaradas con var.

**Ámbito de Bloque:** Introducido en ES6 (ECMAScript 2015), el ámbito de bloque significa que las variables definidas dentro de un

**bloque** (por ejemplo, dentro de un **if, for, while, o un bloque {}**) usando **let o const** son accesibles solo dentro de ese bloque y no fuera de él.

**El concepto de "niveles" en un documento se refiere a la jerarquía de ámbitos (scopes) desde el global hasta el más local.** Aquí está una descripción de estos niveles desde el más alto (más global) hasta el más bajo (más local):

### **1. Ámbito Global:**

**Este es el nivel más alto en cualquier documento JavaScript.**

Las variables y funciones definidas en el ámbito global son accesibles desde cualquier parte del código.

**Puedes definir una variable global simplemente declarándola fuera de cualquier función o bloque.**

En un navegador, el objeto window representa el ámbito global. Todo lo que se define globalmente se convierte en una propiedad de window.

### **2. Ámbito de Archivo o Módulo (en módulos ES6):**

En módulos ES6 (cuando usas import y export en un sistema de módulos), cada archivo tiene su propio ámbito.

Las variables y funciones definidas en un módulo no son accesibles globalmente a menos que se exporten y se importen explícitamente.

### **3. Ámbito de Función:**

Este ámbito es creado cada vez que una función es definida.

Las variables y funciones definidas dentro de una función no pueden ser accedidas fuera de ella.

Este concepto es fundamental para la programación en JavaScript y es utilizado para encapsular variables y funciones, limitando su acceso al ámbito local de la función.

### **4. Ámbito de Bloque:**

Introducido en ES6 con let y const.

Las variables definidas con let y const dentro de bloques (por ejemplo, dentro de un if, for, while, o un bloque {}) son accesibles solo dentro de ese bloque y no fuera de él.

Este ámbito es más restrictivo que el ámbito de función.

## 5. Ámbito de Evaluación de Expresiones (como en una expresión with o catch):

Son ámbitos aún más locales que se crean en ciertas construcciones como catch en bloques try...catch o el obsoleto with.

Por ejemplo, en un bloque catch, el parámetro del catch solo está disponible dentro de ese bloque.

En resumen, el ámbito global es el nivel más alto, accesible desde cualquier parte del código, mientras que los ámbitos de bloque y de función son más locales, restringiendo la accesibilidad a las variables y funciones a bloques o funciones específicas. Los módulos ES6 añaden otra capa de ámbito donde lo que se define en un módulo permanece local a ese módulo a menos que se exporte explícitamente.



## ESTRUCTURAS DE CONTROL

### CONDICIONALES

(dependen del cumplimiento de una condición)

↓ Las estructuras de control en JavaScript son fundamentales para dirigir el flujo de ejecución del código. El código se ejecuta de arriba abajo.



Siguiendo el flujo que dictan las estructuras de control como if, else if, y else. La declaración if juega un papel crucial en este flujo, permitiendo que se ejecute un código específico solo si se cumple una condición dada.

### IF...ELSE IF...ELSE:

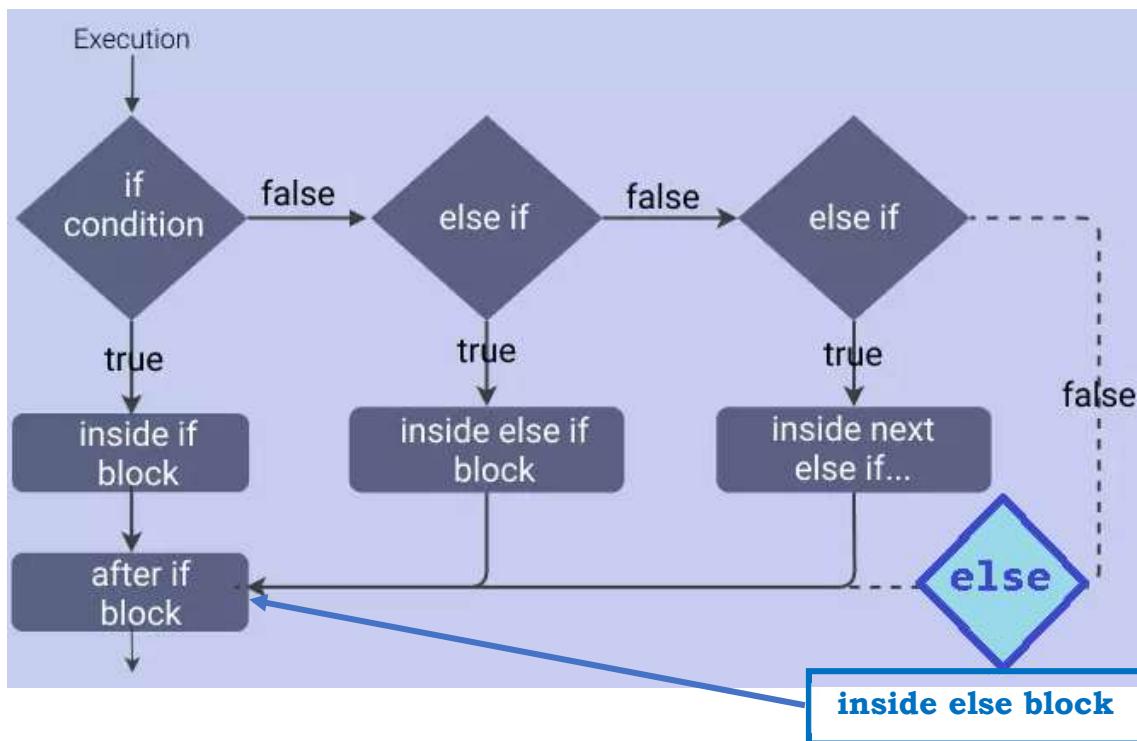
**if:** Permite ejecutar un bloque de código si y solo si se cumple una condición específica.

Si la condición se cumple (es decir, si su valor es true) se ejecutan todas las instrucciones que se encuentran dentro de {...}. Si la condición no se cumple (es decir, si su valor es false) no se ejecuta

ninguna instrucción contenida en {...} y el programa continúa ejecutando el resto de las instrucciones del script.

**else if:** Proporciona una **nueva condición a evaluar si la anterior (o anteriores) no se cumplen para ejecutar algún bloque de código si esta se cumple.**

**else:** Se ejecuta **si y solo si ninguna condición if o if else anterior se cumple.**



## Sintaxis

```

if (condicion1) {
  // Código a ejecutar si condicion1 es verdadera
} else if (condicion2) {
  // Código a ejecutar si condicion1 es falsa pero condicion2 es verdadera
} else {
  // Código a ejecutar si ni condicion1 ni condicion2 son verdaderas
}
  
```

```

let numero = 50;

if (numero > 10) {
  console.log("El número " + numero + " es mayor a 10");
} else if (numero < 10) {
  console.log("El número " + numero + " es menor a 10");
} else {
  console.log("El número es 10");
}
  
```

El número 50 es mayor a 10

```
let numero = 8;

if (numero > 10) {
    console.log("El número " + numero + " es mayor a 10");
} else if (numero < 10) {
    console.log("El número " + numero + " es menor a 10");
} else {
    console.log("El número es 10");
}
```

El número 8 es menor a 10

```
let numero = 10;

if (numero > 10) {
    console.log("El número " + numero + " es mayor a 10");
} else if (numero < 10) {
    console.log("El número " + numero + " es menor a 10");
} else {
    console.log("El número es 10");
}
```

El número es 10

Ante una combinación de **operadores && (AND)** será requisito que todas las comparaciones sean verdaderas para que la condición compuesta sea verdadera.

```
let nombreIngresado = prompt("Ingresar nombre");
let apellidoIngresado = prompt("Ingresar apellido");

if((nombreIngresado != "") && (apellidoIngresado != "")){
    alert("Nombre: "+nombreIngresado +"\nApellido: "+apellidoIngresado);
}else{
    alert("Error: Ingresar nombre y apellido");
}
```

En caso de utilizar **|| (OR)**, será requisito que al menos una de las comparaciones sea verdadera para que la condición compuesta sea verdadera.

```
let nombreIngresado = prompt("Ingresar nombre");

if((nombreIngresado == "ANA") || (nombreIngresado == "ana")){
    alert("El nombre ingresado es Ana");
}else{
    alert("El nombre ingresado NO ES Ana");
}
```

También es posible combinar **|| (OR)** y **&& (AND)** para hacer comparaciones cada vez más complejas.

```
let nombreIngresado = prompt("Ingresar nombre");

if((nombreIngresado != "") && ((nombreIngresado == "EMA") || 
(nombreIngresado == "ema"))){
    alert("Hola Ema");
} else{
    alert("Error: Ingresar nombre valido");
}
```

**Las expresiones lógicas son evaluadas de izquierda a derecha, es necesario agrupar las operaciones para asegurar que se cumplan como uno lo desea.**

Repasando...

## Operadores en JS

OPERADORES LÓGICOS Y RELACIONALES	DESCRIPCIÓN	EJEMPLO
==	Es igual	a == b
===	Es estrictamente igual	a === b
!=	Es distinto	a != b
!==	Es estrictamente distinto	a !== b
<, <=, >, >=	Menor, menor o igual, mayor, mayor o igual	a <= b
&&	Operador and (y)	a && b
	Operador or (o)	a    b
!	Operador not (no)	!a

## Sintaxis ifs anidados

```
if (condicion1) {
    // Código a ejecutar si condicion1 es verdadera

    if (condicion2) {
        // Código a ejecutar si condicion1 y condicion2 son verdaderas
    } else {
        // Código a ejecutar si condicion1 es verdadera pero condicion2 no
    }
} else if (condicion3) {
    // Código a ejecutar si condicion1 es falsa pero condicion3 es verdadera
} else {
    // Código a ejecutar si ninguna de las condiciones anteriores se cumple
}
```

## Operador ternario

**Una forma abreviada del if...else**, que asigna un valor a una variable basado en una condición.

```
let resultado = condicion ? valorSiTrue : valorSiFalse;
```

```
let numero = 10;
let resultado = numero === 10 ? "El número es 10" : "El número es distinto a 10";
console.log(resultado)
```

El número es 10

```
let numero = 23;
let resultado = numero === 10 ? "El número es 10" : "El número es distinto a 10";
console.log(resultado)
```

El número es distinto a 10

## Switch

La estructura **switch** está especialmente **diseñada para manejar de forma sencilla múltiples condiciones sobre la misma variable**.

La declaración switch **es una forma de manejar múltiples casos o condiciones de manera más estructurada y legible que una serie de if...else if...else**. Es especialmente útil cuando tienes muchas condiciones que dependen del valor de una sola variable o expresión.

**La sintaxis básica de una declaración switch es la siguiente:**

```
switch (expresion) {
    case valor1:
        // Código a ejecutar cuando expresion es igual a valor1
        break;
    case valor2:
        // Código a ejecutar cuando expresion es igual a valor2
        break;
    // Puedes tener tantos casos como necesites
    default:
        // Código a ejecutar si ninguno de los casos anteriores se cumple
}
```

**Donde:**

**expresión:** Es la variable o expresión cuyo valor se compara en cada case.

**case:** Especifica un valor con el que se compara expresion. Si el valor coincide, se ejecuta el bloque de código asociado a ese case.

**break:** Este es un punto clave. El break detiene la ejecución dentro del switch, evitando que se ejecute el código de los casos siguientes. Sin break, el código de los siguientes case se ejecutará incluso si no se cumplen sus condiciones.

**default:** Este caso se ejecuta si ninguno de los case anteriores coincide con la expresion. Es similar a un else en una estructura if...else.

**Cada condición se evalúa y, si se cumple, se ejecuta lo que esté indicado dentro de cada case.**

Ejemplo:

```
let fruta = "cereza";

switch (fruta) {
  case "manzana":
    console.log("¡Manzanas rojas!");
    break;
  case "banana":
    console.log("¡Bananas amarillas!");
    break;
  case "cereza":
    console.log("¡Cerezas rojas!");
    break;
  default:
    console.log("Lo siento, no tenemos " + fruta + ".");
}
```

¡Cerezas rojas!

En este ejemplo, la consola mostrará "¡Cerezas rojas!", porque **fruta** coincide con el **tercer case**.

## ESTRUCTURAS DE CONTROL

### ITERATIVAS

**(repiten un bloque de código mientras se cumpla alguna condición)**

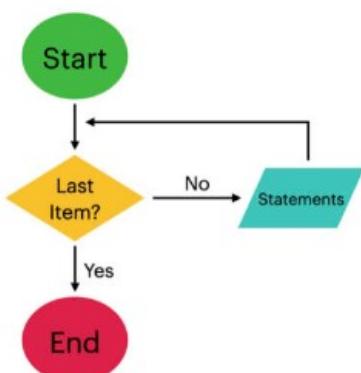


**Los ciclos, también conocidos como bucles (loops) o iteraciones** son un medio rápido y sencillo para hacer algo repetidamente.

Si tenemos que hacer alguna operación más de una vez en el programa, de forma consecutiva, **usaremos las estructuras de bucles de JavaScript: for, while o do...while.**

## Ciclos por conteo FOR LOOP o BUCLE FOR

### For Loop



Repite un bloque de código un número de veces específica. **Estructura for.**

Este bucle **se utiliza para repetir un bloque de código un número determinado de veces**. Se define con tres expresiones: una para inicializar la variable del bucle, una para definir la condición de continuidad del bucle, y una para actualizar la variable del bucle después de cada iteración

## Estructura Básica

El bucle for tiene la siguiente **sintaxis**:

```

for (inicialización; condición; actualización) {
    // Código a ejecutar en cada iteración
}
  
```

**Inicialización:** Es donde se establece el **valor inicial de la variable del bucle**. Normalmente, se utiliza para crear una variable de contador. Esta expresión se ejecuta solo una vez, al comienzo del bucle.

**Condición:** Esta expresión **se evalúa antes de cada iteración** del bucle. Si la condición es verdadera (true), el bucle continúa y el bloque de código dentro del bucle se ejecuta. Si la condición es falsa (false), el bucle termina.

**Actualización:** Esta expresión se ejecuta al final de cada iteración del bucle, antes de la próxima evaluación de la condición. Generalmente, se utiliza para actualizar el contador o la variable del bucle.

## Ejemplo Básico

```
for (let i = 0; i < 5; i++) {
    console.log(i); // Imprime 0, 1, 2, 3, 4
}
```

**Inicialización:** let i = 0 → la variable i se inicializa en 0.

**Condición:** i < 5 → el bucle se ejecutará mientras i sea menor que 5.

**Actualización:** i++ → incrementa el valor de i en 1 después de cada iteración.

## Iterar sobre Arrays:

```
const array = ['a', 'b', 'c', 'd'];
for (let i = 0; i < array.length; i++) {
    console.log(array[i]);
}
```

## Break y continue:

### Break

La declaración break **se utiliza para "romper" o terminar la ejecución del bucle**. Cuando se ejecuta break dentro de un bucle, el control del programa salta fuera del bucle, y el bucle se detiene inmediatamente, sin importar si la condición del bucle todavía se cumple o no.

```
for (let i = 0; i < 10; i++) {
    if (i === 5) {
        break; // Sale del bucle cuando i es igual a 5
    }
    console.log(i); // Imprime números de 0 a 4
}
```

En este ejemplo, el bucle for se detiene cuando i alcanza el valor de 5. La declaración break causa que el bucle termine y el control del programa continúe con las instrucciones que siguen después del bucle.

### Continue

La declaración continue **se utiliza para "saltarse" la iteración actual de un bucle y continuar con la siguiente iteración**. Cuando se

ejecuta `continue`, el control del programa salta al final de la iteración actual y luego procede con la siguiente evaluación de la condición del bucle.

```
for (let i = 0; i < 10; i++) {
    if (i === 5) {
        continue; // Salta la iteración cuando i es igual a 5
    }
    console.log(i); // Imprime números de 0 a 4 y de 6 a 9
}
```

En este ejemplo, cuando `i` es igual a 5, la declaración `continue` hace que el bucle salte la impresión de `console.log(i)` para ese valor específico. Por lo tanto, el número 5 no se imprime, pero el bucle no se detiene y continúa con las siguientes iteraciones.

### **Diferencias y Consideraciones**

**Break:** Termina el bucle por completo.

**Continue:** Solo salta la iteración actual y continúa con el bucle.

Es importante usar estas declaraciones con cuidado, ya que pueden hacer que el flujo de control de tu programa sea más difícil de seguir y entender, especialmente en bucles anidados o en estructuras de control complejas.

## **.forEach y .map**

**El método `.forEach()`** es una **forma de iterar sobre los elementos de un array**. Se utiliza para **ejecutar una función en cada elemento del array**, lo que lo hace útil para realizar operaciones o acciones sobre los elementos de una colección.

La sintaxis básica de `.forEach()` es:

```
array.forEach(function(elemento, indice, arr) {
    // Código a ejecutar para cada elemento del array
});
```

### **Donde:**

**array:** Es el array sobre el que se quiere iterar.

**function(elemento, indice, arr):** Es una función de callback que se ejecuta en cada elemento del array. Esta función puede recibir hasta tres argumentos:

**elemento:** El elemento actual del array que se está procesando.

**indice (opcional):** El índice del elemento actual en el array.

**arr (opcional):** El array sobre el que se está llamando .forEach().

### **Importante:**

.forEach() ejecuta la función de callback una vez para cada elemento del array, en orden.

No es utilizado para modificar el array original, aunque la función de callback puede modificar los elementos del array.

.forEach() no devuelve nada (undefined). Si necesitas crear un nuevo array basado en el original, métodos como .map() son más adecuados.

Ejemplo:

```
let numeros = [1, 2, 3, 4, 5];

numeros.forEach(function(numero) {
    console.log(numero);
});
```

Este código imprimirá cada número del array numeros en la consola. La función dentro de .forEach() se ejecuta para cada elemento del array, y en este caso, simplemente imprime el valor del elemento.

**El método .map()** es una herramienta poderosa y **comúnmente utilizada para transformar arrays**. Este método **crea un nuevo array con los resultados de llamar a una función proporcionada en cada elemento del array original**. Es especialmente útil cuando quieres aplicar una misma operación a todos los elementos de un array y obtener un nuevo array con los resultados.

La sintaxis básica de .map() es:

```
let nuevoArray = arrayOriginal.map(function(elemento, indice, arr) {
    // Transformación o cálculo con el elemento del array
    return nuevoElemento;
});
```

### **Donde:**

**arrayOriginal:** Es el array sobre el que se ejecuta .map().

**function(elemento, indice, arr):** Es una función de callback que se ejecuta en cada elemento del array original. Esta función puede recibir hasta tres argumentos:

**elemento:** El elemento actual del array que se está procesando.

**indice (opcional):** El índice del elemento actual en el array.

**arr (opcional):** El array sobre el que se está llamando .map().

**return nuevoElemento:** Dentro de la función de callback, se retorna el nuevo elemento que se agregará al newArray.

### **Importante:**

.map() no modifica el array original; crea un nuevo array con los resultados.

La longitud del nuevo array será igual a la longitud del array original.

.map() es inmensamente útil para realizar operaciones de transformación de datos.

Ejemplo:

```
let numeros = [1, 2, 3, 4, 5];

let cuadrados = numeros.map(function(numero) {
    return numero * numero;
});

console.log(cuadrados); // [1, 4, 9, 16, 25]
```

En este ejemplo, .map() toma el array numeros, y para cada elemento en él, calcula su cuadrado y construye un nuevo array cuadrados con estos valores. El array original numeros permanece sin cambios.

## for ..... of

**El bucle for...of** es una forma moderna y eficiente de **iterar sobre elementos iterables**, como arrays, strings, Mapas (Maps), Conjuntos (Sets), y otros objetos iterables. Este bucle **es útil para recorrer los elementos de una colección de forma sencilla y directa, accediendo al valor de cada elemento.**

La sintaxis del bucle for...of es la siguiente:

```
for (const elemento of iterable) {
    // Código a ejecutar para cada elemento
}
```

### **Donde:**

**iterable:** Es el objeto iterable (como un array o un string) que deseas recorrer.

**elemento:** Es una variable que representa el valor actual del elemento en la colección a medida que el bucle itera sobre ella.

A diferencia del bucle for...in, que itera sobre las claves o propiedades de un objeto, for...of se enfoca en los valores de los elementos de un iterable.

### Ejemplo

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

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

El bucle for...of recorre el array frutas. En cada iteración, fruta toma el valor de cada elemento del array ('manzana', 'banana', 'cereza'), y ese valor se imprime en la consola.

El bucle for...of es especialmente útil porque es más conciso y legible, especialmente en comparación con un bucle for tradicional, y evita algunos de los problemas comunes asociados con otras formas de iteración.

## for ..... in

**El bucle for...in se utiliza para iterar sobre todas las propiedades enumerables de un objeto.** Es diferente del bucle for...of, que se usa para iterar sobre los valores de un objeto iterable. El bucle for...in **es útil para recorrer las propiedades de un objeto y trabajar con sus claves (nombres de propiedades).**

La sintaxis del bucle for...in es la siguiente:

```
for (var propiedad in objeto) {
    // Código a ejecutar
}
```

### Donde:

**propiedad:** Es una variable que representa el nombre de la propiedad actual en el objeto durante la iteración.

**objeto:** Es el objeto cuyas propiedades se van a recorrer.

El bucle for...in iterará sobre todas las propiedades enumerables del objeto, incluyendo aquellas que han sido heredadas a través de la cadena de prototipos. Por esta razón, **es común (y recomendable) usar hasOwnProperty para filtrar las propiedades que son realmente parte del objeto**, excluyendo las heredadas:

```
for (var propiedad in objeto) {
    if (objeto.hasOwnProperty(propiedad)) {
        // Código a ejecutar con propiedad que es realmente parte de objeto
    }
}
```

## Ejemplo

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

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

En este ejemplo, el bucle `for...in` recorre cada propiedad del objeto `persona`. La consola mostrará algo como:

```
nombre: Juan
edad: 30
profesion: Ingeniero
```

Cada clave en el bucle se refiere a una propiedad del objeto `persona`, y `persona[clave]` accede al valor de esa propiedad. El uso de `hasOwnProperty` garantiza que solo se consideren las propiedades definidas directamente en el objeto `persona`, excluyendo las propiedades heredadas.

## Ejercicio "FizzBuzz"

**El ejercicio "FizzBuzz" es un problema clásico de programación** que se suele utilizar en entrevistas de trabajo. La tarea consiste en escribir un programa que imprima los números del 1 al 100, pero con las siguientes condiciones:

Para los números múltiplos de 3, imprime "Fizz" en lugar del número.

Para los números múltiplos de 5, imprime "Buzz" en lugar del número.

Para los números que son múltiplos de ambos, 3 y 5, imprime "FizzBuzz".

```
for (let i = 1; i <= 100; i++) {
    let output = '';

    if (i % 3 === 0) {
        output += 'Fizz';
    }
    if (i % 5 === 0) {
        output += 'Buzz';
    }

    console.log(output || i);
}
```

El bucle `for` recorre los números del 1 al 100.

La variable `output` se utiliza para construir la cadena que se imprimirá.

Si un número es divisible por 3 (es decir, `i % 3 === 0`), se añade "Fizz" a `output`.

Si un número es divisible por 5, se añade "Buzz" a `output`.

Si un número no es divisible ni por 3 ni por 5, `output` estará vacío (""), por lo que el operador lógico `||` garantiza que se imprima el propio número (`i`).

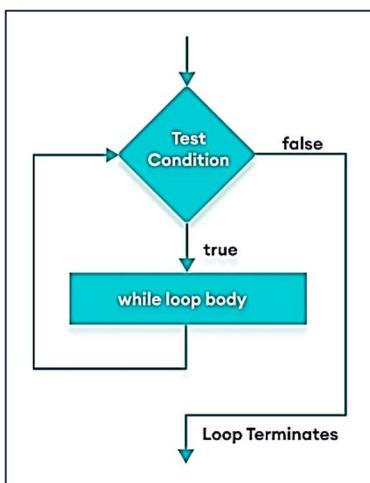
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
FizzBuzz
16
17
Fizz
Buzz



**Este ejercicio es interesante porque pone a prueba tu comprensión de los bucles, las condicionales y los operadores en JavaScript.**

## Ciclos Condicionales

### WHILE



El bucle while en JavaScript es una estructura de control que **permite ejecutar repetidamente un bloque de código mientras una condición dada sea verdadera**. La sintaxis básica del bucle while es bastante simple y directa.

El bucle while es una herramienta flexible en JavaScript, útil en una variedad de situaciones donde **el número de iteraciones no se conoce de antemano**.

```

while (condición) {
    // Código a ejecutar mientras la condición sea verdadera
}
  
```

**Condición:** Una expresión que se evalúa antes de cada iteración del bucle. Si la condición se evalúa como verdadera (true), el bloque de código dentro del bucle se ejecuta. Si se evalúa como falsa (false), el bucle termina.

```

let contador = 0;

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

Este código imprimirá los números del 0 al 4. En cada iteración, se verifica si contador es menor que 5. Si es así, se ejecuta el bloque de código, imprimiendo el valor actual de contador y luego incrementándolo en 1. Cuando contador llega a 5, la condición se vuelve falsa y el bucle termina.

## Consideraciones Importantes

**Condición de Terminación Clara:** Es crucial que el bucle while tenga una condición de terminación clara para evitar bucles infinitos. Asegúrate de que la condición eventualmente se evalúe como falsa.

**Actualización de la Condición:** Dentro del bucle, necesitas tener un mecanismo (como incrementar una variable contador) que eventualmente haga que la condición se evalúe como falsa.

**Uso vs. Bucle for:** A menudo, el bucle while es preferido sobre el bucle for cuando no sabes de antemano cuántas veces necesitas iterar.

### Otro ejemplo

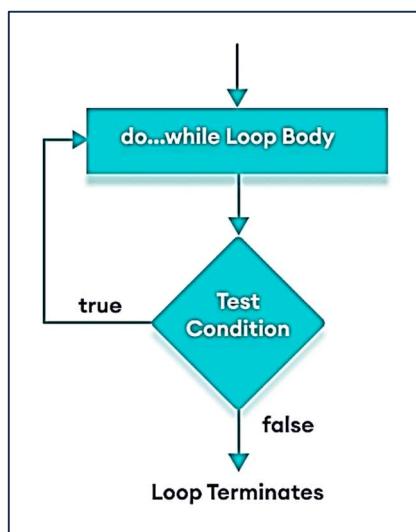
```
let x = 0;
let y = 10;

while (x < 5 && y > 5) {
    console.log(`x: ${x}, y: ${y}`);
    x++;
    y--;
}
```

En este ejemplo, el bucle while se ejecuta mientras ambas condiciones `x < 5` y `y > 5` sean verdaderas. La complejidad de la condición puede aumentar según las necesidades del programa.

## Ciclos Condicionales

### DO WHILE



La estructura do...while es un tipo de bucle que **ejecuta un bloque de código una vez y luego continúa ejecutándolo mientras una condición especificada sea verdadera**. La característica principal de este bucle es que **garantiza que el bloque de código se ejecute al menos una vez, independientemente de la condición**, ya que la condición se evalúa después de la primera ejecución del bloque.

La sintaxis del bucle do...while es la siguiente:

```
do {
    // Código a ejecutar
} while (condicion);
```

El bloque de código dentro de las llaves {} se ejecuta primero.

Después de ejecutar el bloque de código, se evalúa la condición.

Si la condición es verdadera (true), el bucle se repite; si es falsa (false), el bucle termina.

Ejemplo

```
let contador = 0;

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

En este ejemplo, el código dentro del bucle se ejecutará al menos una vez, imprimiendo el valor inicial de contador (0), y luego continuará ejecutándose mientras contador sea menor que 5.

El bucle do...while es útil cuando necesitas que el bloque de código se ejecute al menos una vez antes de evaluar la condición, como en algunos menús de usuario o en situaciones donde la condición depende de algo que ocurre dentro del bucle.

## DOM → DOCUMENT OBJECT MODEL

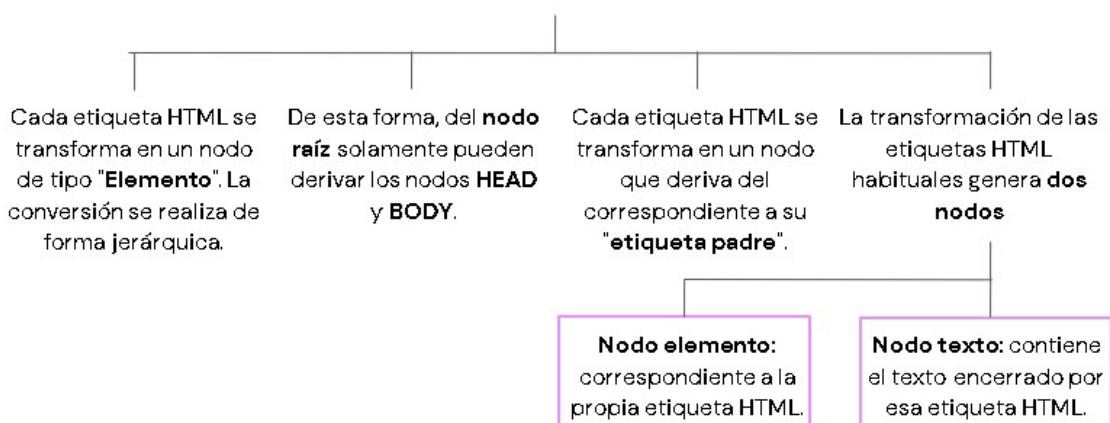
**El DOM (Document Object Model) es una interfaz de programación para documentos web.** Proporciona una representación estructurada del documento HTML o XML (**Se podría decir que el DOM es todo el código HTML**) y define cómo los programas pueden interactuar y modificar la estructura, estilo y contenido de estos documentos.

En el Modelo de Objetos del Documento (DOM), **cada etiqueta HTML, texto, etc. es un objeto**, al que podemos llamar **nodo**.

Las etiquetas anidadas son llamadas “nodos hijos” de la etiqueta “nodo padre” que las contiene.

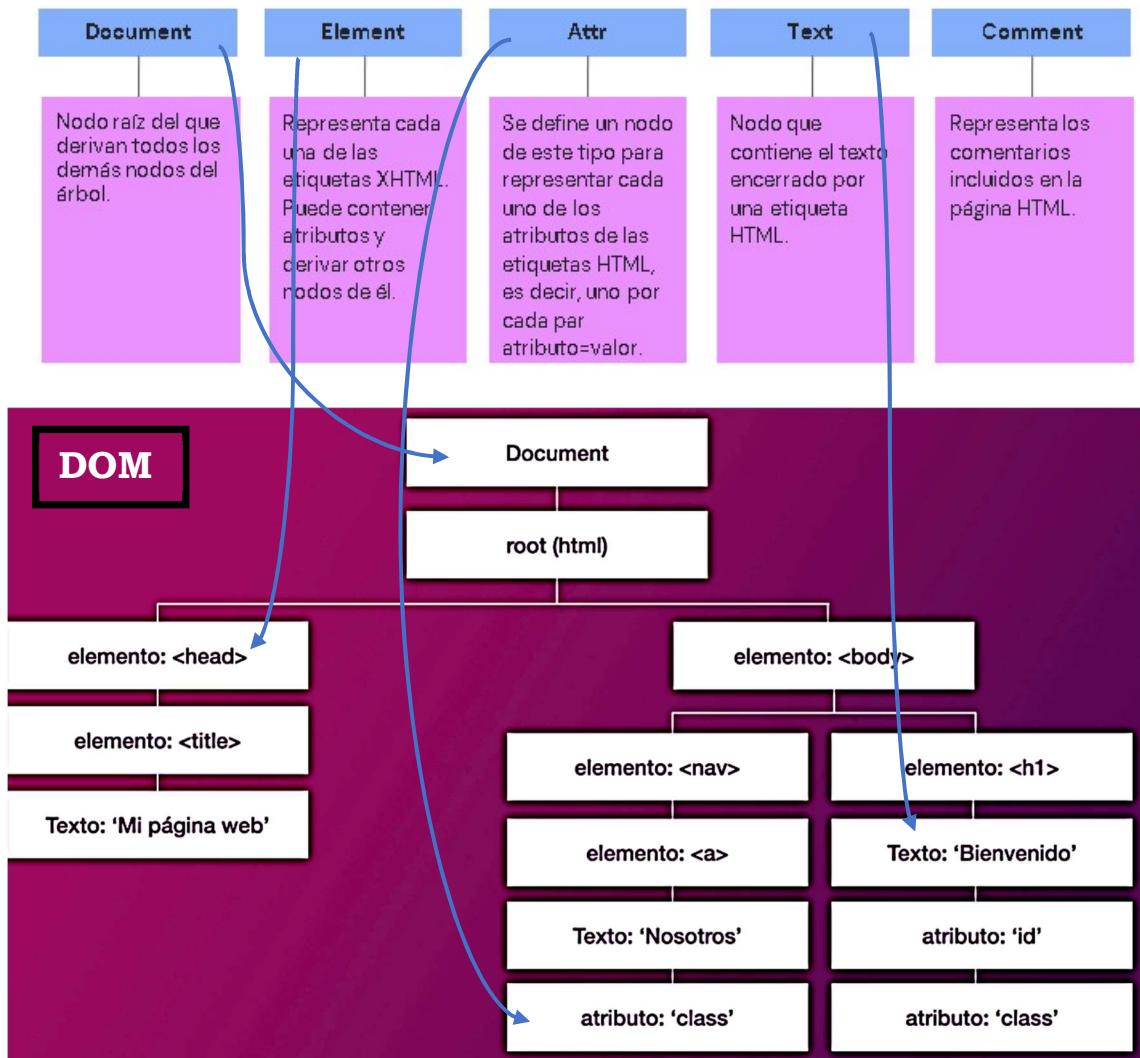
**Todos estos objetos son accesibles** empleando **JavaScript mediante el objeto global document**.

## Estructura DOM



# Tipos de Nodos

La especificación completa de DOM define 12 tipos de nodos, los más usados son:



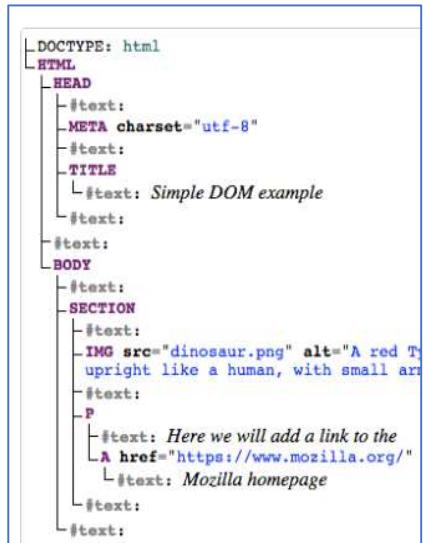
**En el contexto de JavaScript en el navegador, el DOM ofrece una forma de manipular páginas web.**

**Aquí hay algunos aspectos clave del DOM en JS**

## Representación del Documento

**El DOM representa un documento como un árbol de nodos.** Estos nodos pueden ser elementos HTML, texto, comentarios, y otros tipos de nodos.

En el contexto del DOM, **document es un objeto** que actúa como punto de entrada al contenido de la página web. **Representa el documento HTML cargado en el navegador y proporciona métodos y propiedades para acceder y manipular este documento.**



Cada entrada en el árbol se llama **nodo**. Algunos nodos representan elementos (identificados como, HTML, HEAD, BODY, IMG, P, etc.) y otros representan texto.

**También se hace referencia a los nodos por su posición en el árbol en relación con otros nodos**

- **Nodo raíz:** el nodo superior del árbol, que en el caso de HTML es siempre el nodo HTML.
- **Nodo hijo:** un nodo directamente dentro de otro nodo. Por ejemplo, IMG es hijo de SECTION.
- **Nodo descendiente:** un nodo en cualquier lugar dentro de otro nodo. Por ejemplo, IMG es hijo de SECTION y también es descendiente. IMG no es hijo de BODY, ya que está dos niveles por debajo de él en el árbol, pero es descendiente de BODY.
- **Nodo principal:** un nodo que tiene otro nodo en su interior. Por ejemplo, BODY es el nodo principal de SECTION.
- **Nodos hermanos:** nodos que se encuentran en el mismo nivel en el árbol DOM. Por ejemplo, IMG y P son hermanos.

## Navegación en el DOM

Puedes navegar por el árbol del DOM utilizando propiedades como parentNode, childNodes, firstChild, lastChild, nextSibling y previousSibling.

## Seleccionar Elementos:

Puedes seleccionar elementos específicos **utilizando métodos** como:

**document.getElementById()**

**document.getElementsByClassName()**

**document.getElementsByTagName()**

**document.querySelector()**

**document.querySelectorAll()** para selectores CSS más complejos.

**El objeto document es parte del modelo de objetos del navegador** y se refiere específicamente al documento HTML cargado en la ventana o pestaña actual del navegador. Es como una puerta de entrada a todo el contenido de la página web, y los métodos que proporciona, como `getElementById`, son herramientas para interactuar con este contenido.

```
let elemento = document.getElementById('miId');
let elementos = document.querySelectorAll('.miClase');
```

## Modificar Elementos:

Una vez que tienes una referencia a un elemento del DOM, puedes modificar su contenido, estilo y atributos.

```
elemento.textContent = 'Nuevo texto'; // Cambia el texto del elemento
elemento.style.color = 'blue'; // Cambia el color del texto a azul
elemento.setAttribute('href', 'http://ejemplo.com'); // Cambia el atributo
```

## Eventos

El DOM permite manejar eventos, como clics, movimientos del mouse, presiones de teclas, etc.

Puedes añadir manejadores de eventos a los elementos para ejecutar código cuando ocurran estos eventos.

```
elemento.addEventListener('click', function() {
    console.log('Elemento clickeado');
});
```

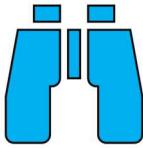
## Creación y Eliminación de Elementos

JavaScript puede crear nuevos nodos del DOM y añadirlos a la página, así como eliminarlos.

```
let nuevoElemento = document.createElement('div'); // Crea un nuevo div
document.body.appendChild(nuevoElemento); // Añade el nuevo div al body

document.body.removeChild(nuevoElemento); // Elimina el div del body
```

**Manipular el DOM con JavaScript es una parte fundamental del desarrollo web, permitiendo crear páginas dinámicas e interactivas.**



**Para manipular un elemento dentro del DOM, primero debe seleccionarlo y almacenar una referencia dentro de una variable.**

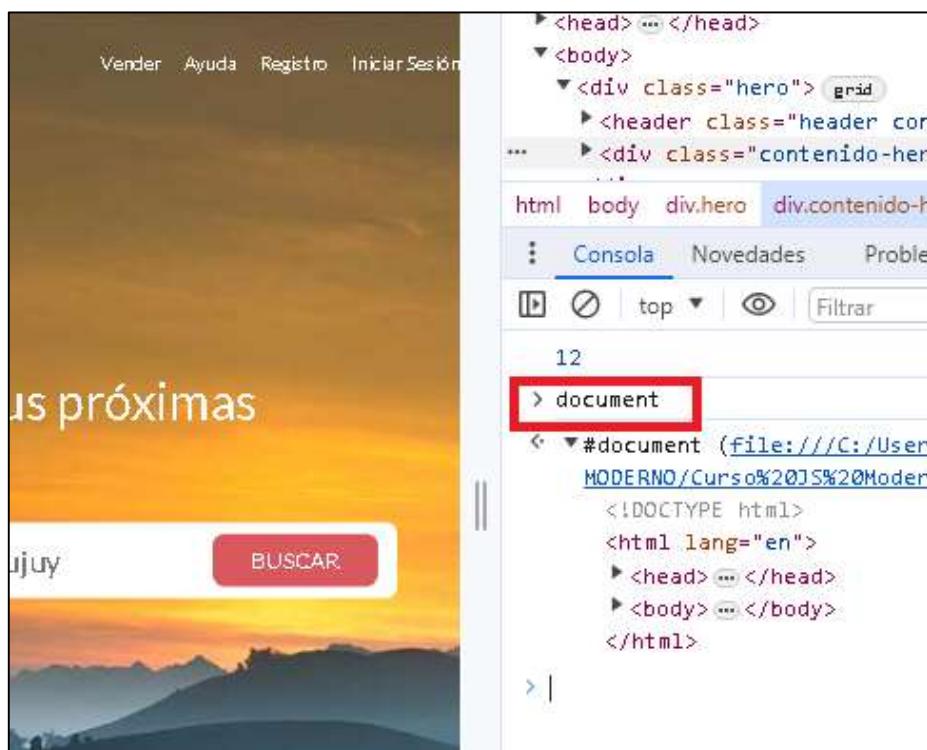
**Todos los ejemplos a continuación van a trabajarse en esta página**

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0, shrink-to-fit=no">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>MiViaje.com</title>
  <link rel="stylesheet" href="https://necolas.github.io/normalize.css/8.0.1/normalize.css">
  <link rel="stylesheet" href="https://fonts.googleapis.com/css?family=Open+Sans:400,700&display=swap">
  <link rel="stylesheet" href="css/fontawesome.css">
  <link rel="stylesheet" href="css/styles.css">
  <!-- Code injected by Five-server -->
  <script async data-id="five-server" data-file="index.js" data-type="js"></script>
</head>
<body>

  <div class="hero">
    <header class="header contenedor">
      <div class="logo">
        
      </div>
      <nav class="navegacion">
        <a href="#">Vender</a>
        <a href="#">Ayuda</a>
        <a href="#">Registro</a>
        <a href="#">Iniciar Sesión</a>
      </nav>
    </header>
    <div class="contenido-hero contenedor">
      <h1>Encuentra hospedaje para tus próximas vacaciones</h1>
      <form action="/buscador" method="POST" class="formulario formulario-buscar" id="formulario">
        <input type="text" name="busqueda" class="busqueda" placeholder="New York, Londres, San Sa...">
        <input type="submit" value="Buscar" id="btn-submit" style="background-color: #ff6347; color: white; border: none; padding: 5px; border-radius: 5px; font-weight: bold;">
      </form>
    </div>
  </div> <!--.hero-->

  <main class="contenido contenedor">
    <section class="hacer">
      <h2>Que Hacer</h2>
      <div class="contenedor-cards">
        <div class="card">
          
          <div class="info">
            <p class="categoria concierto">concierto</p>
            <p class="titulo">Música electrónica 2021</p>
            <p class="precio">$1,200 por persona</p>
          </div>
        </div> <!--.card-->
        <div class="card">
          
          <div class="info">
            <p class="categoria concierto">concierto</p>
            <p class="titulo">Rock en Los Ángeles</p>
            <p class="precio">$300 por persona</p>
          </div>
        </div> <!--.card-->
        <div class="card">
          
          <div class="info">
            <p class="categoria clase">Clase Cocina</p>
            <p class="titulo">Comida Española para Principiantes</p>
            <p class="precio">$400 por persona</p>
          </div>
        </div> <!--.card-->
      </div>
    </section>
  </main>

```



**Si abro la página web y voy a la consola al escribir document me muestra toda la estructura de la pagina web actual.**

## SELECCIONAR ELEMENTOS

Para seleccionar elementos siempre se hace **referencia primero al objeto document**

```
let elemento;
elemento = document
console.log(elemento)
```

```
:
Consola Novedades Problemas
Filtrar

▼ #document ⓘ
▶ location: Location {ancestorOrigin: "file:///C:/Users/Alberto/D", URL: "file:///C:/Users/Alberto/D", activeElement: body, adoptedStyleSheets: Proxy(Array), alinkColor: "", all: HTMLAllCollection(179) [ht..., anchors: HTMLCollection [], applets: HTMLCollection [], baseURI: "file:///C:/Users/Alberto/D", ...}
```

```
let elemento;
elemento = document.querySelectorAll("*");
console.log(elemento)
```

Para seleccionar todos los elementos del HTML

```
HTMLAllCollection(169) [html, head, meta, meta, meta, title, link, link, link, link, body, div.hero, header.header.contenedor, div.logo, img, nav.navegacion, a, a, a, a, a, a, div.contenedor-hero.contenedor, h1, span, form#formulario.formulario.formulario-buscar, input.busqueda, input#btn-submit, main.contenido.contenedor, section.hacer, h2, div.contenedor-cards, div.card, img, div.info, p.categoría.concierto, p.título, p.precio, div.card, img, div.info, p.categoría.concierto, p.título, p.precio, div.card, img, div.info, p.categoría.paseo, p.título, p.precio, section.hacer, h2.mi-viaje-plus, div.contenedor-cards.premium, div.info, h3, a.botón.btn-mi-viaje, section.hospedaje, h2, div.contenedor-cards, div.card, img, div.info, p.categoría.hospedaje, p.título, p.precio, div.card, img, div.info, p.categoría.hospedaje, p.título, p.precio, p.título, p.precio, div.card, img, div.info, p.categoría.hospedaje, p.título, p.precio, section.destinos, h2, div.contenedor-cards, div.card, img, div.info, p.título, div.card, img, div.info, p.título, div.card, img, div.info, p.título, ...]
```

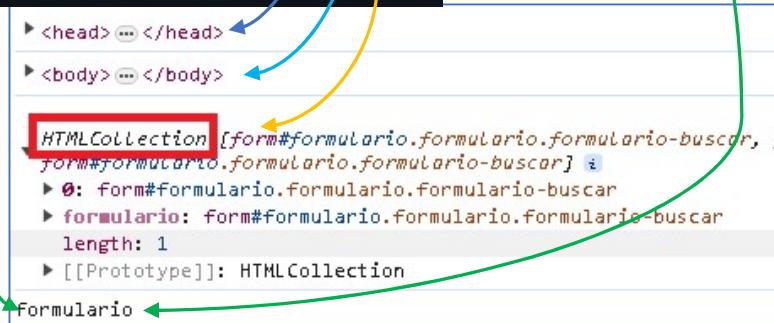
```
let elemento, elemento1, elemento2, elemento3;
elemento = document.head
console.log(elemento)

elemento1 = document.body
console.log(elemento1)

elemento2 = document.forms
console.log(elemento2)

elemento3 = document.forms[0].id
console.log(elemento3)
```

El elemento form **HTML COLLECTION** se estructura como un array en este caso de un solo elemento (por eso el índice es 0) por lo tanto se puede acceder a sus elementos por índices



De esta forma se pueden acceder a todos los elementos de un html (otros ejemplos).

```
let elemento, elemento1, elemento2, elemento3;
elemento = document.head
elemento1 = document.body
elemento2 = document.forms
elemento3 = document.forms[0].id
elemento3 = document.forms[0].classList
elemento3 = document.links[4].className
elemento3 = document.images
elemento3 = document.scripts
```

**Sin embargo, la mayoría de las veces la sintaxis anterior no se usa, lo más común es la selección como sigue a continuación:**

**document.getElementsByClassName()** → se utiliza para **seleccionar elementos HTML por su nombre de clase.**

```
let elemento;
elemento = document.getElementsByClassName("header")
console.log(elemento)
```

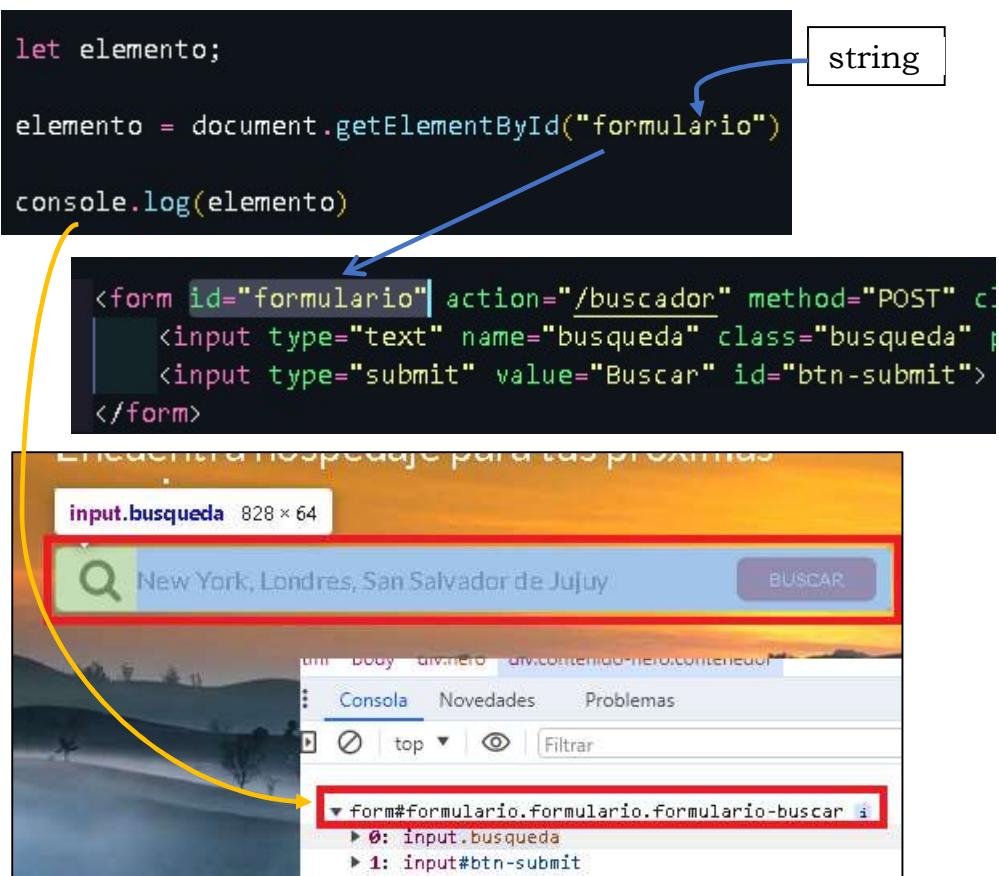
The diagram shows a code snippet using `getElementsByClassName` to select elements with the class "header". The resulting object is a `HTMLCollection` containing one element, which is then expanded to show its internal structure. This element is highlighted in the browser's DOM tree. The developer tools' Elements tab shows the selected collection in the console.

**Si las clases se repiten en el documento las seleccionará a todas**

```
elemento = document.getElementsByClassName("contenedor")
```

The diagram shows a code snippet using `getElementsByClassName` to select elements with the class "contenedor". The resulting collection contains four elements, all of which are highlighted in the browser's DOM tree. The developer tools' Elements tab shows the selected collection in the console.

**document.getElementById()** → Se utiliza para **seleccionar elementos HTML por su id.** Si hay dos id iguales solo selecciona la primera que encuentre.



**document.getElementsByTagName()** → → Se utiliza para **seleccionar elementos HTML por su nombre de etiqueta** (por ejemplo, "div", "p", "span").

**Retorna una colección HTML:** devuelve una colección HTML en vivo de todos los elementos en el árbol DOM que tienen el nombre de etiqueta especificado. La colección es "en vivo", lo que significa que se actualiza automáticamente cuando el documento cambia.



Tenga en cuenta que, como ocurre con muchas cosas en JavaScript, hay **muchas formas** de **seleccionar un elemento y almacenar una**

referencia a él en una variable. Los enfoques modernos recomendados son los siguientes:

`document.querySelector()` → seleccionar el primer elemento del DOM que coincide con un determinado selector CSS.

### Puntos clave

**Selección Única:** A diferencia de `document.querySelectorAll()`, `document.querySelector()` solo devuelve el primer elemento que coincide con el selector especificado. Si hay varios elementos que coinciden, solo obtendrás el primero.

**Uso de Selectores CSS:** Puedes utilizar cualquier selector válido de CSS dentro de `document.querySelector()`. Esto incluye **selectores de clase (.className)**, **selectores de ID (#idName)**, **selectores de atributo ([attribute=value])**, etc.

#### Ejemplo - Seleccionar por ID:

string

```
const elementById = document.querySelector("#miId");
```

#### Ejemplo - Seleccionar por Clase:

```
const elementByClass = document.querySelector(".miClase");
```

#### Ejemplo - Seleccionar por Tipo de Elemento:

```
const firstParagraph = document.querySelector("p");
```

#### Ejemplo - Seleccionar con un Selector Complejo:

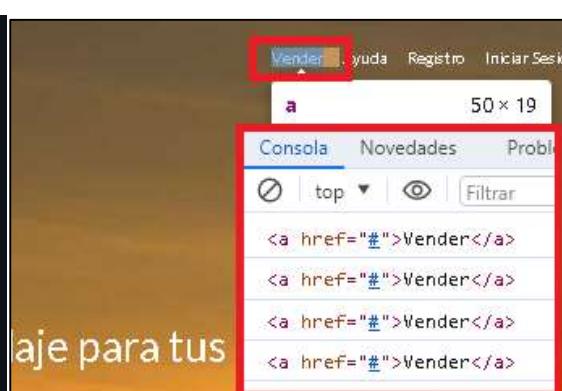
```
const firstInputActive = document.querySelector("input[type='text']:enabled");
```

Esto seleccionará el primer input de tipo texto que esté habilitado.

### Más ejemplos (notar que solo selecciona el primero que encuentra)

```
elemento = document.querySelector(".navegacion a")
elemento2 = document.querySelector("nav a")
elemento3 = document.querySelector("header nav a")
elemento4 = document.querySelector(".hero .contenedor .navegacion a")
```

```
<div class="hero">
  <header class="header contenedor">
    <div class="logo">
      
    </div>
    <nav class="navegacion">
      <a href="#">Vender</a>
      <a href="#">Ayuda</a>
      <a href="#">Registro</a>
      <a href="#">Iniciar Sesión</a>
    </nav>
  </header>
</div>
```



Si quisiera **seleccionar la segunda etiqueta a**, primero debo subir al padre (nav o .navegacion) y luego seleccionar el hijo específico:

```
elemento = document.querySelector(".navegacion a:nth-child(2)")
```

```
<div class="hero">
  <header class="header contenedor">
    <div class="logo">
      
    </div>
    <nav class="navegacion">
      <a href="#">Vender</a>
      <a href="#">Ayuda</a> ←
      <a href="#">Registro</a>
      <a href="#">Iniciar Sesión</a>
    </nav>
```

The developer tools console shows the output of `document.querySelectorAll('a')`:

- Vender
- Ayuda** 44 × 19
- Consola Novedades Problema
- ▶ <a href="#">Ayuda</a>
- [Five Server] connecting...
- [Five Server] connected.

**document.querySelectorAll()** → seleccionar todos los elementos del DOM que coinciden con un determinado selector CSS.

### Aspectos clave

**Selección Múltiple:** Retorna una NodeList que representa una **lista no viva** (no se actualiza automáticamente si el DOM cambia posteriormente) de todos los elementos del documento que coinciden con el conjunto de selectores CSS especificado.

**Uso de Selectores CSS:** Al igual que document.querySelector(), puedes usar cualquier selector válido de CSS (es decir se selecciona de la misma manera). Esto incluye combinaciones complejas de selectores.

#### Ejemplo - Seleccionar por Clase:

```
const elementosByClass = document.querySelectorAll(".miClase");
```

Esto seleccionará todos los elementos en el documento con la clase miClase.

#### Ejemplo - Seleccionar por Selector de Atributo:

```
const inputTexts = document.querySelectorAll("input[type='text']");
```

Esto seleccionará todos los elementos <input> de tipo texto.

#### Ejemplo - Seleccionar Varios Tipos de Elementos:

```
const titlesAndParagraphs = document.querySelectorAll("h1, h2, p");
```

Esto seleccionará todos los elementos <h1>, <h2> y <p> en el documento.

## Más ejemplos (notar que selecciona todo lo que encuentra)

```

let elemento;

elemento = document.querySelectorAll(".card")

console.log(elemento)

```

```

<div class="contenedor-cards">
  <div class="card"> ...
  </div> <!--.card-->
  <div class="card"> ...
  </div> <!--.card-->
  <div class="card"> ...
  </div> <!--.card-->
  <div class="card">
    
    <div class="info"> ...
  </div>
</div>

<div class="contenedor-cards">
  <div class="card"> ...
  </div> <!--.card-->
  <div class="card"> ...
  </div> <!--.card-->
  <div class="card">
    
    <div class="info"> ...
  </div>
</div>

```

```

NodeList(15) [div.card, div
, div.card, div.card, div.c
▶ 0: div.card
▶ 1: div.card
▶ 2: div.card
▶ 3: div.card
▶ 4: div.card
▶ 5: div.card
▶ 6: div.card
▶ 7: div.card
▶ 8: div.card
▶ 9: div.card
▶ 10: div.card
▶ 11: div.card
▶ 12: div.card
▶ 13: div.card
▶ 14: div.card
length: 15
[[Prototype]]: NodeList

```

**Iterar sobre la NodeList:** Puedes iterar sobre los elementos devueltos usando bucles como **forEach**:

```

document.querySelectorAll("p").forEach(function(p) {
  console.log(p.textContent);
});

```

Esto imprimirá el contenido de texto de todos los párrafos.

## MODIFICAR ELEMENTOS

Modificar elementos seleccionados del DOM en JavaScript se puede realizar de varias maneras, dependiendo de lo que necesites hacer con esos elementos. Aquí hay algunas técnicas comunes:

### Modificar el Contenido

Para **cambiar el contenido interno de un elemento**, puedes usar la propiedad **innerHTML** o **textContent**.

**innerHTML → Cambia el HTML interno del elemento.**

Se utiliza para obtener o establecer el contenido HTML de un elemento. Al usar esta propiedad, puedes cambiar **no solo el texto** del elemento sino también su **estructura HTML interna** (**no se puede cambiar la etiqueta principal, pero si las sub etiquetas dentro de esta**).

**Consideraciones de Seguridad:** Al utilizar innerHTML, es importante tener cuidado con la inyección de contenido malicioso (XSS - Cross-Site Scripting). Si estás insertando contenido basado en la entrada del usuario, es esencial validar y escapar esa entrada para evitar vulnerabilidades de seguridad.

```
<div class="contenido-hero contenedor">
  <h1>Encuentra <span> hospedaje </span>para tus próximas vacaciones</h1>
```



Seleccionando el elemento `<h1>`, puedes agregar nuevos elementos HTML dentro de este elemento. Por ejemplo, si quieres agregar negrita y subrayado dentro de tu `<h1>` (y también cambiar el texto), puedes hacerlo de la siguiente manera:

```
let titulo = document.querySelector("h1");
titulo.innerHTML = 'Encuentra <span>hospedaje</span> y <b><u>aventuras</u></b> en vacaciones';
```



**textContent → Cambia solo el texto**, ignorando cualquier etiqueta HTML. Proporciona una manera de obtener o establecer el contenido de texto de un nodo y de todos sus descendientes. A diferencia de innerHTML, textContent trata todo el contenido exclusivamente como texto, lo que significa que no interpreta ninguna etiqueta HTML.

**Consideraciones de Seguridad:**.textContent es más seguro en comparación con innerHTML cuando se trata de prevenir ataques XSS, ya que no interpreta el contenido como HTML.

```
<div class="contenido-hero contenedor">
  <h1>Encuentra <span> hospedaje </span>para tus próximas vacaciones</h1>
```



```
let titulo = document.querySelector("h1");
titulo.textContent = 'Encuentra hospedaje y aventuras en vacaciones';
```



**innerText → obtiene o establece el contenido de texto "visible" de un nodo y sus descendientes.** A diferencia de innerHTML y textContent, innerText está consciente del estilo y no incluye los elementos ocultos.

### Características de innerText

**Contenido Visible:** innerText solo devuelve el texto que es "visible" para el usuario en un navegador. Esto significa que ignora los elementos que están ocultos mediante CSS por ejemplo, **display: none o visibility: hidden**.

**Respeta el Espaciado Visual:** innerText respeta el espaciado y el salto de línea visual del contenido en la página web. Por ejemplo, si hay saltos de línea o espacios adicionales en la presentación visual del contenido, innerText los reflejará.

## Cambiar Estilos → Modificar CSS

Para **modificar los estilos CSS de un elemento**, puedes acceder a la **propiedad style** del elemento.

```
<div class="contenido-hero contenedor">
  <h1>Encuentra <span> hospedaje </span>para tus próximas vacaciones</h1>
```



Para ver y cambiar los estilos:

```
let titulo = document.querySelector("h1");
console.log(titulo)
```



```
▶ style: CSSStyleDeclaration {accentColor: '#', additiveSymbols: '#',
```



```
let titulo = document.querySelector("h1");
titulo.style.backgroundColor="lightgray";
titulo.style.textDecoration="underline";
titulo.style.fontStyle = "bold";
titulo.style.color= "darkblue";
titulo.style.fontSize= "3rem";
console.log(titulo)
```

Encuentra hospedaje para tus próximas vacaciones

BUSCAR

## Manipular Clases

Puedes agregar, eliminar o alternar clases CSS utilizando el **objeto classList**.

**classList.add()** → Agrega una o más clases a un elemento.

```
let botonesNav = document.querySelector(".navegacion");

botonesNav.classList.add("navegacion1")

console.log(botonesNav)
```

```
.navegacion a{
    color: var(--claro);
    text-decoration: none;
    margin-right: 1rem;
}
.navegacion a:last-of-type {
    margin-right: 0;
}
```

Vender Ayuda Registro Iniciar Sesión

► <nav class="navegacion navegacion1">...</nav>

Vender Ayuda Registro Iniciar Sesión

**classList.remove()** → Elimina una o más clases de un elemento.

```
let botonesNav = document.querySelector(".navegacion");

botonesNav.classList.remove("navegacion")
botonesNav.classList.add("navegacion1")
```

► <nav class="navegacion1">...</nav>

**classList.toggle()** → Alternar una clase en un elemento. Si la clase existe en el elemento, la elimina, y si no existe, la agrega.

```
botonesNav.classList.remove("navegacion")
botonesNav.classList.add("navegacion1")
botonesNav.classList.toggle("navegacion3")
```

```
▼ nav.navegacion1.navegacion3 ⓘ
  ▼ classList: DOMTokenList(2)
    0: "navegacion1"
    1: "navegacion3"
    length: 2
    value: "navegacion1 navegacion3"
  ► [[Prototype]]: DOMTokenList
```

Estos métodos hacen que trabajar con clases CSS en elementos del DOM sea mucho más manejable y menos propenso a errores que **manipular directamente la propiedad `className` del elemento**, la cual es simplemente una cadena de texto.

## Modificar Atributos

Puedes cambiar los atributos de un elemento con métodos como **`setAttribute`** y **`removeAttribute`**.

Para **modificar el atributo `src`** se procede de la siguiente manera:

```
<section class="hacer">
  <h2>Que Hacer</h2>
  <div class="contenedor-cards">
    <div class="card">
      
      <div class="info">
        <p class="categoria concierto">concierto</p>
        <p class="titulo">Música electrónica 2021</p>
        <p class="precio">$1,200 por persona</p>
      </div>
    </div> <!--.card-->
```



```
let imagen = document.querySelector(".card img");
imagen.src="img/hacer4.jpg"
console.log(titulo)
```



## ELIMINAR ELEMENTOS

Eliminar elementos del DOM en JavaScript puede realizarse de varias maneras, **pero el enfoque más común y directo es usar el método `removeChild()` o `remove()`**. Aquí te explico cómo puedes hacerlo:

### Usando `removeChild()`

Este método es un poco más antiguo, pero ampliamente soportado y funciona eliminando un nodo hijo específico de un nodo padre.

```
html
<div id="contenedor">
    <p id="parrafo">Este es un párrafo que será eliminado.</p>
</div>

// Primero, selecciona el elemento padre
var contenedor = document.getElementById("contenedor");

// Luego, selecciona el elemento que quieras eliminar
var parrafo = document.getElementById("parrafo");

// Finalmente, elimina el elemento hijo del padre
contenedor.removeChild(parrafo);
```

### Usando `remove()`

Este es un método más moderno y permite eliminar directamente un elemento sin necesidad de referenciar al elemento padre. Sin embargo, puede no ser compatible con navegadores más antiguos.

```
// Selecciona el elemento que quieras eliminar
var parrafo = document.getElementById("parrafo");

// Elimina el elemento
parrafo.remove();
```

## Consideraciones

**removeChild()** necesita que accedas tanto al elemento padre como al hijo, lo que puede ser un poco más tedioso si no tienes una referencia directa al padre.

**remove()** es más sencillo y directo, pero asegúrate de que es compatible con los navegadores que tu público objetivo utiliza.

Ambos métodos son efectivos para eliminar elementos del DOM. La elección entre uno u otro dependerá de tus necesidades específicas y del soporte del navegador que requieras.

# CREAR ELEMENTOS

Crear elementos en el Document Object Model (DOM) desde JavaScript es un proceso fundamental en la manipulación dinámica de páginas web. Se puede hacer creando nuevos nodos de elementos y luego insertándolos en el DOM existente.

## Paso 1: Crear un Nuevo Elemento

Usa el método **document.createElement()** para crear un nuevo nodo de elemento.

Ejemplo: Crear un nuevo párrafo (<p>)

```
let nuevoParrafo = document.createElement("p");
```

## Paso 2: Añadir Contenido al Elemento

Puedes añadir texto al elemento con textContent o innerText, o incluso HTML con innerHTML.

Ejemplo: Añadir texto al párrafo

```
nuevoParrafo.textContent = "Este es un nuevo párrafo.";  
// o  
nuevoParrafo.innerHTML = "Este es un nuevo <strong>párrafo</strong&gt.";
```

## Paso 3: Insertar el Elemento en el DOM

Elige dónde quieres colocar el nuevo elemento en tu página y usa métodos como appendChild() o insertBefore() para insertarlo en el DOM.

Ejemplo: Añadir el nuevo párrafo al final de un div existente

```

html
<div id="miDiv"></div>

javascript
let contenedor = document.getElementById("miDiv");
contenedor.appendChild(nuevoParrafo);

```

## Otras Consideraciones

**Añadir Atributos:** Puedes añadir atributos al nuevo elemento con **setAttribute()**.

```
nuevoParrafo.setAttribute("id", "parrafoNuevo");
```

**Insertar Antes de Otro Elemento:** Usa **insertBefore()** si necesitas insertar el elemento en un lugar específico que no sea el final.

```
contenedor.insertBefore(nuevoParrafo, contenedor.firstChild); // Inserta al inicio del contenedor
```

**Uso de insertAdjacentElement e insertAdjacentHTML:** Estos métodos proporcionan más control sobre dónde insertar el nuevo elemento en relación con un elemento existente.

## TRAVERSING THE DOM → RECORRIDO

Se refiere a la **habilidad de navegar por los nodos** del árbol del Document Object Model (DOM) para seleccionar, acceder y manipular elementos en una página web.

```

<div class="hero">
  <header class="header contenedor">
    <div class="logo">
      
    </div>
    <nav class="navegacion">
      <a href="#">Vender</a>
      <a href="#">Ayuda</a>
      <a href="#">Registro</a>
      <a href="#">Iniciar Sesión</a>
    </nav>
  </header>
  <div class="contenido-hero contenedor">
    <h1>Encuentra <span> hospedaje </span>para tus próximas vacaciones</h1>
    <form action="/buscador" method="POST" class="formulario formulario-buscar" id="formulario" >
      <input type="text" name="busqueda" class="busqueda" placeholder="New York, Londres, Roma,>
      <input type="submit" value="Buscar" id="btn-submit">
    </form>
  </div>
</div> <!--.hero-->

```

## Conceptos Básicos del Recorrido del DOM

### Nodos Padre, Hijo y Hermano:

**Padre (Parent):** Cada elemento en el DOM, excepto el elemento raíz (<html>), tiene un nodo padre. Puedes acceder al padre de un elemento con parentNode o parentElement.

```
let botonesNav1 = document.querySelector(".navegacion").parentNode;
let botonesNav2 = document.querySelector(".navegacion").parentElement;
```

- ▶ header.header.contenedor
- ▶ header.header.contenedor

Otros elementos contenidos dentro de ellos. Accedes a los hijos mediante childNodes, children, firstChild, lastChild, firstElementChild y lastElementChild.

```
let botonesNav3 = document.querySelector(".navegacion").childNodes;
```

- ▶ NodeList(9) [text, a, text, a, text, a, text, a, text]

<a href="#">Vender</a>
<a href="#">Ayuda</a>
<a href="#">Registro</a>
<a href="#">Iniciar Sesión</a>

Estos espacios en blanco (saltos de línea) se consideran un elemento text.

```
let botonesNav3 = document.querySelector(".navegacion").children;
```

- ▼ HTMLCollection(4) [a, a, a, a] ⓘ
- ▶ 0: a
- ▶ 1: a
- ▶ 2: a
- ▶ 3: a
- ▶ length: 4
- ▶ [[Prototype]]: HTMLCollection

```
let botonesNav3 = document.querySelector(".navegacion").lastChild;
```

- ▶ #text

```
let botonesNav3 = document.querySelector(".navegacion").lastElementChild;
```

- ▶ a

## Acceder a elementos por índice

```
let botonesNav3 = document.querySelector(".card").children[1].children[2];
```

```
<div class="card">
  
  <div class="info">
    <p class="categoria concierto">concierto</p>
    <p class="titulo">Música electrónica 2021</p>
    <p class="precio">$1,200 por persona</p>
  </div>
```

```
▼ HTMLCollection(3) [p.categoria.concierto, p.titulo, p.precio] ⓘ
  ▶ 0: p.categoria.concierto
  ▶ 1: p.titulo
  ▶ 2: p.precio
  length: 3
  ▶ [[Prototype]]: HTMLCollection
```

► p.precio

**Hermanos (Siblings):** Los elementos al mismo nivel en la estructura del DOM son hermanos. Usa `nextSibling`, `previousSibling`, `nextElementSibling` y `previousElementSibling` para navegar entre ellos.

```
<div class="info">
  <p class="categoria concierto">concierto</p>
  <p class="titulo">Música electrónica 2021</p>
  <p class="precio">$1,200 por persona</p>
</div>
```

```
let botonesNav1 = document.querySelector(".info p");
```

► p.categoria.concierto

```
botonesNav1 = document.querySelector(".info p").nextElementSibling;
```

► p.titulo

## Seleccionar Elementos:

`getElementById`, `getElementsByClassName`, `getElementsByTagName`, `querySelector` y `querySelectorAll`

## Traversing Horizontal (Hermanos):

`nextSibling` / `previousSibling`: Selecciona el siguiente o anterior nodo hermano, respectivamente, incluyendo nodos de texto y comentarios.

`nextElementSibling` / `previousElementSibling`: Selecciona el siguiente o anterior elemento hermano, respectivamente, excluyendo nodos de texto y comentarios.

## Traversing Vertical (Padres e Hijos):

**parentNode / parentElement:** Retorna el nodo o elemento padre.

**childNodes:** Retorna una NodeList que contiene todos los nodos hijos, incluyendo nodos de texto y comentarios.

**children:** Retorna una HTMLCollection solo con los elementos hijos.

**firstChild / lastChild:** Retorna el primer o último nodo hijo.

**firstElementChild / lastElementChild:** Retorna el primer o último elemento hijo.

## Consideraciones

**NodeList vs HTMLCollection:** **childNodes** devuelve una NodeList, que puede incluir nodos de texto y comentarios, mientras que **children** devuelve una HTMLCollection que solo contiene elementos.

**Live vs Static Collections:** **Algunas colecciones son "en vivo"** (como childNodes y children), lo que significa que reflejan los cambios en el DOM en tiempo real. Por otro lado, los resultados de querySelectorAll son estáticos y no cambian si el DOM se modifica después de su ejecución.

**Rendimiento:** Ten cuidado al usar estas propiedades y métodos en bucles o en operaciones repetidas, ya que el acceso excesivo al DOM puede afectar el rendimiento.

# EVENTOS

Los eventos son fundamentales para interactuar con el usuario y el navegador. **Son acciones o sucesos que ocurren en el navegador cuando sucede algo como clics, pulsaciones de teclas, movimientos del ratón, cargas de páginas, etc.,** a los que puedes responder con código JavaScript.

JavaScript permite **asignar una función a cada uno de los eventos.** Reciben el nombre de **event handlers** o **manejadores de eventos.**

Así, ante cada evento, JavaScript asigna y ejecuta la **función asociada** al mismo.

Hay que entender que los eventos **suceden** constantemente en el navegador.

JavaScript lo que nos permite hacer es **escuchar** eventos sobre elementos seleccionados.

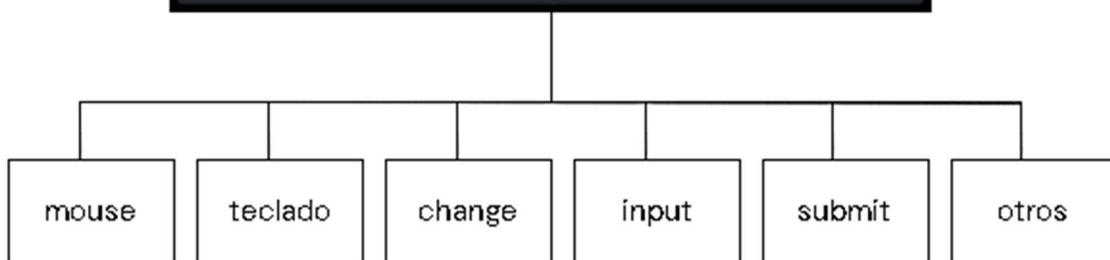
Cuando escuchamos el evento que esperamos, se ejecuta la función que definimos en **respuesta**.

A esta escucha se la denomina **event listener**.

### Tipos Comunes de Eventos

- 1. Eventos del Mouse:** click, dblclick, mouseover, mouseout, mousemove, etc.
- 2. Eventos del Teclado:** keypress, keydown, keyup.
- 3. Eventos de Formulario:** submit, change, focus, blur.
- 4. Eventos de la Ventana:** load, resize, scroll, unload.

## Eventos más comunes



### Cómo Trabajar con Eventos

```
<button id="btnPrincipal">CLICK</button>
```

#### Opción 1

El método **addEventListener()** permite definir (registrar) qué evento escuchar sobre cualquier elemento seleccionado.

El primer parámetro corresponde al **nombre del evento** y el segundo a la **función de respuesta**.

```

<script>
  let boton = document.getElementById("btnPrincipal")
  boton.addEventListener("click", respuestaClick)
  function respuestaClick(){
    console.log("Respuesta evento")
  }
</script>
  
```

## Opción 2

Emplear una **propiedad del nodo para definir la respuesta al evento**. Las propiedades se identifican con el **nombre del evento** y el **prefijo on**.

```
<script>
  let boton = document.getElementById("btnPrincipal")
  boton.onclick = () =>(console.log("Respuesta 2"))
</script>
```

También es posible emplear funciones anónimas para definir los manejadores de eventos.



**Las opciones 1 y 2 son las recomendadas.**

## Opción 3: Sintaxis

Determinar el evento **especificando el manejador de evento en el atributo de una etiqueta HTML**. La denominación del atributo es idéntica al de la propiedad de la opción 2 (prefijo on).

```
<input type="button" value="CLICK2" onclick="alert('Respuesta 3');" />
```

La función puede declararse entre las comillas o bien tomarse una referencia existente en el script.

## Eventos con el mouse

Se producen por la **interacción del usuario con el mouse**.

Entre ellos se destacarán los que se encuentran a continuación:

- ✓ **Mousedown(similar a dar un clic)/mouseup:** Se oprime/suelta el botón del ratón sobre un elemento.
- ✓ **mouseover(mouseenter)/mouseout:** El puntero del mouse se mueve sobre/sale del elemento.
- ✓ **mousemove:** El movimiento del mouse sobre el elemento activa el evento.
- ✓ **click:** Se activa después de mousedown o mouseup sobre un elemento válido.
- ✓ **dbleclick:** doble click.

```
//CODIGO HTML DE REFERENCIA
<button id="btnMain">CLICK</button>
//CODIGO JS
let boton = document.getElementById("btnMain")
boton.onclick = () => {console.log("Click")}
boton.onmousemove = () => {console.log("Move")}
```

## Eventos del teclado

Se producen por la interacción del usuario con el teclado.

Entre ellos se destacarán los que se encuentran a continuación.

- ✓ **keydown:** Cuando se presiona una tecla.
- ✓ **keyup:** Cuando se suelta una tecla.
- ✓ **blur:** Se dispara cuando un elemento pierde el foco. Este evento se suele utilizar en formularios para validar un campo cuando el usuario se mueve a otro campo o cierra el formulario.
- ✓ **copy:** Se dispara cuando se copia un elemento (con CTRL + C en Windows por ejemplo).
- ✓ **copy:** Se dispara cuando se pega un elemento (con CTRL + V en Windows por ejemplo).
- ✓ **cut:** Se dispara cuando se corta un elemento (con CTRL + X en Windows por ejemplo).

```
//CODIGO HTML DE REFERENCIA
<input id = "nombre" type="text">
<input id = "edad"    type="number">
//CODIGO JS
let input1 = document.getElementById("nombre")
let input2 = document.getElementById("edad")
input1.onkeyup = () => {console.log("keyUp")}
input2.onkeydown = () =>
{console.log("keyDown")}
```

## Evento change

El evento **change** se activa cuando se detecta un cambio en el valor del elemento.

Por ejemplo, mientras se escribe en un input de tipo texto **no hay evento change**, pero cuando se pasa a otra sección de la aplicación entonces sí ocurre.

```
//CODIGO HTML DE REFERENCIA
<input id = "nombre" type="text">
<input id = "edad"    type="number">
//CODIGO JS
let input1 =
document.getElementById("nombre");
let input2 = document.getElementById("edad");
input1.onchange = () =>
{console.log("valor1");}
input2.onchange = () =>
{console.log("valor2");}
```

## Evento input

Si queremos **ejecutar una función cada vez que se tipea sobre el campo**, conviene trabajar directamente con el evento **input**.

**Se ejecuta también cuando se pega un texto, o se corta, o se presiona una tecla o se suelta etc. etc (salvo el evento blur)**

```
//CODIGO HTML DE REFERENCIA
<input id = "nombre" type="text">

//CODIGO JS
let input1 = document.getElementById("nombre")
input1.addEventListener('input', () => {
    console.log(input1.value)
})
```

## Evento submit

```
//CODIGO HTML DE REFERENCIA
<form id="formulario">
    <input type="text">
    <input type="number">
    <input type="submit" value="Enviar">
</form>
//CODIGO JS
let miFormulario =
document.getElementById("formulario");
miFormulario.addEventListener("submit", validarFormulario);

function validarFormulario(e){
    e.preventDefault();
    console.log("Formulario Enviado");
}
```

El evento **submit** se activa cdo. el formulario es enviado. Normalmente se utiliza para validar el formulario antes de ser enviado al servidor o bien para abortar el envío y procesarlo con JavaScript.

## Objeto event

En algunos casos, necesitamos obtener **información contextual** del evento para poder realizar acciones.

Por ejemplo, ante el evento submit necesitamos prevenir el comportamiento por defecto para operar correctamente.

Para esto existe en JavaScript el **objeto event**.

**Es un objeto que se pasa automáticamente como argumento a los manejadores de eventos cuando ocurren.** Este objeto contiene información detallada y funciones relacionadas con el evento que se ha producido.

En todos los navegadores modernos se crea de forma automática un parámetro que se pasa a la función manejadora, por lo que no es necesario incluirlo en la **llamada**

Ese parámetro puede o no usarse en el handler, **pero siempre estará disponible en la llamada.**

### Propiedades Comunes del Objeto evento

**type:** Indica el tipo de evento que ha ocurrido (por ejemplo, "click", "mouseover").

**target:** Se refiere al elemento que desencadenó el evento.

**currentTarget:** Se refiere al elemento al que el manejador de eventos ha sido adjuntado.

**preventDefault():** Método que previene la acción por defecto asociada al evento (si es cancelable) por ejemplo en un formulario la acción por default es submit (enviar).

**stopPropagation():** Detiene la propagación del evento en las fases de captura y burbujeo.

**stopImmediatePropagation():** Detiene la propagación del evento y previene la ejecución de otros manejadores de eventos en el mismo elemento.

**bubbles:** Indica si un evento se propaga en la fase de burbujeo.

**keyCode / key (para eventos de teclado):** Indica la tecla que fue presionada.

## Ejemplo aplicado: Datos del formulario usando event

```
//CÓDIGO HTML DE REFERENCIA
<form id="formulario">
  <input type="text">
  <input type="number">
  <input type="submit" value="Enviar">
</form>
//CÓDIGO JS
let miFormulario = document.getElementById("formulario");
miFormulario.addEventListener("submit", validarFormulario);

function validarFormulario(e){
  //Cancelamos el comportamiento del evento
  e.preventDefault();
  //Obtenemos el elemento desde el cual se disparó el evento
  let formulario = e.target
  //Obtengo el valor del primero hijo <input type="text">
  console.log(formulario.children[0].value);
  //Obtengo el valor del segundo hijo <input type="number">
  console.log(formulario.children[1].value);
}
```

## Eventos de scroll

Los eventos de scroll en JavaScript **se refieren a las acciones que ocurren cuando un usuario desplaza una página o un elemento en particular**. Estos eventos se pueden manejar para realizar diversas funciones, como cargar contenido dinámicamente, animaciones basadas en el desplazamiento, seguimiento de la posición del scroll, entre otros.

**El evento principal asociado con el desplazamiento es scroll. Este evento se dispara en el objeto window o en cualquier elemento desplazable cuando se produce un desplazamiento.**

### Ejemplo Básico de Uso del Evento scroll

```
// Escuchar el evento de scroll en toda la ventana
window.addEventListener('scroll', function() {
  console.log("Se está desplazando la página");
});
```

### Ejemplo con un Elemento Específico

Si deseas escuchar el evento de scroll en un elemento específico, primero asegúrate de que el elemento pueda desplazarse (por ejemplo, con un alto fijo y overflow).

html

```
<div id="miDivScrollable" style="height: 100px; overflow-y: scroll;">
    <!-- Contenido largo aquí -->
</div>
```

```
document.getElementById('miDivScrollable').addEventListener('scroll', function() {
    console.log("Se está desplazando dentro del div");
})
```

## Event Propagation: Bubbling y Capturing

**Los eventos en JavaScript se propagan en dos fases:** bubbling (burbujeo) y capturing (captura).

Puedes **especificar el modo de propagación** en el método **addEventListener** usando un **tercer argumento booleano** (true para capturing, false para bubbling, siendo false el valor por defecto).

### Fase de Captura (Capturing)

**¿Qué es?** En la fase de captura, el evento comienza desde el objeto window y se propaga hacia abajo en el árbol del DOM hasta llegar al elemento que desencadenó el evento.

**Cómo Funciona:** Si haces clic en un elemento, el evento de clic primero pasa por el window, luego por los ancestros del elemento en el árbol del DOM, hasta que finalmente llega al elemento de destino.

### Fase de Burbujeo (Bubbling)

**¿Qué es?** Después de alcanzar el elemento objetivo, el evento comienza a "burbujear" hacia arriba a través de los ancestros del elemento en el árbol del DOM, hasta que llega al objeto window.

**Cómo Funciona:** En el mismo escenario de clic, después de llegar al elemento de destino, el evento de clic se propaga de vuelta hacia arriba, pasando por los ancestros del elemento, hasta que posiblemente alcance el objeto window.

**Uso en JavaScript:** La mayoría de los eventos en JavaScript se propagan en la fase de burbujeo. No necesitas hacer nada especial para capturar eventos en esta fase; es el comportamiento por defecto.

### Detener la Propagación

**event.stopPropagation():** Este método puede ser llamado en un manejador de eventos para detener la propagación del evento más allá del elemento actual, ya sea en la fase de captura o de burbujeo.



## ID y DATA-ID o DATA-\* atributos

### id

**Uso Principal:** El **atributo id proporciona un identificador único para un elemento HTML**. Este identificador debe ser único dentro de toda la página

**Acceso en JavaScript y CSS:** Se usa comúnmente para referenciar elementos en JavaScript (por ejemplo, con document.getElementById) y para definir estilos específicos en CSS.

```
html
<div id="miDivUnico">Contenido del Div</div>
```

**Restricciones:** Dado que debe ser único, **solo se debe usar un id por elemento y no se debe repetir en la página.**

### data-id

**Uso Principal:** El atributo data-id es un ejemplo de un "atributo de datos" (data-\*). Estos atributos permiten almacenar información adicional en un elemento HTML, que puede ser utilizada por JavaScript o CSS. A diferencia de id, los atributos data-\* no tienen restricciones en cuanto a unicidad; puedes tener varios elementos con el mismo data-id o diferentes data-\* atributos.

**Acceso en JavaScript:** Se accede a través de element.getAttribute('data-id') o element.dataset.id.

```
html
<div data-id="123">Contenido del Div</div>
```

**Flexibilidad:** Los data-\* atributos son extremadamente flexibles y pueden ser utilizados para almacenar una amplia variedad de datos adicionales sobre un elemento, lo que es particularmente útil para la manipulación con JavaScript.

**En resumen,** mientras que **id se utiliza para identificación única** y es fundamental para la estructura del documento y la aplicación de estilos, **data-id (y otros data-\* atributos) se utilizan para almacenar información adicional que puede ser utilizada por scripts y estilos**, sin la restricción de ser únicos en la página.



## VENTANAS EMERGENTES

Las ventanas emergentes en JavaScript se pueden crear utilizando varias técnicas y funciones. **Las más comunes son alert(), confirm(), y prompt()**. Estas funciones crean ventanas emergentes sencillas para mostrar mensajes, pedir confirmación o solicitar información al usuario.

**Alert:** Muestra un mensaje y un botón de "Aceptar". Se utiliza principalmente para informar al usuario.

```
alert("Este es un mensaje de alerta");
```

**Confirm:** Muestra un mensaje con botones de "Aceptar" y "Cancelar". Se utiliza para pedir confirmación al usuario. Devuelve **true** si el usuario hace clic en "Aceptar" y **false** si elige "Cancelar".

```
if (confirm("¿Estás seguro de querer continuar?")) {
    // El usuario hizo clic en "Aceptar"
} else {
    // El usuario hizo clic en "Cancelar"
}
```

**Prompt:** Muestra un cuadro de diálogo con un campo de texto para que el usuario ingrese información. Tiene botones de "Aceptar" y "Cancelar". Devuelve el texto ingresado si el usuario hace clic en "Aceptar" y null si elige "Cancelar".

```
var respuesta = prompt("¿Cuál es tu nombre?", "Escribe tu nombre aquí");
if (respuesta != null) {
    // El usuario ingresó un texto y presionó "Aceptar"
} else {
    // El usuario presionó "Cancelar"
}
```

Estas funciones son muy útiles para interacciones simples con el usuario. Sin embargo, tienen algunas limitaciones:

- No puedes personalizar su apariencia; el estilo y la presentación dependen del navegador y del sistema operativo del usuario.
- La ejecución del script se detiene hasta que el usuario responde a la ventana emergente.

Para ventanas emergentes más avanzadas y personalizables, puedes utilizar librerías de JavaScript como SweetAlert o Bootstrap Modal. Estas librerías ofrecen una mayor flexibilidad en términos de diseño y

comportamiento y pueden integrarse mejor en la experiencia de usuario de tu sitio web.

## STORAGE O ALMACENAMIENTO

El objeto **Storage (API de almacenamiento web)** permite **almacenar datos de manera local en el navegador sin necesidad de realizar ninguna conexión con el servidor.**

De esta manera, cada cliente puede **preservar información** de la aplicación.

**El navegador** nos ofrece **dos tipos de storage: localStorage y sessionStorage.**

Los datos almacenados en **localStorage** (variable global preexistente) **se almacenan en el navegador de forma indefinida** (o hasta que se borren los datos de navegación del browser):

La información persiste reinicio de navegador y hasta del sistema operativo.

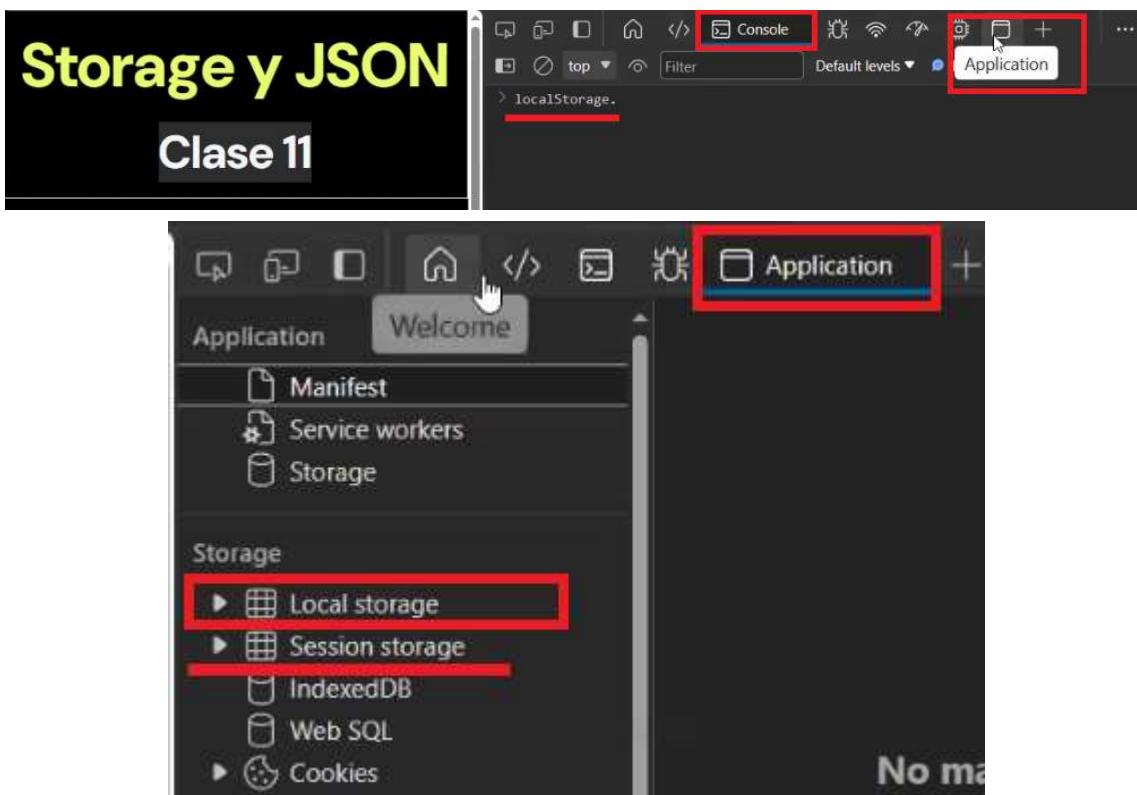
**La información almacenada en sessionStorage** (variable global preexistente) **se almacena en el navegador hasta que el usuario cierra la ventana.**

Solo existe dentro de la pestaña actual del navegador. Otra pestaña con la misma página tendrá otro sessionStorage distinto, pero se comparte entre iframes en la pestaña (asumiendo que tengan el mismo origen).

**Los datos almacenados en LocalStorage son específicos del navegador y del dominio en la máquina local donde se establecen.** Esto significa que los datos almacenados en LocalStorage no son compartidos entre diferentes navegadores, dispositivos o incluso perfiles de usuario en el mismo navegador. Aquí te explico las razones principales de esto:

**Almacenamiento Local del Navegador:** LocalStorage almacena datos en el disco duro del usuario en la computadora donde se ejecuta el navegador. Estos datos no se sincronizan ni se transfieren a otros dispositivos o navegadores.

**Independencia del Navegador y del Dispositivo:** Cada navegador en cada dispositivo tiene su propia implementación de LocalStorage. Incluso si usas el mismo navegador (como Chrome) en dos computadoras diferentes, cada una tendrá su propio almacenamiento independiente.



En JavaScript, **localStorage** es un objeto que proporciona acceso al almacenamiento local del navegador. Los objetos en JavaScript son colecciones de propiedades, y **localStorage** se comporta de manera similar al permitirte establecer, recuperar y eliminar propiedades que corresponden a pares clave-valor.

Sin embargo, **localStorage** es un tipo especial de objeto que está integrado en los navegadores web y tiene una API específica que permite el almacenamiento de datos de forma persistente. Por lo tanto, aunque técnicamente es un objeto debido a su estructura y funcionalidad en el lenguaje JavaScript, también es una parte de la Web Storage API que proporciona funcionalidades adicionales para el almacenamiento de datos en el lado del cliente.

En **localStorage** se guardan datos en forma de pares clave-valor, y aunque se manipulan como si fueran un objeto en JavaScript, hay algunas diferencias importantes:

1. **Clave-Valor:** Cada dato almacenado en **localStorage** es un par de clave-valor, donde **ambas son cadenas de texto (strings)**.
2. **Tipo de Datos:** Aunque puedes acceder a **localStorage** usando una sintaxis similar a la de los objetos en JavaScript (por ejemplo,

`localStorage.miClave)`, `localStorage` solo puede almacenar cadenas de texto. Esto significa que, si intentas almacenar otro tipo de datos, como un objeto o un array, estos serán convertidos a una cadena de texto (normalmente usando `JSON.stringify` para almacenarlos y `JSON.parse` para recuperarlos en su forma original).

3. **Persistencia:** A diferencia de los objetos de JavaScript normales, los datos almacenados en `localStorage` son persistentes. Es decir, sobreviven entre diferentes sesiones del navegador, incluso después de cerrar y abrir el navegador.
4. **Limitaciones de Tamaño:** `localStorage` tiene un límite de cuántos datos puedes almacenar, que generalmente es de aproximadamente 5MB por dominio.
5. **Acceso y Seguridad:** Los datos almacenados en `localStorage` son específicos del dominio, lo que significa que solo los scripts que se ejecutan en el mismo dominio que los guardó pueden acceder a ellos. Esto ayuda a prevenir ciertos tipos de ataques de scripts cruzados (XSS).

Entonces, mientras que la manipulación de `localStorage` puede parecerse a la de un objeto JavaScript, es importante recordar estas particularidades para su correcto uso.

## LocalStorage: Setitem

Para **almacenar información** se utiliza `setItem`:

Puedes almacenar datos en el `localStorage` utilizando la **propiedad `setItem(key, value)`** donde **key es una cadena** que actúa como un identificador único para el dato y **value es el valor que deseas almacenar**.

```
// Método -> localStorage.setItem(clave, valor)
// clave = nombre para identificar el elemento
// valor = valor/contenido del elemento

localStorage.setItem('bienvenida', '¡Hola Coder!');
localStorage.setItem('esValido', true);
localStorage.setItem('unNumero', 20);
```



**La información almacenada en el Storage se guarda en la forma de clave-valor.**

Similar al tratamiento de objetos, definimos **claves** (tienen que ser un string) en el storage donde almacenamos **valores**.



**Las claves y valores de Storage se guardan en formato de cadena de caracteres (DOMString).**

The screenshot shows the Chrome DevTools interface with the Application tab selected. In the left sidebar, under 'Almacenamiento', the 'LocalStorage' item is expanded, showing 'chrome://newtab' as the origin. A table lists a single item: 'Clave' (Key) 'usuario' (with value 'Alberto') and 'Valor' (Value) 'Alberto'. Arrows from the text above point to the 'localStorage' object in the console and the 'localStorage' entry in the DevTools table.

```
> localStorage
< Storage {usuario: 'Alberto', length: 1}
```

Clave	Valor
usuario	Alberto

Los datos en el **localStorage** se almacenan como **cadenas de texto**. Si necesitas almacenar objetos JavaScript, primero debes convertirlos a **cadenas JSON** utilizando **JSON.stringify** y luego almacenarlos.

## LocalStorage: getItem

Podemos **acceder a la información almacenada en localStorage** utilizando **getItem**. Las claves y valores de Storage se guardan en formato de **cadena de caracteres (DOMString)**.

Para **recuperar datos del localStorage**, puedes utilizar la **propiedad getItem(key)** y proporcionar la clave (key) del dato que deseas recuperar

```

let mensaje = localStorage.getItem('bienvenida');
let bandera = localStorage.getItem('esValido');
let numero = localStorage.getItem('unNumero');

console.log(mensaje); // '¡Hola Coder!'
console.log(bandera); // 'true'
console.log(numero); // '20'

```

Clave  
Valor

A través de la clave recupero el valor

**Si almacenaste un objeto como una cadena JSON, debes convertirlo** de nuevo a un objeto JavaScript utilizando **JSON.parse**.

**La conversión de una cadena JSON a un objeto JavaScript** utilizando **JSON.parse** **debe hacerse después de recuperar la cadena del almacenamiento local**, justo antes de usar el objeto.

```

> localStorage.setItem("usuario",Alberto)
> localStorage
< Storage {usuario: 'Alberto', length: 1}

> localStorage.getItem("usuario")
< 'Alberto'

```

## sessionStorage

La información almacenada en **sessionStorage** (variable global preexistente) **se almacena en el navegador hasta que el usuario cierra la ventana**.

**Solo existe dentro de la pestaña actual del navegador. Otra pestaña con la misma página tendrá otro sessionStorage distinto**, pero se comparte entre iframes en la pestaña (asumiendo que tengan el mismo origen).

El tratamiento es similar al **localStorage**

```

// Método -> sessionStorage.setItem(clave, valor)
// clave = nombre del elemento
// valor = Contenido del elemento
sessionStorage.setItem('seleccionados', [1,2,3]);
sessionStorage.setItem('esValido', false);
sessionStorage.setItem('email', 'info@email.com');

```

Podemos acceder a la información almacenada en sessionStorage utilizando **getItem**. Las claves y valores de Storage se guardan siempre en formato de **cadena de caracteres**

```
let lista = sessionStorage.getItem('seleccionados').split(",");
let bandera = (sessionStorage.getItem('esValido') == 'true');
let email = sessionStorage.getItem('email');

console.log(typeof lista); //object ["1","2","3"];
console.log(typeof bandera); //boolean;
console.log(typeof email); //string;
```

## Recorriendo el storage

Es posible obtener todos los valores almacenados en localStorage o sessionStorage con un **bucle**.

Pero no podemos usar **for...of** porque no son objetos iterables, ni **for...in** porque obtenemos otras propiedades del objeto que no son valores almacenados.

### El bucle a emplear es for con el método key:

```
//Ciclo para recorrer las claves almacenadas en el objeto
localStorage
for (let i = 0; i < localStorage.length; i++) {
    let clave = localStorage.key(i);
    console.log("Clave: "+ clave);
    console.log("Valor: "+ localStorage.getItem(clave));
}
```

## Eliminar datos del storage

Podemos eliminar la información almacenada en sessionStorage o localStorage usando el **método removeItem o clear**:

Para eliminar datos del **localStorage**, puedes utilizar el método **removeItem(key)** proporcionando la clave del dato que deseas eliminar.

```
localStorage.setItem('bienvenida', '¡Hola Code!');
sessionStorage.setItem('esValido', true);

localStorage.removeItem('bienvenida');
sessionStorage.removeItem('esValido');
localStorage.clear() //elimina toda la información
sessionStorage.clear() //elimina toda la información
```

Si deseas borrar todos los datos almacenados en el `localStorage`, puedes utilizar el método `clear()`:

```
localStorage.clear();
```

## JSON

**Si queremos almacenar la información de un objeto en un storage**, hay que tener en cuenta que tanto la clave como el valor se almacenan en **strings**.

**Ante cualquier otro tipo a guardar, como un número o un objeto**, se convierte a **cadena de texto automáticamente**

Entonces, al buscar almacenar un objeto sin una transformación previa, guardamos **[object Object]**, la **conversión por defecto de objeto a string**. Para guardar la información correctamente hay que transformar el objeto a **JSON**.

### ¿Qué es JSON?

**JavaScript Object Notation (JSON)** es un **formato basado en texto plano**, para **representar datos estructurados con la sintaxis de objetos de JavaScript**. Es comúnmente utilizado para enviar y almacenar datos en aplicaciones web.

Aunque **es muy parecido (casi similar) a la sintaxis de JavaScript**, puede ser utilizado independientemente de JavaScript, y muchos entornos de programación poseen la capacidad de **leer (convertir→ parsear) y generar JSON**.

**JSON es un string con un formato específico.**

**JSON**, es un formato de intercambio de datos ligero y fácil de leer para humanos, y máquinas.

**Es un formato de texto completamente independiente del lenguaje**, pero utiliza convenciones que son familiares para los programadores de la familia de lenguajes C, incluyendo C, C++, C#, Java, JavaScript, Perl, Python, y muchos otros. Estas propiedades hacen de JSON un lenguaje ideal para el intercambio de datos.

**JSON está construido sobre dos estructuras:**

**1. Una colección de pares de nombre/valor.** En varios lenguajes, esto se realiza mediante un objeto, un registro, una estructura, un diccionario, una tabla hash, una lista con claves o un arreglo asociativo.

**2. Una lista ordenada de valores.** En la mayoría de los lenguajes, esto se implementa como un arreglo, vector, lista o secuencia.

Estos son algunos puntos clave sobre JSON:

- **Sintaxis Universal:** Aunque JSON se deriva de la sintaxis de JavaScript, es independiente del lenguaje y se puede usar en muchos entornos de programación.
- **Datos en Formato de Texto:** JSON se representa como un string, lo que facilita su paso a través de sistemas y su almacenamiento en texto.
- **Facilidad de Uso:** JSON es fácil de leer y escribir para humanos y fácil de analizar y generar para máquinas.
- **Estructura de Datos:** Utiliza una estructura de datos compuesta por arrays y objetos, lo que lo hace muy versátil.

Un ejemplo de un objeto JSON podría ser:

```
{
  "nombre": "Juan",
  "edad": 30,
  "esEstudiante": false,
  "habilidades": ["Java", "JavaScript", "C++"],
  "direccion": {
    "calle": "Av. Siempre Viva",
    "ciudad": "Springfield"
  }
}
```

Es importante recordar que JSON no es un tipo de dato propio en JavaScript; es simplemente un formato de texto que sigue una cierta estructura. Cuando trabajas con JSON en JavaScript, normalmente lo manejas como un string o como un objeto JavaScript después de parsearlo.

```
let miJSON = '{"nombre": "Juan", "edad": 30}';
console.log(typeof miJSON); // Esto mostrará "string"
```

```
let miObjeto = JSON.parse(miJSON);
console.log(typeof miObjeto); // Esto mostrará "object"
```

# Conversión de objetos y almacenamiento

## Convertir (parsear) de/hacia JSON

Cuando sea necesario enviar un objeto Javascript al servidor o almacenarlo en storage, será necesario convertirlo a un JSON (un string) antes de ser enviado.

Para eso usamos los siguientes métodos:



### Stringify (es un método de los objetos JSON)

Con `JSON.stringify` podemos transformar un objeto JavaScript a un string en formato JSON. (esto para poder guardarlo en local storage ya que solo admite texto)

```

const producto1 = { id: 2, producto: "Arroz" };
const enJSON    = JSON.stringify(producto1);

console.log(enJSON); // {"id":2,"producto":"Arroz"}
console.log(typeof producto1); // object
console.log(typeof enJSON); // string

localStorage.setItem("producto1", enJSON);
// Se guarda {"id":2,"producto":"Arroz"}
  
```

### Parse (es un método de los objetos JSON)

Con `JSON.parse` podemos transformar string en formato JSON a objeto JavaScript. (de local storage se obtiene un string y luego, para manipular esos datos hay que transformar ese string en un objeto JS).

```

const enJSON    = '{"id":2,"producto":"Arroz"}';
const producto1 = JSON.parse(enJSON);

console.log(typeof enJSON); // string
console.log(typeof producto1); // object
console.log(producto1.producto); // Arroz

const producto2 = JSON.parse(localStorage.getItem("producto1"));
console.log(producto2.id); // 2
  
```

```

const productos = [{ id: 1, producto: "Arroz", precio: 125 },
                  { id: 2, producto: "Fideo", precio: 70 },
                  { id: 3, producto: "Pan" , precio: 50},
                  { id: 4, producto: "Flan" , precio: 100}];

const guardarLocal = (clave, valor) => { localStorage.setItem(clave, valor) };
//Almacenar producto por producto
for (const producto of productos) {
    guardarLocal(producto.id, JSON.stringify(producto));
}
// o almacenar array completo
guardarLocal("listaProductos", JSON.stringify(productos));

```

```

class Producto {
    constructor(obj) {
        this.nombre = obj.producto.toUpperCase();
        this.precio = parseFloat(obj.precio);
    }
    sumaIva() {
        this.precio = this.precio * 1.21;
    }
}

//Obtenemos el listado de productos almacenado
const almacenados = JSON.parse(localStorage.getItem("listaProductos"));
const productos = [];
//Iteraremos almacenados con for...of para transformar todos sus objetos a tipo producto.
for (const objeto of almacenados)
    productos.push(new Producto(objeto));
//Ahora tenemos objetos productos y podemos usar sus métodos
for (const producto of productos)
    producto.sumaIva();

```

## Recuperar datos

Muchas veces usamos el Storage para recuperar datos relacionados a la última navegación del usuario. Por ejemplo, su última sesión de login o el último estado de su carrito de compras.

Para esto, pensamos en inicializar las variables de la app consultando el Storage en el momento de inicio.

```

let usuario;
let usuarioEnLS = JSON.stringify(localStorage.getItem('usuario'))

// Si había algo almacenado, lo recupero. Si no le pido un ingreso
if (usuarioEnLS) {
    usuario = usuarioEnLS
} else {
    usuario = prompt('Ingrese su nombre de usuario')
}

```

```

let carrito = []
let carritoEnLS = JSON.stringify(localStorage.getItem('carrito'))

// Inicializo mi app con carrito como array vacío o con el registro que haya quedado en LS
if (carritoEnLS) {
    carrito = carritoEnLS
}

// Función que renderizaría el carrito
renderCarrito(carrito)

```

## JSON: otros puntos a tener en cuenta

Las datos en formato JSON se pueden almacenar en archivos externos <b>.json</b> . Exemplo: <b>datos.json</b>	JSON es sólo un formato de datos - contiene sólo <b>propiedades</b> , no métodos.	Una coma o dos puntos mal ubicados pueden producir que un archivo JSON no funcione. Se debe ser cuidadoso para validar cualquier dato que se quiera utilizar. <a href="https://jsonformatter.curiousconcept.com/">https://jsonformatter.curiousconcept.com/</a>	A diferencia del código JavaScript en que las propiedades del objeto pueden no estar entre comillas, en JSON sólo las cadenas entre comillas pueden ser utilizadas como propiedades.

## LIBRERIAS

Son **códigos pre-escritos que facilitan el desarrollo de aplicaciones**. Podemos pensar las librerías como **pequeños programas escritos por terceros** que **podemos incorporar a nuestra aplicación** para resolver problemas determinados.

Funcionan como **cajas de herramientas que resuelven problemas recurrentes de forma rápida y eficiente**. Podemos incorporarlas y utilizarlas a discreción según nuestra demanda.

### Implementación

Las librerías **se incorporan al proyecto como archivos** . Se vinculan a nuestra aplicación en el HTML como cualquier otro **script** de Javascript.

Puede ser con la descarga de los archivos de la librería:

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

Y también **puede ser a través de un CDN**:

```
src="https://cdnjs.cloudflare.com/ajax/libs/moment.js/2.29.1/moment.min.js"
</script>
```

## Documentación

La **documentación** es el **manual de instrucciones** para la implementación y manejo de las librerías . Es muy importante trabajar con ella.

**Getting Started**

Welcome to the Next.js documentation!

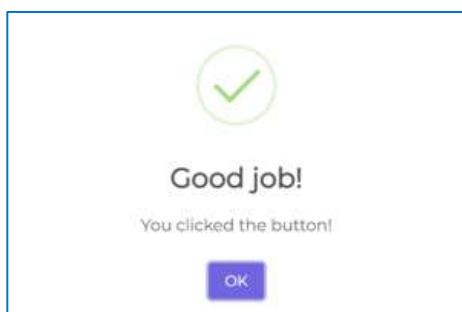
If you're new to Next.js we recommend that you start with the [learn course](#).

The interactive course with quizzes will guide you through everything you need

**Un buen desarrollador se destaca por poseer la habilidad de trabajar con estos tipos de documentos y herramientas.**

## Sweet alert

Permite crear **alertas personalizadas** atractivas, sencillas, customizables e interactivas. Reemplaza el típico **alert() tradicional** .



### ¿Cómo instalarla?

1

Primero debemos **implementarla** como vimos anteriormente.

2

Luego, siguiendo la **documentación**, en el ítem **installation** tenemos las instrucciones de cómo proceder:

3

¡Y listo! Una vez agregada, ya podemos utilizarla .

```
<script src="//cdn.jsdelivr.net/npm/sweetalert2@11"></script>
```

## ¿Cómo usarla?

Ahora, podemos disparar alertas a través del método `.fire` del objeto global **Swal**:

```
Swal.fire({
    title: 'Error!',
    text: 'Do you want to continue?',
    icon: 'error',
    confirmButtonText: 'Cool'
})
```

El método recibe un **objeto por parámetro**. Puede recibir distintas propiedades y valores, generando distintos resultados. Esto nos permite customizar la alerta.

### Algunos TIPS de Sweet Alert

Sweet Alert viene estilado con **bootstrap**, por lo que se recomienda tenerlo integrado para lograr un mejor resultado visual.

En la **documentación** se listan todas las propiedades y valores posibles que se pueden definir para configurar el alert. Y en su sección “Recipe Gallery” hay muchos ejemplos.

Es importante definir siempre: los **eventos** y el **comportamiento** esperado en nuestro script.

## JS ASINCRÓNICO

### ASINCRONÍA Y PROMESAS

#### MODELOS DE PROGRAMACIÓN SINCRÓNICA Y ASINCRÓNICA

##### Programación sincrónica

En este modelo, nuestro programa funciona de manera **lineal, ejecutando una acción y después otra**. Sólo podemos realizar **una tarea a la vez y cada tarea es bloqueante de la siguiente**.

##### 1era. petición



##### 2da. petición



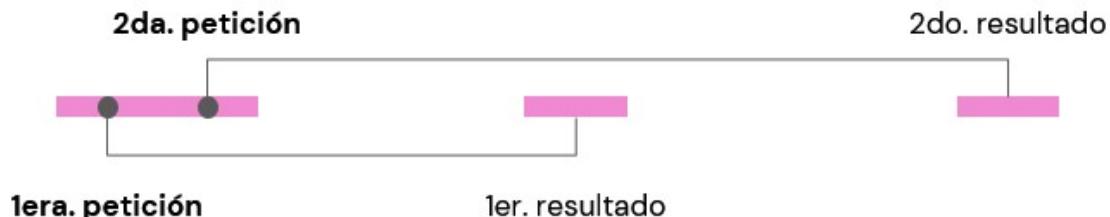
1er. resultado

2do. resultado

##### Programación asincrónica

Este modelo permite que **múltiples tareas sucedan a la vez**. Al comenzar una acción, nuestro programa sigue en ejecución; y cuando la

acción termina nuestro programa es informado y consigue acceso al resultado correspondiente.



👉 Una de las principales ventajas del modelo asincrónico: facilita el manejo de programas que realizan múltiples acciones a la vez.

👉 Uno de sus principales riesgos: puede dificultar la comprensión de aquellos programas que tienden a seguir una única línea de acción

### SETTIMEOUT

Es una **función** que permite realizar acciones asincrónicamente. **La función recibe dos parámetros:**

**Una función de callback y un valor numérico que representa milisegundos.**

```
setTimeout(fn, time)
```

Así, **la función que pasamos por primer parámetro se ejecuta luego de que transcurra el tiempo definido en el segundo parámetro.**  
Por ejemplo:

```
setTimeout(()=> {
  console.log("Proceso asincrónico")
}, 3000)
```

En modelo sincrónico, esperaríamos ver el próximo ejemplo en el siguiente orden:

1. “Inicia proceso”
2. “Mitad de proceso” (tras 2 segundos)
3. “Fin proceso”

```
console.log("Inicia proceso")

setTimeout(()=> {
  console.log("Mitad de proceso")
}, 2000)

console.log("Fin proceso")
```

Sin embargo, la salida se produce de la siguiente forma:

```
// Inicia proceso
// Fin proceso

// Mitad de proceso - tras 2 segundos
```

Esto sucede porque **setTimeout** funciona de forma asincrónica. Por eso es que los dos **console.log** se ejecutan antes, y por último vemos el resultado del setTimeout que va en el medio.

**Ahora bien, podríamos suponer que un setTimeout con 0 milisegundos se ejecutaría de forma inmediata**, sin irrumpir el orden sincrónico del programa. Pero vemos que el efecto sigue siendo igual que antes:

<pre>console.log("Inicia proceso") setTimeout(()=&gt; {     console.log("Mitad de proceso") }, 0)  console.log("Fin proceso")</pre>	Inicia proceso Fin proceso Mitad de proceso
---	---

**Lo visto en el ejemplo no se explica precisamente con el tiempo de ejecución del proceso, sino con el orden que ocupa en el listado de peticiones a ejecutar.**

Para ello, debemos entender **Call Stack** y **Event loop**.

### CALL STACK (PILA DE LLAMADAS A FUNCIONES)

Cuando se habla de "call stack" en programación, se está haciendo referencia a esta **estructura de datos específica que gestiona el flujo de ejecución de llamadas a funciones dentro de un programa**.

Es una **lista donde se apilan las distintas tareas a ejecutar por nuestro programa**. Javascript es un **lenguaje single threaded**, o de un único hilo, lo que significa que **tiene un único stack o pila de ejecución**. De ahí que la ejecución es implícitamente sincrónica.

¿Cómo es el proceso de **Call Stack**?

**Cuando se está a punto de ejecutar una función, ésta es añadida al stack. Si la función llama a la vez, a otra función, ésta es agregada sobre la anterior:**

```

function multiply (x, y) {
    return x * y;
}

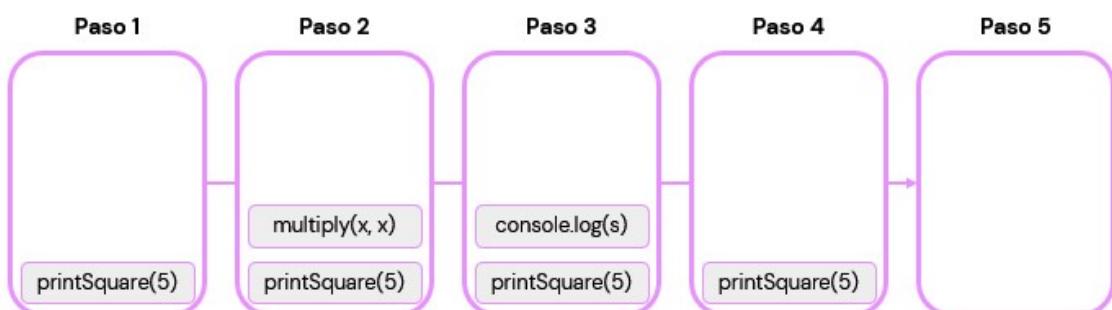
function printSquare (x) {
    let s = multiply(x,
    x);
    console.log(s);
}

printSquare(5);

```

El "stack", o pila, en el contexto de la programación y en particular en JavaScript, es una estructura de datos que sigue el principio de LIFO (Last In, First Out), lo que significa "El último en entrar es el primero en salir". Esta estructura es utilizada para gestionar la ejecución de funciones y el seguimiento de sus contextos de ejecución en el call stack (pila de llamadas).

Los estados de Call Stack son:



Es una **lista de tareas** de JS a ejecutar durante el programa. Cada nueva instrucción se agrega en el orden que corresponde al stack y el motor de JS resuelve una a una.

## EVENT LOOP

**Muchas funciones asincrónicas se ejecutan en un stack diferente.** El Event Loop es la herramienta que permite la sincronización entre nuestro call stack con estas tareas asincrónicas que funcionan en un thread aparte.

**Si el stack está vacío, el Event Loop envía la primera función que esté en la **callback queue** al call stack y comienza a ejecutarse.**

Cuando hablamos de la "**callback queue**" en JavaScript, nos referimos específicamente a la cola de funciones de callback que han completado su ejecución en el ambiente asíncrono (como operaciones de I/O, temporizadores, solicitudes de red, etc.) y están esperando ser transferidas al call stack para su ejecución.

Así, entendemos cómo funciona de tal manera una instrucción como la siguiente:

```

console.log("Inicia
proceso")

setTimeout(()=> {
    console.log("Mitad de
proceso")
}, 0)

console.log("Fin proceso")

// Inicia proceso
// Fin proceso
// Mitad de proceso

```

Por más que hagamos el timeout con 0 milisegundos, éste se envía al stack de **web apis** primero, luego al **callback queue**, y finalmente el event loop lo incorpora al **callstack** para su ejecución, luego de los console.log anteriores ✨

### SETINTERVAL

Tiene la misma sintaxis que setTimeout, pero la **unidad de tiempo es un intervalo para la repetición de la función asociada**:

```

setInterval(() => {
    console.log("Tic")
}, 1000)

```

Permite **ejecutar funciones de manera reiterativa** tras los milisegundos indicados hasta que indiquemos su detención o se cierre la aplicación.

### CLEARINTERVAL & CLEARTIMEOUT

En caso de querer **remover un Intervalo**, utilizamos la función **clearInterval ()**.

También podemos **detener la ejecución de un setTimeout** invocando **clearTimeout ()**.

Cuando llamamos un setInterval() éste retorna una referencia al intervalo generado, el cual podemos almacenar en una variable. Es esta referencia la que debemos pasar a la función clearInterval para que la limpieza tenga efecto:

```

let counter = 0
const interval = setInterval(() => {
    counter++
    console.log("Counter: ", counter)

    if (counter >= 5) {
        clearInterval(interval)
        console.log("Se removió el intervalo")
    }
}, 1000)

```

Funciona igual con los timeout. Si guardamos en una variable la referencia al timeout generado, podemos usarla para removerlo luego. En el siguiente caso, el timeout generado nunca llega a ejecutarse:

```

console.log("Inicio")

const fin = setTimeout(() => {
    console.log("fin")
}, 2000)

clearTimeout(fin)

```

## PROMESAS

### ¿QUÉ ES UNA PROMESA EN JS?

**Es un objeto de Javascript que representa un evento a futuro (por lo que requiere que exista siempre un tiempo de espera).** Es una acción asincrónica que se puede completar en algún momento y producir un valor, y notificar cuando esto suceda.

**Se puede completar:** Una promesa en JavaScript representa una operación que puede completarse en el futuro, ya sea exitosamente (cumplida) o con un error (rechazada). Esto indica la posibilidad y la intención de que la acción asíncrona llegue a un resultado final, pero no garantiza cuándo sucederá.

**Las Promesas en JavaScript son un mecanismo para manejar operaciones asíncronas.** Permiten manejar el resultado de una acción que tomará algún tiempo en completarse, como la carga de datos desde un servidor, de una manera más flexible y legible que las técnicas más antiguas, como los callbacks.

**Objeto Promesa:** Una promesa es un objeto que representa la eventual finalización (o falla) de una operación asíncrona y su resultado. Tiene tres estados:

**Pending (Pendiente):** Estado inicial, ni cumplida ni rechazada.

**Fulfilled (Cumplida):** Significa que la operación se completó satisfactoriamente.

**Rejected (Rechazada):** Significa que la operación falló.

**Thenable:** Se refiere a cualquier objeto o función que tiene un método `.then`. Las promesas pueden encadenarse gracias a este método.

Podemos **crear promesas** a través de su constructor `new Promise`. Su sintaxis es algo compleja, ya que **recibe una función ejecutora como argumento**. Esta función ejecutora **se ejecuta inmediatamente por el constructor de la Promesa y recibe dos funciones como argumentos, resolve y reject**:

```
let promesa = new Promise((resolve, reject) => {
    // Operación asíncrona aquí
    if /* operación exitosa */ {
        resolve(valor); // Cumple la promesa
    } else {
        reject(error); // Rechaza la promesa
    }
});
```

**No es normal que nosotros creamos promesas; lo normal es trabajar con promesas de otros.**

### Resolve & Reject

En principio, una promesa se retorna con estado **pending**, entendiendo que el valor a generar aún no fue resuelto :

```
const eventoFuturo = () => {
    return new Promise( (resolve, reject) => {
        //cuerpo de la promesa
    })
}
console.log( eventoFuturo() ) // Promise { <pending> }
```

Esta función retorna una promesa que **no se resuelve**. Por lo tanto, veremos que el valor que genera es un objeto Promise con estado pendiente.

El valor de retorno de la promesa se define a través del llamado a las funciones de **resolve** o **reject**:

- ✓ Si el cuerpo de la promesa llama a resolve(), la promesa cambiará su estado a **fulfilled**, con el valor enviado a resolve().
- ✓ Si la promesa llama a reject(), cambiará su estado a **rejected** con el valor enviado al reject().

Aquí podemos ver cómo cambia de estado la promesa con distintos valores. Según el llamado de la función la promesa se verá **resuelta** o **rechazada**. Sin embargo, lo que vemos por consola es el objeto Promise que retorna la función, y con lo que nos interesa trabajar en realidad es con el **valor** de resolución de la promesa.

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

console.log( eventoFuturo(true) ) // Promise { 'Promesa resuelta' }
console.log( eventoFuturo(false) ) // Promise { <rejected> 'Promesa rechazada' }
```

Miremos el mismo caso agregando un delay con **setTimeout**:

```
const eventoFuturo = (res) => {
    return new Promise( (resolve, reject) => {
        setTimeout( () => {
            res ? resolve('Promesa resuelta') : reject('Promesa rechazada')
        }, 2000)
    })
}

console.log( eventoFuturo(true) ) // Promise { <pending> }
console.log( eventoFuturo(false) ) // Promise { <pending> }
```

En este caso, el console.log es sincrónico y vemos que la promesa está en **pending** en ambos llamados (su resolución se generará dentro de 2s). Las promesas tienen un mecanismo para trabajar esta asincronía y poder ejecutar funciones cuando cambie su estado.

## Then & Catch

**Para consumir una promesa y actuar sobre su resultado o posible error**, se utilizan los métodos `.then()` y `.catch()`:

Al llamado de una función que retorne una promesa, podemos concatenar el método **.then()** o **.catch()**, los cuales reciben una función por parámetro con la cual se captura el valor de la promesa:

- ✓ **.then()** : Si la promesa es resuelta , su valor de retorno se captura dentro del `.then()`, recibiendo por parámetro de su función ese valor.
- ✓ **.catch()** : si la promesa es rechazada , su valor se captura dentro de un `.catch()` siguiendo la misma lógica.

Lo que queramos ejecutar cuando la promesa se **resuelva** o **rechace**, debemos definirlo dentro de un `.then()` o `.catch()`, según el caso:

```
eventoFuturo(true)
    .then( (response) => {
        console.log(response) // Promesa resuelta
    })

eventoFuturo(false)
    .catch( (error) => {
        console.log(error) // Promesa rechazada
    })
```

Se aprecian los `console.log` tras 2 segundos de delay y lo que vemos es, precisamente, el **valor** que retornan el `resolve` o `reject` de la promesa.

Como una promesa puede tener varios estados posibles, se puede concatenar varios **.then()** o **.catch()** en un mismo llamado, y caeremos en el caso que corresponda según cómo se haya resuelto la promesa:

```
eventoFuturo(true)
    .then( (response) => {
        console.log(response) // Promesa resuelta
    })
    .catch( (error) => {
        console.log(error)
    })

eventoFuturo(false)
    .then( (response) => {
        console.log(response)
    })
    .catch( (error) => {
        console.log(error) // Promesa rechazada
    })
```

### **Esto quiere decir que:**

Para cada promesa podemos definir una estructura para trabajar los distintos casos posibles. Cada promesa sólo puede **resolverse o rechazarse** una única vez. Es un mecanismo de control claro y ordenado para trabajar la asincronía y los posibles valores a recibir.

### **Finally**

**finally()** es un método que recibe una función la cual se ejecutará siempre al finalizar la secuencia, sin importar si se haya resuelto o no la promesa:

```
eventoFuturo(true)
    .then( response) => {
        console.log(response)
    })
    .catch( error) => {
        console.log(error)
    })
    .finally( () => {
        console.log("Fin del proceso")
    })
// Promesa resuelta
// Fin del proceso
```

### **Ejemplos de Uso**

- Operaciones de I/O (como lecturas de archivos en Node.js o solicitudes HTTP en el navegador).
- Acciones que requieren esperar por la disponibilidad de un recurso.
- Secuenciación de tareas asíncronas que dependen unas de otras.
- 

### **Conclusiones**

Las promesas **son una parte fundamental de la programación asíncrona** en JavaScript, **proporcionando una manera poderosa y flexible de trabajar con operaciones que toman tiempo en completarse**. Su diseño permite escribir código asíncrono que es tanto fácil de entender como de mantener, esencial en el desarrollo moderno de aplicaciones web.

## EJEMPLO ASÍNCRONÍA Y PROMESAS

### **SIMULANDO PETICIÓN DE DATOS**

En este ejemplo se va a simular cómo es la **peticIÓN dE dATOS** a algún servidor y generar alguna interacción con el resultado.

En el ejemplo tenemos un array de productos vacíos. Al cargar la aplicación queremos simular un delay para la actualización de estos

datos y lo haremos llamando una promesa que retorna el array de productos real. Cuando captemos su resolución, en el .then() actualizamos nuestro array y llamamos una función para generar la vista del resultado.

```
// función que tras 3 segundo retorna un array de objetos

const BD = [
    {id: 1, nombre: 'Producto 1', precio: 1500},
    {id: 2, nombre: 'Producto 2', precio: 2500},
    {id: 3, nombre: 'Producto 3', precio: 3500},
    {id: 4, nombre: 'Producto 4', precio: 3500},
]

const pedirProductos = () => {
    return new Promise( (resolve, reject) => {
        setTimeout(() => {
            resolve(BD)
        }, 3000)
    })
}
```

```
// inicializamos con un array vacío
let productos = []

const renderProductos = (arr) => {
    // función que genere la vista de los productos
}

// asincrónicamente pedimos los datos y generamos la vista
pedirProductos()
    .then((res) => {
        productos = res
        renderProductos(productos)
    })
}
```

## AJAX

En los 2000, los desarrolladores web se encontraban ante el constante desafío de obtener o enviar información sin afectar el estado actual de la página (sin requerir una recarga completa refresco de la página).

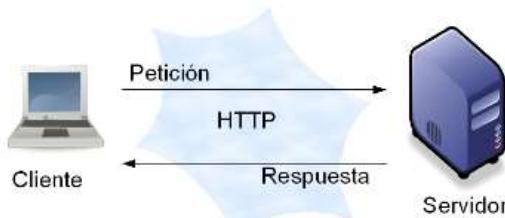
Para dar respuesta a este problema, surgió **AJAX (Asynchronous JavaScript and XML)** es una **técnica de desarrollo web utilizada para crear aplicaciones interactivas mediante la combinación de varias tecnologías, incluyendo JavaScript, XML (aunque ahora se utiliza más comúnmente JSON), HTML y CSS.** Permite a las páginas web enviar y recibir datos del servidor de forma asíncrona, es decir, sin tener que recargar toda la página.

En consecuencia, **cualquier app o web que emplee AJAX puede enviar y recibir datos sin volver a cargar toda la página**, evitando la interrupción de acciones realizadas por el usuario, **añadiendo interactividad y dinamismo a nuestra aplicación**. Esto hace a las características esenciales del software moderno.

En lugar de cargar una página completa cada vez que se necesita interactuar con el servidor, **AJAX permite realizar solicitudes al servidor en segundo plano, generalmente utilizando el objeto XMLHttpRequest en JavaScript, y luego actualizar partes específicas de la página con la respuesta del servidor.** Esto resulta en una experiencia de usuario más fluida y rápida, ya que solo se actualizan los elementos necesarios en lugar de recargar toda la página.

### MODELO CLIENTE-SERVIDOR

En el modelo cliente servidor, **el cliente envía un mensaje solicitando un determinado servicio a un servidor (hace una petición), y este envía uno o varios mensajes con la respuesta (provee el servicio)** (Ver Figura). En un sistema distribuido cada máquina puede cumplir el rol de servidor para algunas tareas y el rol de cliente para otras.



**Cliente:**

El cliente es el **proceso que permite al usuario formular los requerimientos y pasarlo al servidor, se le conoce con el término front-end**. El Cliente normalmente maneja todas las funciones relacionadas con la manipulación y despliegue de datos, por lo que están desarrollados sobre plataformas que permiten construir interfaces gráficas de usuario (GUI), además de acceder a los servicios distribuidos en cualquier parte de una red.

**El cliente en el modelo cliente-servidor no está limitado únicamente a los navegadores web. También puede ser una aplicación móvil, un programa de escritorio**, un dispositivo IoT (Internet de las cosas) u otro tipo de aplicación que solicite y consuma servicios o recursos de un servidor a través de una red.

Las funciones que lleva a cabo el proceso cliente se resumen en los siguientes puntos:

- Administrar la interfaz de usuario.
- Interactuar con el usuario.
- Procesar la lógica de la aplicación y hacer validaciones locales.
- Generar requerimientos de bases de datos.
- Recibir resultados del servidor.
- Formatear resultados.

### **Servidor:**

Es el **proceso encargado de atender a múltiples clientes que hacen peticiones de algún recurso administrado por él**. Al proceso servidor se le conoce con el término **back-end**. El servidor normalmente maneja todas las funciones relacionadas con la mayoría de las reglas del negocio y los recursos de datos.

### **Un servidor puede ser:**

**1. Un Servidor remoto en la web** → un servidor puede ser una computadora remota en la web. Estos servidores están ubicados en centros de datos o en la infraestructura de proveedores de servicios en la nube. Son accesibles a través de Internet y ofrecen una amplia gama de servicios, como hosting de sitios web, almacenamiento de datos, servicios de correo electrónico, aplicaciones en la nube, entre otros)

**2. Servidor local o en tu propia computadora** → puedes configurar tu propia computadora para que actúe como un servidor. Esto puede ser útil para el desarrollo y pruebas locales de aplicaciones web, hosting de sitios web en un entorno controlado, compartir archivos en una red local, ejecutar servidores de juegos, o cualquier otra necesidad que requiera ofrecer servicios a otros dispositivos en tu red local. Por ejemplo, puedes instalar software de servidor web como Apache, Nginx o Microsoft IIS para alojar sitios web en tu propia computadora.

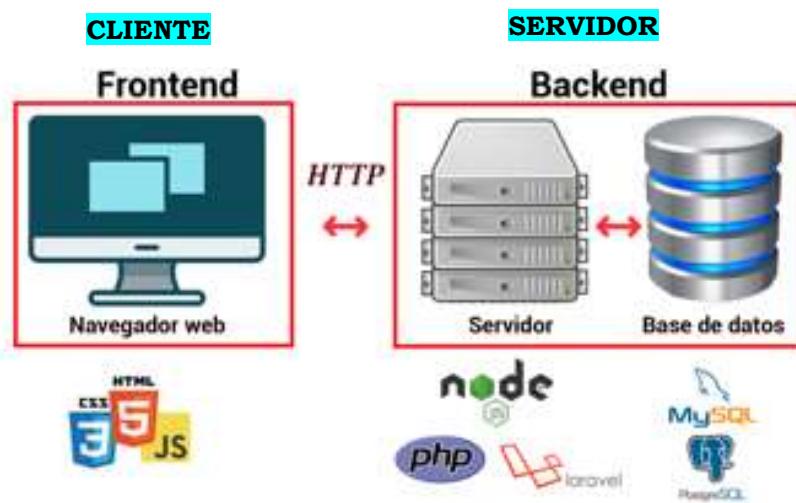
En resumen, un servidor puede ser una computadora remota en la web, alojada en un centro de datos o en la infraestructura de la nube, o puede ser tu propia computadora configurada para proporcionar servicios a otros dispositivos en una red local o a través de Internet. La elección depende de las necesidades específicas de tu proyecto o aplicación.

Las funciones que lleva a cabo el proceso servidor se resumen en los siguientes puntos:

- Aceptar los requerimientos de bases de datos que hacen los clientes.
- Procesar requerimientos de bases de datos.
- Formatear datos para trasmitirlos a los clientes.
- Procesar la lógica de la aplicación y realizar validaciones a nivel de bases de datos.

### **Repasemos el funcionamiento de una aplicación web:**

1. Se accede a las mismas mediante conexión a internet, empleando un navegador y referenciando la **dirección web del sitio (también llamado dominio**, por ejemplo <https://www.coderhouse.com/> ).



**2. El dominio está asociado a un servidor**, es decir, **un ordenador que tiene la aplicación web (el servidor aloja la aplicación web)**. **La comunicación entre Cliente y Servidor (pedido y envío de datos) se realiza a través del protocolo HTTP.**

**3. Cuando carga el sitio, el usuario visualiza la parte frontal de la aplicación llamada front-end o lado del cliente**, con la que puede interactuar.

**4. Las apps consumen recursos provistos por algún servidor (back-end), o envía datos a éste para almacenarlos de forma persistente.** Las apps modernas suelen generar **experiencias de usuario enriquecidas** gracias a su conexión a servicios de datos.

Es fundamental aprender a dominar los métodos para realizar este intercambio de información y comprender el protocolo implicado.

## PETICIONES (REQUEST) HTTP (COMUNICACIÓN CON EL SERVIDOR)

El mecanismo por el cual se piden y proveen datos a través de internet es **HTTP** (Hypertext Transfer Protocol).

Cuando emitimos una orden al navegador, hace una petición (o request) HTTP a algún servidor. Luego, la recibirá, procesará y nos devolverá una respuesta con información que utilizaremos en la aplicación.

Estas peticiones que debemos hacer están definidas por varias partes:



- ✓ Una URL o dirección.
- ✓ Un método (GET, POST, PUT, DELETE).
- ✓ Headers.
- ✓ Body.
- ✓ Parámetros (Query Params o URL Params)

### **URL:**



Cuando nos comunicamos con un servidor para pedir información lo hacemos a través de una URL, ya que éste es un programa alojado en algún host y nos comunicamos con él a través de la dirección correcta.

Una URL (Uniform Resource Locator) es una cadena de caracteres que se utiliza para **especificar la ubicación de un recurso en la web, como una página web, un archivo, una imagen, etc. Básicamente, una URL es la dirección que se utiliza para acceder a recursos en internet.**

### **MÉTODO:**

**Cada petición que hacemos está acompañada por un verbo que indica al servidor cuál es nuestra intención.** El servidor tiene la capacidad de escuchar distintas peticiones en la misma URL, decidir a cuál responder y cómo. Son 4 los verbos más utilizados, aunque hay muchos más: **Get, Post, Put & Delete.**

- **GET:** Para obtener información (o recurso) del servidor. Suelen ser las más utilizadas.
- **POST:** Para enviar información al servidor para crear algún recurso.
- **PUT:** Para crear o modificar algún recurso en el servidor.

- **DELETE:** Para eliminar algún recurso en el servidor.

Las peticiones de tipo **POST** y **PUT** van acompañadas de un **body** (cuerpo de la request) **donde se definen los datos o información a enviar al servidor**. GET o DELETE, por su parte, no tienen body ya que no necesitan enviar datos adjuntos.

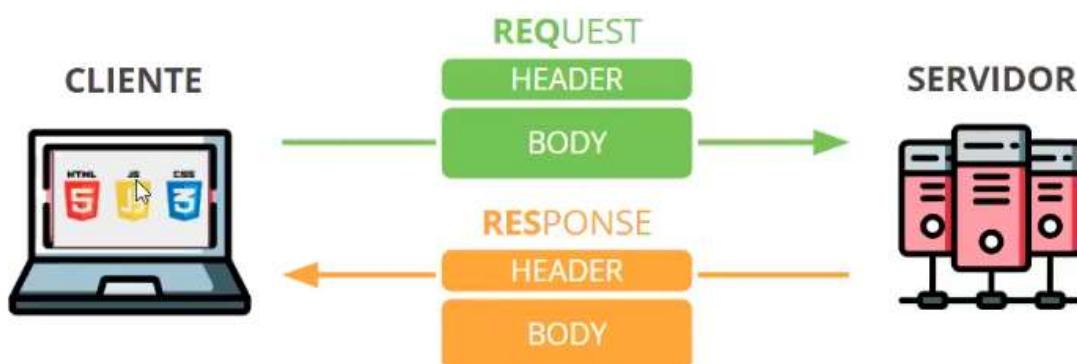
### BODY:

Es el espacio en la petición donde se definen **los datos a enviar al servidor**.



### HEADERS:

Las cabeceras (headers) HTTP permiten al cliente y servidor enviar información sobre la petición y la respuesta. Los headers incluyen información sobre la petición para establecer una **transferencia segura y clara**, y de ser necesario se pueden modificar para agregar datos adicionales. No debemos confundir información sobre la petición (headers) con los datos que la petición puede transferir (body).



General	
Request URL:	https://developer.mozilla.org/static/media/menu-open.c1036.svg
Request Method:	GET
Status Code:	200
Remote Address:	18.65.48.76:443
Referrer Policy:	strict-origin-when-cross-origin
Response Headers	
age:	2761636
cache-control:	max-age=31536000, public
content-length:	271
content-type:	image/svg+xml
date:	Sat, 08 Jan 2022 21:14:31 GMT
etag:	"6ea943d3661a5c64f196863cc4595859"
last-modified:	Fri, 07 Jan 2022 21:14:31 GMT

## PARÁMETROS:

Para especificar una petición, se puede enviar información adicional en la forma de parámetros a través de la URL. Tenemos dos formas de definir parámetros a través de la URL:

- Query params
- URL params

### ✓ QUERY PARAMS:

Esta sintaxis permite adjuntar en la URL una serie de parámetros en la forma de pares **clave=valor**. Por ejemplo, si queremos buscar algo por google, debemos enviarle un valor de búsqueda por el parámetro q, a través de la url:

<https://www.google.com.ar/search?q=coderhouse>

Se utiliza el símbolo **?** para indicar el final de la parte de la dirección de la url y el comienzo del query. A partir de allí, se escriben parámetros con la forma clave=valor, pudiendo definir varios separándolos con el signo ampersand (**&**).

Por ejemplo, en la siguiente URL se hace una consulta a la PokeApi (<https://pokeapi.co/docs/v2>), pidiendo información al endpoint de /pokemon, y se envían los parámetros offset=0 y limit=20 :

<https://pokeapi.co/api/v2/pokemon?offset=0&limit=20>

Esto condiciona la búsqueda que queremos hacer en ese servidor.

### ✓ URL PARAMS:

Esta sintaxis permite enviar parámetros directamente en la forma de segmentos de la URL, es decir separados por **/** . Por ejemplo, la PokeApi nos indica lo siguiente:

<https://pokeapi.co/api/v2/pokemon/{id or name}/>

Significa que ese **{id or name}** es un parámetro, un valor dinámico que insertamos en la URL, en este caso para obtener información sobre un pokemon según su ID o nombre. Para obtener aquel con id = 1, haríamos una petición GET a la siguiente url:

<https://pokeapi.co/api/v2/pokemon/1>

## APIS

Una **API (Application Programming Interfaces)** es una **aplicación web construida en base a la arquitectura API REST a la cual podemos solicitar y enviar información desde el cliente**. Generalmente, nos comunicamos con aplicaciones de este tipo y es la tendencia actual de desarrollo.

La ventaja de este modelo es que está **orientado a recursos** y define métodos claros para solicitar y enviar información.

Hay muchas APIs disponibles que podemos utilizar para acceder a distintos **recursos útiles** para nuestra aplicación: Servicios de contenido (CMS), Plataformas de pago, Servicios de e-mail, etcétera.

Incluso hay APIs creadas como bancos de información sobre series y videojuegos populares, como la PokeApi ([Documentation - PokéAPI](#)) o Star Wars API ([SWAPI](#)).

**API REST** → describe cualquier interfaz entre sistemas que utilice directamente [HTTP](#) para obtener datos o indicar la ejecución de operaciones sobre los datos, en cualquier formato ([XML](#), [JSON](#), etc) sin las abstracciones adicionales de los protocolos basados en patrones de intercambio de mensajes, como por ejemplo [SOAP](#).

## API

Una API suele tener una **URL base** (el dominio donde está alojada la aplicación) y luego puede tener varios **endpoints**, es decir, distintas secciones a las que podemos acceder.  
A la vez, se pueden hacer peticiones con distintos métodos al mismo endpoint y **obtener distintos resultados**.



Generalmente, similar a cuando queremos incorporar una librería, al momento de consumir una API **debemos revisar su documentación**. Allí se definen los distintos endpoints disponibles, los métodos a utilizar para hacer una petición y qué se nos ofrecerá en respuesta.

## FETCH

Javascript nos ofrece el **método fetch() para hacer peticiones HTTP a algún servicio externo**. Como estas peticiones son asincrónicas, convenientemente el método fetch() trabaja con **promesas**.

```
fetch(url, config)
```

El método recibe un primer parámetro que es la URL a la cual hacer la petición, y un segundo parámetro opcional de configuración.

Para los siguientes ejemplos utilizaremos la API de [JSON Placeholder](#), diseñada para hacer pruebas de peticiones simulando un listado de posts.

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

Por defecto el método fetch hace peticiones del tipo GET. Según la documentación, para obtener una lista de posts debemos hacer una petición del siguiente tipo.

Vayamos por parte para entender este proceso. Primero, llamemos al método con la URL correspondiente y veamos qué retorna:

```
console.log( fetch('https://jsonplaceholder.typicode.com/posts') )
// Promise {<pending>}
```

Retorna una Promesa pendiente. Para trabajar con la resolución de la petición, debemos hacerlo dentro del .then() correspondiente:

```
fetch('https://jsonplaceholder.typicode.com/posts')
  .then( (resp) => console.log(resp) )
```

Haciendo **console.log** de la respuesta, no vemos el listado de posts que esperamos sino un objeto del tipo Response.



```
fetch.js:6
Response {type: 'cors', url: 'https://jsonplaceholder.typicode.com/posts', redirected: false, status: 200, ok: true, ...}
```

## Response

Llamar a `fetch()` retorna una promesa que resuelve en un objeto `Response` que contiene información sobre la respuesta del servidor, como su código de estado y headers.

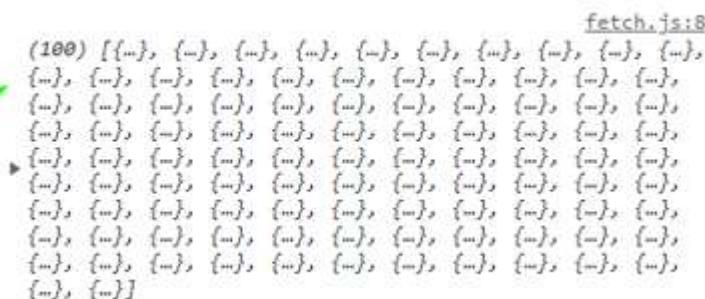
Para acceder al contenido de la respuesta debemos dar un paso adicional, y por eso es que se ven dos `.then()` concatenados.

Generalmente, se transfieren datos en formato JSON. Por lo tanto, para obtener el contenido de la respuesta debemos aplicar el método `.json()` a ese objeto. Éste retorna a su vez una Promesa, por lo que

capturamos su contenido (los datos enviados por la API) **en un segundo .then()** :

```
fetch('https://jsonplaceholder.typicode.com/posts')
  .then( (resp) => resp.json() )
  .then( (data) => {
    console.log(data)
  })
```

En el parámetro **data** tenemos el contenido de la respuesta de nuestra petición. En este caso, la API nos responde con un array de 100 elemento donde cada elemento es un post.



```
fetch.js:8
(100) [{}]
  ↪ 0: {userId: 1, id: 1, title: 'sunt aut facere repellat provi...', body: 'est rerum tempore vitae\nsequi sint nihil reprehend...', id: 2, title: 'qui est esse', userId: 1}
  ↪ 1: {userId: 2, id: 2, title: 'et iusto sed quo iure\nvoluptatem occaecati omnis e...', id: 3, title: 'ea molestias quasi exercitationem repellat qui ips...', userId: 1}
```

## ANALIZANDO RESPUESTAS

Trabajar con APIs nos ofrece un entorno claro sobre cómo comunicarnos y obtener respuestas con recursos. Sin embargo, cada API define qué responder, qué formato darle a los datos que envía y cómo estructurarlos.

Por lo tanto, siempre debemos analizar las respuestas obtenidas para ver qué datos utilizar de ellas.

En este caso, veamos cómo son los objetos del array anterior:

```
(100) [{}]
  ↪ 0: {userId: 1, id: 1, title: 'sunt aut facere repellat provi...', body: 'est rerum tempore vitae\nsequi sint nihil reprehend...', id: 2, title: 'qui est esse', userId: 1}
  ↪ 1: {userId: 2, id: 2, title: 'et iusto sed quo iure\nvoluptatem occaecati omnis e...', id: 3, title: 'ea molestias quasi exercitationem repellat qui ips...', userId: 1}

  ↪ 0: {userId: 1, id: 1, title: 'sunt aut facere repellat provi...', body: 'est rerum tempore vitae\nsequi sint nihil reprehend...', id: 2, title: 'qui est esse', userId: 1}
  ↪ 1: {body: "est rerum tempore vitae\nsequi sint nihil reprehend...", id: 2, title: "qui est esse", userId: 1}
  ↪ [[Prototype]]: Object
  ↪ 2: {body: "et iusto sed quo iure\nvoluptatem occaecati omnis e...', id: 3, title: "ea molestias quasi exercitationem repellat qui ips...", userId: 1}
```

Cada elemento tiene propiedades body, id, title, userId. Estamos trabajando con el parámetro data definido que es un array de objetos. Por ello, podemos recorrerlo y acceder a sus objetos y propiedades.

```
fetch('https://jsonplaceholder.typicode.com/posts')
  .then( (resp) => resp.json() )
  .then( (data) => {
    console.log( data[0].title )
    console.log( data[0].body )
  })

```

sunt aut facere repellat provident occaecati excepturi optio reprehenderit	fetch.js:8
quia et suscipit suscipit recusandae consequuntur expedita et cum	fetch.js:9
reprehenderit molestiae ut ut quas totam	
nostrum rerum est autem sunt rem eveniet architecto	

Teniendo esto disponible dentro del **.then()**, ¡podemos volcarlo al DOM utilizando los métodos vistos previamente!

```
<h2>Posts!</h2>
<hr/>

<ul id="listado">
</ul>
```

Al obtener la respuesta de la API, recorremos el array obtenido y agregamos a la **<ul>** un elemento **<li>** con el contenido de cada post en el array.

```
// JS
const lista = document.querySelector('#listado')

fetch('https://jsonplaceholder.typicode.com/posts')
  .then( (resp) => resp.json() )
  .then( (data) => {

    data.forEach((post) => {
      const li = document.createElement('li')
      li.innerHTML =
        `<h4>${post.title}</h4>
        <p>${post.body}</p>
        `

      lista.append(li)
    })
  })

```

## ENVIANDO DATOS CON POST

La API de **JSON Placeholder** también nos permite simular peticiones POST, es decir, podemos hacer una petición para enviar datos a la API. Al ser una simulación, no se crean recursos realmente en el servidor, pero sí obtenemos una respuesta aceptando el POST.

Dijimos que el segundo parámetro del método fetch es un objeto de configuración. En éste podemos definir el método, los headers y el body

de la petición. Si bien fetch trae valores por defecto para esto (como el método que es GET), podemos modificarlo a discreción según sea necesario.

En este caso la documentación nos indica que para hacer un post debemos hacer **un fetch con las siguientes características:**

```
fetch('https://jsonplaceholder.typicode.com/posts',
{
    method: 'POST',
    body: JSON.stringify({
        title: 'Coderhouse',
        body: 'Post de prueba',
        userId: 1,
    }),
    headers: {
        'Content-type': 'application/json;
charset=UTF-8',
    },
})
.then((response) => response.json())
.then((data) => console.log(data))
```

En el objeto de configuración tenemos varias propiedades a definir:

- ✓ **method:** 'POST'. Significa que el método de la petición será POST  
⚠ Si no lo modificamos será de tipo GET por defecto.
- ✓ **headers:** En este caso se agrega una propiedad 'Content-type', con el valor que nos indica la documentación de la API ⚠ Si no se agrega la petición sería rechazada por el servidor.
- ✓ **body:** Aquí se adjuntan los datos a enviar al servidor. En este caso se envía un objeto con la forma { **title**, **body**, **userId** }. El body debe enviarse en formato JSON, por eso lo vemos envuelto en un JSON.stringify().

Por lo general, al hacer un POST obtenemos una respuesta que nos envía una copia del recurso creado en el servidor. La forma de trabajar la respuesta es la misma que la anterior:

```
fetch.js:33
{title: 'Coderhouse', body: 'Post de prueba', userId: 1, id: 101}
  ↴
  body: "Post de prueba"
  id: 101
  title: "Coderhouse"
  userId: 1
  ▶ [[Prototype]]: Object
```

## RUTAS RELATIVAS

**Si la URL utilizada no contiene el prefijo ‘https:’, la ruta es relativa.** Así, podemos hacer una petición a algún archivo local en formato JSON usando fetch.

Por ejemplo, creemos un **archivo data.json que simule un array de productos:**

```
// data.json
[
    {"nombre": "Producto 1", "precio": 1500, "id": 1},
    {"nombre": "Producto 2", "precio": 2500, "id": 2},
    {"nombre": "Producto 3", "precio": 3500, "id": 3},
    {"nombre": "Producto 4", "precio": 4500, "id": 4},
    {"nombre": "Producto 5", "precio": 5500, "id": 5}
]
```

Nótese que debe estar escrito con el formato json válido.

Ahora al momento de cargar la aplicación, podemos llamar a este archivo con fetch y generar una vista de forma asincrónica:

```
const lista = document.querySelector('#listado')

fetch('/data.json')
    .then( (res) => res.json())
    .then( (data) => {

        data.forEach((producto) => {
            const li = document.createElement('li')
            li.innerHTML =
                `

#### ${producto.nombre}



${producto.precio}



Código: ${producto.id}



---

` 

            lista.append(li)
        })
    })
}
```

Al ser un archivo local la respuesta es casi inmediata, pero sigue siendo un proceso asincrónico.



## ASYNC - AWAIT

Trabajar con promesas facilita mucho el control de los procesos asincrónicos. Sin embargo, en procesos extensos se puede dificultar el trabajo escribiendo todo dentro de varios `.then()`.

Por suerte, los desarrolladores de JS ya pensaron en esto y nos ofrecen una herramienta que nos permite trabajar las promesas como si escribiéramos código sincrónico: **async await**.

El método `fetch` retorna una promesa. De forma sincrónica, si guardamos esta promesa en una variable veremos la promesa pendiente, porque esto sucede sincrónicamente:

```
const resp =
fetch('https://jsonplaceholder.typicode.com/posts')
console.log(resp) // Promise {<pending>}
```

Significa que el `console.log()` no espera a que se resuelva la promesa de la línea anterior para ejecutarse.

La **sentencia await** nos permite establecer un punto de espera en el código. Aplicado como prefijo a una promesa (en este caso, el `return` del `fetch`) se bloquea la ejecución de la siguiente instrucción hasta que la promesa se resuelva.

Así, agregando esta sentencia podemos ver que ahora en la variable vemos el objeto `Response`, o sea la promesa resuelta:

```
const resp = await
fetch('https://jsonplaceholder.typicode.com/posts')
console.log(resp) // Response
```

Pero `await` sólo puede utilizarse dentro de una función asincrónica. Aquí es donde entra la **sentencia async**. Esta palabra reservada sirve para declarar una función como asincrónica, y se agrega como prefijo a la función:

```
async function pedirPosts() { }
// o bien
const pedirPosts = async () => { }
```

Así, dentro de una función `async` podemos utilizar la sentencia `await` vista previamente. Esto nos permite esperar a que se resuelvan las promesas vistas para continuar con la instrucción siguiente.

El resultado es una sintaxis que se asemeja a la escritura sincrónica tradicional.

Obtenemos el mismo resultado que antes, pero con una sintaxis más clara. El **async-await** funcionan de la mano.

Recordamos que es una herramienta adicional que puede facilitar la escritura, no es una obligación.

```
const pedirPosts = async () => {
  const resp = await
fetch('https://jsonplaceholder.typicode.com/posts')
  const data = await resp.json()

  data.forEach((post) => {
    const li = document.createElement('li')
    li.innerHTML =
      `

#### ${post.title}



${post.body}


      `

    lista.append(li)
  })
}

pedirPosts()
```