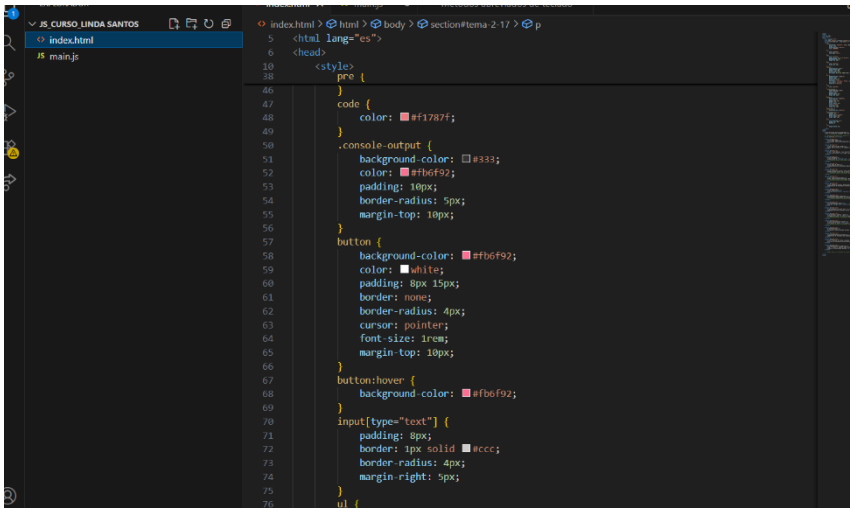
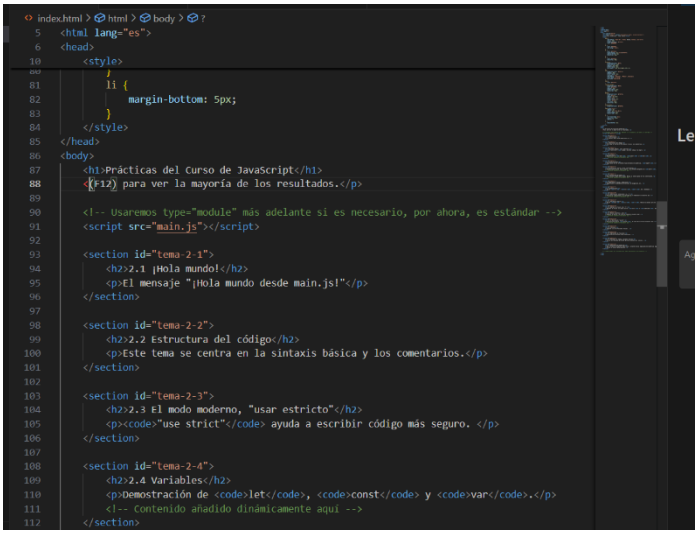
# 2.parte de fundamentos

```'use strict'

// 2.1 hola mundo
// alert('hola mundo')

// 2.2 estrucrura del codigo
// alert('Linda');
// alert('Sofia');
// alert('Santos');
// alert('Galeano');

2.4 variables
const noCambia = 'mi perro es lindo'
console.log(noCambia)
let siCambia = 'mi perro es el mas lindo'
console.log(siCambia);
```



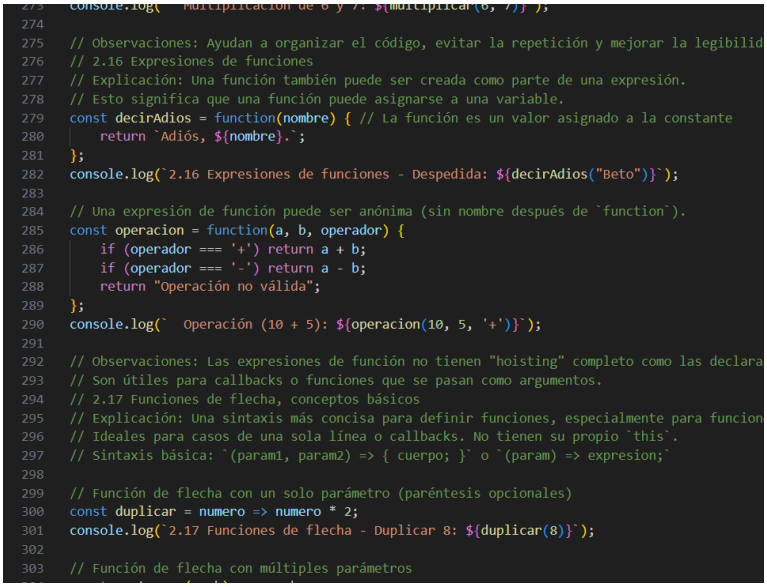


```
// 2.4 Variables
// Explicación: `let` (variable reasignable, ámbito de bloque), `const` (consta
// `var` (antigua, ámbito de función, evitar).
let contadorEventos = 0;
contadorEventos = 5;
const IVA = 0.19;
var mensajeLegado = "Usando var, evitar en código nuevo.";

// Práctica: Mostrar los valores en el DOM.
const varSection = document.getElementById('tema-2-4');
if (varSection) {
    varSection.innerHTML += `<p>Variable <code>contadorEventos</code> (let): ${
    varSection.innerHTML += `<p>Constante <code>IVA</code> (const): ${IVA}</p>`
    varSection.innerHTML += `<p>Variable <code>mensajeLegado</code> (var): ${me
}
// Observaciones: Priorizar `const` y `let` sobre `var`.
```

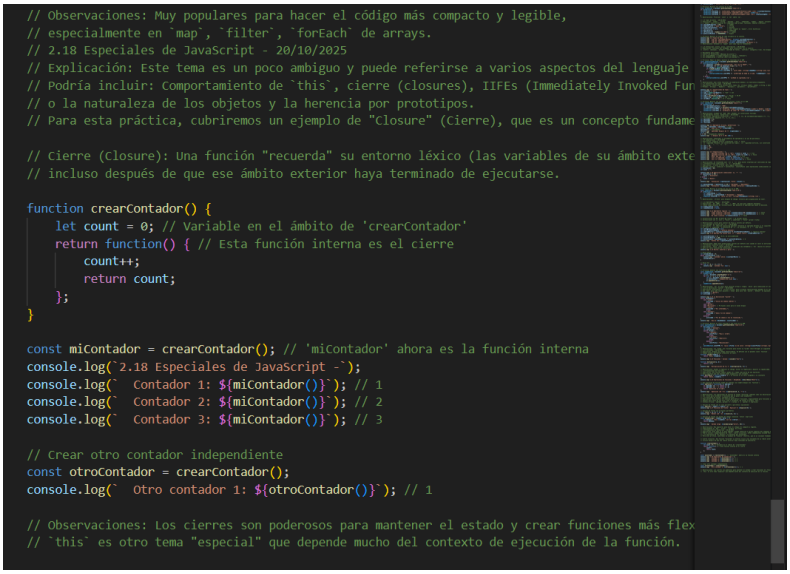
```
// Observaciones: Comprender la precedencia de operadores y el uso de paréntesis
// 2.9 Comparaciones - 20/10/2025
// Explicación: Comparan valores y devuelven `true` o `false`.
// `==` (igualdad flexible, con conversión de tipos), `===` (igualdad estricta,
// `!=`, `!==`, `>`, `<`, `>=`, `<=`.
let valA = 10;
let valB = "10";
let valC = 5;

console.log(`2.9 Comparaciones -`);
console.log(`  Igualdad flexible (10 == "10"): ${valA == valB}`); // true
console.log(`  Igualdad estricta (10 === "10"): ${valA === valB}`); // false
console.log(`  Mayor que (10 > 5): ${valA > valC}`); // true
console.log(`  null == undefined: ${null == undefined}`); // true
console.log(`  null === undefined: ${null === undefined}`); // false

// Observaciones: Se recomienda usar `===` y `!==` para evitar sorpresas por coe
// 2.10 Ramificación condicional: si, '?' - 20/10/2025
// Explicación: `if-else if-else` para lógica condicional.
// Operador ternario `condicion ? valorSiTrue : valorSiFalse` para expresiones c
let puntuacion = 85;
let nivel = "";

console.log(`2.10 Ramificación condicional: si, '?' -`);
if (puntuacion >= 90) {
    nivel = "Excelente";
} else {
    nivel = "Bueno";
```

```
// 2.11 Operadores lógicos - 20/10/2025
// Explicación: `&&` (AND), `||` (OR), `!` (NOT). Se usan para combinar booleanos.
// Tienen comportamiento de "cortocircuito" que devuelve el operando que causó la decisi
let tieneLicencia = true;
let tieneVehiculo = false;

console.log(`2.11 Operadores lógicos -`);
console.log(`  Tiene licencia Y vehículo: ${tieneLicencia && tieneVehiculo}`); // false
console.log(`  Tiene licencia O vehículo: ${tieneLicencia || tieneVehiculo}`); // true
console.log(`  No tiene licencia: ${!tieneLicencia}`); // false

// Cortocircuito con &&: 10 && 0 && "final" -> 0 (primer falsy)
// Cortocircuito con ||: null || "texto" || undefined -> "texto" (primer truthy)

// Observaciones: Útiles para control de flujo y valores por defecto.
// 2.12 Operador de fusión nulo '??' - 20/10/2025
// Explicación: `??` (Nullish coalescing operator) devuelve el operando derecho si el iz
// es `null` o `undefined`. A diferencia de `||`, no considera `0` o `''` como falsy.
let configuracionUsuario = null;
let valorPorDefectoNombre = "Usuario Anónimo";
let nombreAMostrar = configuracionUsuario ?? valorPorDefectoNombre; // "Usuario Anónimo"
console.log(`2.12 Operador de fusión nulo '??' - Nombre (null): ${nombreAMostrar}`);

let puntosObtenidos = 0; // Es 0, no null/undefined
let puntosPorDefecto = 100;
let puntosFinales = puntosObtenidos ?? puntosPorDefecto; // 0
console.log(`  Puntos (0): ${puntosFinales}`);

// Observaciones: Ideal para proporcionar valores por defecto solo cuando el valor es es
// 2.13 Bucles: mientras y para - 20/10/2025
// Explicación: `while` (repite mientras la condición sea verdadera) y `for` (bucle más
// para un número conocido de iteraciones).
console.log(`2.13 Bucles: mientras y para -`);

// Bucle while
let contadorWhile = 0;
while (contadorWhile < 3) {
```

# 2.parte de fundamentos



```
console.log(`  Multiplicación de 6 y 7: ${multiplicar(6, 7)}`);

// Observaciones: Ayudan a organizar el código, evitar la repetición y mejorar la legibilid
// 2.16 Expresiones de funciones
// Explicación: Una función también puede ser creada como parte de una expresión.
// Esto significa que una función puede asignarse a una variable.
const decirAdios = function(nombre) { // La función es un valor asignado a la constante
    return `Adiós, ${nombre}.`;
};
console.log(`2.16 Expresiones de funciones - Despedida: ${decirAdios("Beto")}`);

// Una expresión de función puede ser anónima (sin nombre después de `function`).
const operacion = function(a, b, operador) {
    if (operador === '+') return a + b;
    if (operador === '-') return a - b;
    return "Operación no válida";
};
console.log(`  Operación (10 + 5): ${operacion(10, 5, '+')}`);

// Observaciones: Las expresiones de función no tienen "hoisting" completo como las declara
// Son útiles para callbacks o funciones que se pasan como argumentos.
// 2.17 Funciones de flecha, conceptos básicos
// Explicación: Una sintaxis más concisa para definir funciones, especialmente para funcion
// Ideales para casos de una sola línea o callbacks. No tienen su propio `this`.
// Sintaxis básica: `(param1, param2) => { cuerpo; }` o `(param) => expresion;`

// Función de flecha con un solo parámetro (paréntesis opcionales)
const duplicar = numero => numero * 2;
console.log(`2.17 Funciones de flecha - Duplicar 8: ${duplicar(8)}`);

// Función de flecha con múltiples parámetros
```



```
// Observaciones: Muy populares para hacer el código más compacto y legible,
// especialmente en `map`, `filter`, `forEach` de arrays.
// 2.18 Especiales de JavaScript - 20/10/2025
// Explicación: Este tema es un poco ambiguo y puede referirse a varios aspectos del lenguaje
// Podría incluir: Comportamiento de `this`, cierre (closures), IIFEs (Immediately Invoked Fun
// o la naturaleza de los objetos y la herencia por prototipos.
// Para esta práctica, cubriremos un ejemplo de "Closure" (Cierre), que es un concepto fundame

// Cierre (Closure): Una función "recuerda" su entorno léxico (las variables de su ámbito exte
// incluso después de que ese ámbito exterior haya terminado de ejecutarse.

function crearContador() {
    let count = 0; // Variable en el ámbito de 'crearContador'
    return function() { // Esta función interna es el cierre
        count++;
        return count;
    };
}

const miContador = crearContador(); // 'miContador' ahora es la función interna
console.log(`2.18 Especiales de JavaScript -`);
console.log(`  Contador 1: ${miContador()}`); // 1
console.log(`  Contador 2: ${miContador()}`); // 2
console.log(`  Contador 3: ${miContador()}`); // 3

// Crear otro contador independiente
const otroContador = crearContador();
console.log(`  Otro contador 1: ${otroContador()}`); // 1

// Observaciones: Los cierres son poderosos para mantener el estado y crear funciones más flex
// `this` es otro tema "especial" que depende mucho del contexto de ejecución de la función.
```