Tabla de contenidos

1. Arreglos

¿Qué es un arreglo y cuándo lo necesitamos? Un arreglo es una estructura de datos que nos permite almacenar una lista ordenada de elementos. Cada elemento dentro del arreglo tiene una posición única, llamada **índice**, que comienza en 0. Necesitamos arreglos cuando queremos agrupar datos relacionados, como una lista de nombres de estudiantes, una serie de temperaturas diarias o los productos en un carrito de compras.

Analogía: Un arreglo es como un tren. El tren (el arreglo) contiene varios vagones (los elementos). Para encontrar un vagón específico, no usas su nombre, sino su posición: el primer vagón, el segundo, y así sucesivamente. Esa posición es el índice.

Crear Arreglos: La forma más común de crear un arreglo es usando corchetes `[]`.

```
// Un arreglo de strings
const nombres = ["Ana", "Juan", "Diego"];

// Un arreglo de números
const edades = [25, 30, 28, 45];

// Un arreglo con tipos de datos mixtos (posible, pero menos común)
const datosMixtos = ["Juan", 30, true, { ciudad: "Madrid" }];

// Un arreglo vacío para llenarlo después
const carrito = [];
```

Acceder y Modificar Elementos: Se utiliza la notación de corchetes `[índice]` tanto para leer el valor de un elemento como para modificarlo.

```
const nombres = ["Ana", "Juan", "Diego"];

console.log(nombres[0]); // Lee: "Ana" (el primer elemento)
console.log(nombres[2]); // Lee: "Diego" (el tercer elemento)
console.log(nombres[3]); // Lee: undefined (el índice está fuera de los límites)

// Modificar un elemento existente
nombres[1] = "Juanito";
console.log(nombres); // ["Ana", "Juanito", "Diego"]
```

Propiedad `.length`: Todos los arreglos tienen una propiedad `.length` que devuelve el número de elementos que contienen. Es una propiedad dinámica que se actualiza automáticamente.

```
const frutas = ["Manzana", "Banana", "Cereza"];
console.log(frutas.length); // 3

// La propiedad .length es siempre el último índice + 1.
```

Métodos para Manipular Arreglos:

Los arreglos en JavaScript vienen con un conjunto de métodos incorporados (funciones que pertenecen al objeto arreglo) que facilitan su manipulación. Podemos agruparlos según su función principal.

Métodos para Añadir y Eliminar Elementos:

- `push()`: **Añade** uno o más elementos al **final** del arreglo.
- `pop()`: **Elimina** y devuelve el **último** elemento del arreglo.

```
const colores = ["rojo", "verde"];
colores.push("azul", "morado"); // colores es ahora ["rojo", "verde", "azul", "morado"]
console.log(colores);

const ultimoColor = colores.pop(); // ultimoColor es "morado", colores es ahora ["rojo", "verde", "azul"]
console.log(ultimoColor);
console.log(colores);
```

- `unshift()`: **Añade** uno o más elementos al **inicio** del arreglo.
- `shift()`: **Elimina** y devuelve el **primer** elemento del arreglo.

```
const numeros = [3, 4];
numeros.unshift(1, 2); // numeros es ahora [1, 2, 3, 4]
console.log(numeros);

const primerNumero = numeros.shift(); // primerNumero es 1, numeros es ahora [2, 3, 4]
console.log(primerNumero);
console.log(numeros);
```

Métodos para Búsqueda y Localización:

- `indexOf(elemento)`: Devuelve el **primer índice** en el que se encuentra un elemento. Si no lo encuentra, devuelve `-1`.
- `lastIndexOf(elemento)`: Similar a `indexOf`, pero busca desde el final del arreglo hacia el principio.

```
const herramientas = ["martillo", "destornillador", "sierra", "martillo"];

console.log(herramientas.indexOf("sierra")); // 2
console.log(herramientas.indexOf("martillo")); // 0 (encuentra la primera aparición)
console.log(herramientas.lastIndexOf("martillo")); // 3 (encuentra la última aparición)
console.log(herramientas.indexOf("llave")); // -1 (no se encontró el elemento)

// Caso de uso: verificar si un elemento existe
if (herramientas.indexOf("sierra") !== -1) {
    console.log("¡Tenemos una sierra!");
}
```

Métodos para Reordenar y Extraer Partes:

- `reverse()`: Invierte el orden de los elementos del arreglo **original**. Es un método destructivo (modifica el arreglo).

```
const letras = ['a', 'b', 'c', 'd'];
letras.reverse();
console.log(letras); // ['d', 'c', 'b', 'a']
```

- `sort()`: Ordena los elementos del arreglo **original**. **¡Cuidado!** Por defecto, `sort()` convierte los elementos a strings y los ordena alfabéticamente. Esto puede dar resultados inesperados con números.

```
const nombres = ["Diego", "Ana", "Carlos", "Beatriz"];
nombres.sort();
console.log(nombres); // ["Ana", "Beatriz", "Carlos", "Diego"]

const precios = [100, 5, 25, 0.99];
precios.sort();
console.log(precios); // [0.99, 100, 25, 5] (orden incorrecto, como si fueran strings)
// La ordenación numérica avanzada se verá más adelante en el curso.
```

- `slice(inicio, fin)`: Devuelve una **copia superficial** de una porción del arreglo en un nuevo arreglo. **No modifica el arreglo original**.
  - `inicio`: Índice donde empieza la extracción.
  - `fin` (opcional): Índice *antes* del cual termina la extracción (no se incluye el elemento en `fin`).

```
const animales = ['pato', 'loro', 'gato', 'perro', 'conejo'];

const mascotas = animales.slice(2, 4); // Extrae desde el índice 2 ('gato') hasta el 4 (sin incluirlo).
console.log(mascotas); // ['gato', 'perro']

const aves = animales.slice(0, 2);
console.log(aves); // ['pato', 'loro']

console.log(animales); // El original no ha cambiado: ['pato', 'loro', 'gato', 'perro', 'conejo']
```

- `splice(inicio, cantidadAEliminar, ...itemsAInsertar)`: Un método muy poderoso que **cambia el contenido** de un arreglo eliminando, reemplazando o agregando elementos. **Modifica el arreglo original**.

```
const meses = ['Ene', 'Mar', 'Abr', 'Jun'];

// Insertar 'Feb' en el índice 1
meses.splice(1, 0, 'Feb');
console.log(meses); // ['Ene', 'Feb', 'Mar', 'Abr', 'Jun']

// Reemplazar 'Jun' por 'May' y 'Jun'
meses.splice(4, 1, 'May', 'Jun');
console.log(meses); // ['Ene', 'Feb', 'Mar', 'Abr', 'May', 'Jun']
```

Métodos de Conversión:

- `join(separador)`: Une todos los elementos de un arreglo en un string, separados por el `separador` especificado.

```
const partesDeFecha = ['2024', '08', '29'];

const fechaConGuiones = partesDeFecha.join('-');
console.log(fechaConGuiones); // "2024-08-29"

const fechaConBarras = partesDeFecha.join('/');
console.log(fechaConBarras); // "2024/08/29"
```

.

2. Ciclos Iterativos

¿Para qué sirven los ciclos?

En programación, a menudo nos encontramos con la necesidad de repetir una misma tarea varias veces. Por ejemplo, saludar a cada uno de los 100 usuarios de una lista, procesar cada producto de un carrito de compras o calcular la nota final para cada estudiante de una clase. Escribir el mismo código una y otra vez sería ineficiente y propenso a errores.

Los **ciclos** (o bucles) son estructuras de control que nos permiten ejecutar un bloque de código de forma repetida mientras se cumpla una condición. Son la herramienta fundamental para automatizar tareas repetitivas.

Analogía: Imagina que tienes que dar 10 vueltas a una pista de atletismo. En lugar de pensar "corro un metro, luego otro metro, luego otro...", tu cerebro automatiza la tarea con una instrucción simple: "Mientras no haya completado 10 vueltas, sigo corriendo". El ciclo es esa instrucción: la condición es "no haber completado 10 vueltas" y la acción repetitiva es "seguir corriendo".

Creando un ciclo con `while`

El ciclo `while` (mientras) es una de las formas más fundamentales de crear un bucle. Repite un bloque de código siempre y cuando una condición especificada se evalúe como `true`.

Sintaxis:

```
while (condicion) {
    // Código a ejecutar en cada iteración
    // ¡Importante! Debe haber algo aquí que eventualmente haga la condición false.
}
```

El flujo es el siguiente:

1. Se evalúa la `condicion`.
2. Si es `true`, se ejecuta el bloque de código.
3. Se vuelve al paso 1.
4. Si la `condicion` es `false`, el ciclo termina y el programa continúa con la siguiente línea después del bucle.

¡Peligro! Ciclos Infinitos: Si la condición de un `while` nunca se vuelve `false`, el ciclo se ejecutará para siempre, bloqueando el programa. Siempre debes asegurarte de que haya un "mecanismo de salida".

Ejemplo: Un cohete en cuenta regresiva.

```
let contador = 10;

console.log("Iniciando cuenta regresiva...");

while (contador > 0) {
    console.log(contador);
    contador = contador - 1; // Mecanismo de salida: decrementamos el contador en cada paso.
}

console.log("¡Despegue! 🚀");
```

Asignación con `+=` y Sumatorias

A menudo, dentro de un ciclo, necesitamos acumular un valor. Para esto, los operadores de asignación compuesta son muy útiles. `suma += valor` es una forma abreviada y más legible de escribir `suma = suma + valor`.

Ejemplo: Calcular la sumatoria de los primeros 5 números (1 + 2 + 3 + 4 + 5).

```
let numeroMaximo = 5;
let contador = 1;
let sumatoria = 0; // Variable "acumuladora", debe inicializarse en 0.

while (contador <= numeroMaximo) {
    sumatoria += contador; // En cada paso, añadimos el valor actual del contador a la sumatoria.
    console.log(`Sumando ${contador}. Sumatoria actual: ${sumatoria}`);
    contador++; // Incrementamos el contador para avanzar al siguiente número.
}

console.log(`El resultado de la sumatoria es: ${sumatoria}`); // Resultado: 15
```

Scope de variables en un ciclo

Al igual que con los condicionales `if`, las variables declaradas con `let` o `const` dentro de un ciclo tienen **scope de bloque**. Esto significa que solo existen y son accesibles dentro de las llaves `{...}` del ciclo.

```
let i = 0;
while (i < 3) {
    let mensaje = `Iteración número ${i}`;
    console.log(mensaje); // Funciona, 'mensaje' existe aquí.
    i++;
}

// console.log(mensaje); // Error: ReferenceError: mensaje is not defined
// La variable 'mensaje' fue destruida al terminar el ciclo.
```

Por esta razón, las variables que necesitan acumular un valor (como `sumatoria` en el ejemplo anterior) deben ser declaradas **antes** de que comience el ciclo.

Instrucción `do/while`

El ciclo `do/while` es una variante del `while`. La principal diferencia es que la condición se evalúa **al final** de cada iteración, en lugar de al principio. Esto garantiza que el bloque de código se ejecute **al menos una vez**, sin importar si la condición es verdadera o falsa desde el inicio.

Sintaxis:

```
do {
    // Código a ejecutar
} while (condicion);
```

Caso de uso: Es ideal para situaciones donde necesitas realizar una acción primero y luego comprobar si debes repetirla, como pedir una entrada al usuario hasta que sea válida.

Ejemplo: Pedir una contraseña hasta que sea correcta.

```
let contraseñaCorrecta = "1234";
let inputUsuario;

do {
    // Esta parte se ejecuta al menos una vez.
    inputUsuario = prompt("Por favor, ingrese su contraseña:");
} while (inputUsuario !== contraseñaCorrecta);

alert("¡Contraseña correcta! Acceso concedido.");
```

Instrucción `for`

El ciclo `for` es, a menudo, la opción más conveniente y legible, especialmente cuando sabemos de antemano cuántas veces queremos que se repita el ciclo (por ejemplo, al recorrer un arreglo). Consolida en una sola línea la inicialización, la condición y la actualización del contador.

Sintaxis:

```
for (inicializacion; condicion; actualizacion) {
    // Código a ejecutar en cada iteración
}
```

- **`inicializacion`**: Se ejecuta una sola vez, antes de que el ciclo comience. Usualmente se declara e inicializa una variable contadora (ej. `let i = 0`).
- **`condicion`**: Se evalúa antes de cada iteración. Si es `true`, el bloque se ejecuta. Si es `false`, el ciclo termina.
- **`actualizacion`**: Se ejecuta al final de cada iteración. Usualmente se incrementa o decrementa la variable contadora (ej. `i++`).

Ejemplo: Imprimir los números del 1 al 5.

```
// El ciclo 'for' equivalente al 'while' de la cuenta regresiva es más compacto.
console.log("Iniciando cuenta regresiva con 'for'...");
for (let i = 10; i > 0; i--) {
    console.log(i);
}
console.log("¡Despegue! 🚀");
```

Este ciclo es ideal para recorrer arreglos, ya que la longitud del arreglo nos dice exactamente cuántas iteraciones necesitamos.

```
const frutas = ["Manzana", "Banana", "Cereza", "Damasco"];

for (let i = 0; i < frutas.length; i++) {
    console.log(`En el índice ${i} se encuentra la fruta: ${frutas[i]}`);
}
```

3. Ciclos Anidados

Un ciclo anidado es simplemente un ciclo que se encuentra dentro de otro. El ciclo interno se completará todas sus iteraciones por cada iteración individual del ciclo externo.

Analogía: Piensa en las manecillas de un reloj. Por cada vuelta completa que da la manecilla de las horas (ciclo externo), la manecilla de los minutos (ciclo interno) da 60 vueltas completas.

Son muy útiles para trabajar con estructuras de datos bidimensionales, como las matrices.

Ejemplo: Dibujar un pequeño cuadrado de 3x3 usando asteriscos.

```
let tamaño = 3;
let linea = "";

// Ciclo externo: se encarga de las filas.
for (let fila = 0; fila < tamaño; fila++) {
    linea = ""; // Reseteamos la línea en cada nueva fila.

    // Ciclo interno: se encarga de las columnas de cada fila.
    for (let col = 0; col < tamaño; col++) {
        linea += "* "; // Añadimos un asterisco a la línea actual.
    }

    console.log(linea); // Imprimimos la fila completa.
}

/* Salida en la consola:
* * *
* * *
* * *
 * /
```

4. Iterar sobre los elementos de un arreglo:

Recorrer un arreglo para acceder a cada uno de sus elementos es una operación muy común.

- **Usando un ciclo `for` (tradicional):** Permite un control total sobre el índice, útil si necesitas la posición del elemento.

```
const nombres = ["Ana", "Juan", "Diego", "Laura"];
for (let i = 0; i < nombres.length; i++) {
    console.log(`Índice ${i}: ${nombres[i]}`);
}
```

Ejemplos Prácticos con Arreglos y Ciclos:

- **Ejemplo 1: Calcular la suma de una lista de notas.**

```
const notas = [8, 7, 10, 5, 9];
let sumaTotal = 0;

for (let i = 0; i < notas.length; i++) {
    sumaTotal = sumaTotal + notas[i]; // o sumaTotal += notas[i];
}
const promedio = sumaTotal / notas.length;

console.log(`La suma de las notas es: ${sumaTotal}`);
console.log(`El promedio es: ${promedio}`);
```

- Ejemplo 2: Encontrar el número más alto en un arreglo.

```
const temperaturas = [18, 25, 22, 31, 19, 28];
let tempMasAlta = temperaturas[0]; // Asumimos que la primera es la más alta para empezar a comparar

for (let i = 1; i < temperaturas.length; i++) {
    if (temperaturas[i] > tempMasAlta) {
        tempMasAlta = temperaturas[i];
    }
}
console.log(`La temperatura máxima registrada fue: ${tempMasAlta}°C`);
```

Álgebra con Arreglos (Operaciones de Conjuntos):

- **Concatenación y Unión (Unir arreglos):**

```
* **Método `concat()`:** Crea un nuevo arreglo uniendo los arreglos existentes. Es el método tradicional para esta operación.
const arr1 = [1, 2, 3];
const arr2 = [4, 5, 6];
const arr3 = [7, 8];

// Usando concat()
const concatenado1 = arr1.concat(arr2, arr3);
console.log(concatenado1); // [1, 2, 3, 4, 5, 6, 7, 8]
```

- **Intersección (Elementos comunes):** Para obtener los elementos que existen en *ambos* arreglos, podemos usar ciclos anidados para comparar cada elemento del primer arreglo con todos los elementos del segundo.

```
const grupoA = ['Juan', 'Ana', 'Luis', 'Maria'];
const grupoB = ['Pedro', 'Maria', 'Ana', 'Sofia'];
const interseccion = [];

for (let i = 0; i < grupoA.length; i++) {
  const personaA = grupoA[i];
  for (let j = 0; j < grupoB.length; j++) {
      if (personaA === grupoB[j]) {
          interseccion.push(personaA);
          break; // Rompemos el ciclo interno para evitar duplicados si un nombre se repite en grupoB
      }
  }
}
console.log(interseccion); // ['Ana', 'Maria']
```

- **Diferencia (Elementos únicos en un arreglo):** Para obtener los elementos que están en el primer arreglo pero *no* en el segundo, recorremos el primer arreglo y, por cada elemento, verificamos si existe en el segundo.

```
const todosLosAlumnos = ['Juan', 'Ana', 'Luis', 'Maria', 'Pedro'];
const alumnosPresentes = ['Ana', 'Maria', 'Pedro'];
const alumnosAusentes = [];

for (let i = 0; i < todosLosAlumnos.length; i++) {
    const alumno = todosLosAlumnos[i];
    let estaPresente = false;

    for (let j = 0; j < alumnosPresentes.length; j++) {
        if (alumno === alumnosPresentes[j]) {
            estaPresente = true;
            break;
        }
    }

    if (!estaPresente) {
        alumnosAusentes.push(alumno);
    }
}
console.log(alumnosAusentes); // ['Juan', 'Luis']
```

Matrices y Arreglos Asociativos:

- **Matrices (Arreglos anidados):** Un arreglo puede contener otros arreglos. Esto es útil para representar estructuras de datos bidimensionales, como un tablero de ajedrez, un mapa o una hoja de cálculo.

```
const tablero = [
  ['X', 'O', 'X'],
  ['O', 'X', 'O'],
  ['X', 'O', 'O']
];

// Para acceder a un elemento, se usan dos índices: [fila][columna]
console.log(tablero[1][2]); // 'O' (fila con índice 1, columna con índice 2)

// Modificar un valor
tablero[2][2] = 'X';
console.log(tablero);
```

- **Arreglos Asociativos:** En otros lenguajes, existen arreglos donde los índices son strings en lugar de números (claves). En JavaScript, esta funcionalidad se logra de manera nativa y más eficiente con **Objetos**, no con Arreglos. Si necesitas asociar una `clave` (string) a un `valor`, debes usar un objeto.

```
// INCORRECTO en JavaScript (aunque funcione, es un anti-patrón)
let arrAsociativo = [];
arrAsociativo["nombre"] = "Juan";
console.log(arrAsociativo.length); // 0 (¡el arreglo sigue vacío!)

// CORRECTO en JavaScript
const objAsociativo = {
    nombre: "Juan",
    edad: 30
};
console.log(objAsociativo.nombre); // "Juan"
```

5. Combinación de Ciclos con Instrucciones if/else

La verdadera potencia de los ciclos se desata cuando los combinamos con sentencias condicionales. Esto nos permite no solo repetir una acción, sino también tomar decisiones diferentes en cada una de esas repeticiones. Podemos evaluar cada elemento que estamos iterando y actuar en consecuencia.

Caso de uso: Filtrar datos, buscar elementos específicos, modificar valores que cumplan cierta condición, etc.

Ejemplo 1: Separar números pares e impares de una lista.

```
const numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
const pares = [];
const impares = [];

for (let i = 0; i < numeros.length; i++) {
    const numeroActual = numeros[i];
    // Usamos el operador módulo (%) para saber el resto de la división por 2.
    // Si el resto es 0, el número es par.
    if (numeroActual % 2 === 0) {
        pares.push(numeroActual);
    } else {
        impares.push(numeroActual);
    }
}

console.log("Números originales:", numeros);
console.log("Números pares:", pares); // [2, 4, 6, 8, 10]
console.log("Números impares:", impares); // [1, 3, 5, 7, 9]
```

Ejemplo 2: Encontrar el primer usuario mayor de 30 años y detener la búsqueda. A veces, no necesitamos recorrer todo el arreglo. Podemos usar la instrucción `break` para salir de un ciclo prematuramente una vez que hemos encontrado lo que buscábamos.

```
const usuarios = [
    { nombre: "Ana", edad: 25 },
    { nombre: "Juan", edad: 28 },
    { nombre: "Carlos", edad: 32 },
    { nombre: "Maria", edad: 29 }
];

let usuarioEncontrado = null; // null indica que aún no hemos encontrado nada

for (let i = 0; i < usuarios.length; i++) {
    const usuario = usuarios[i];
    console.log(`Evaluando a ${usuario.nombre}...`);

    if (usuario.edad > 30) {
        usuarioEncontrado = usuario;
        console.log("¡Usuario encontrado!");
        break; // Salimos del ciclo para no seguir buscando innecesariamente.
    }
}

if (usuarioEncontrado !== null) {
    console.log("El primer usuario mayor de 30 es:", usuarioEncontrado.nombre);
} else {
    console.log("No se encontraron usuarios mayores de 30.");
}
```

6. Estilos, Convenciones y Buenas Prácticas de Codificación

Escribir código que funcione es solo la mitad del trabajo. Un programador profesional también debe escribir código que sea legible, mantenible y comprensible para otros desarrolladores (y para su "yo" del futuro).

El Concepto de Código Limpio (Clean Code)

El "Código Limpio" es una filosofía popularizada por Robert C. Martin que aboga por escribir código de la manera más clara y simple posible. Un código limpio es:

- **Legible:** Se lee casi como prosa bien escrita. Otros pueden entender la intención del código fácilmente.
- **Simple:** Se enfoca en resolver el problema de la manera más directa posible, sin complejidad innecesaria.
- **Mantenible:** Es fácil de depurar, modificar y extender sin introducir nuevos errores.

¿Por qué seguir una Guía de Estilo y Convenciones?

Imagina un libro donde cada página usa un tipo de letra, tamaño y márgenes diferentes. Sería muy difícil y agotador de leer. Lo mismo sucede con el código. Las guías de estilo y las convenciones son un conjunto de reglas acordadas que aportan consistencia al código de un proyecto.

- **Mejora la legibilidad:** Un estilo consistente permite a los desarrolladores centrarse en la lógica del programa en lugar de descifrar la sintaxis.
- **Facilita la colaboración:** Cuando todo el equipo sigue las mismas reglas, es más fácil para cualquiera leer, revisar y contribuir al código de otros.
- **Reduce errores:** Muchas convenciones ayudan a prevenir errores comunes. Por ejemplo, usar `===` en lugar de `==`.
- **Acelera el desarrollo:** Un código legible es más rápido de entender, lo que acelera el proceso de añadir nuevas funcionalidades o corregir errores.

Convenciones y Buenas Prácticas Fundamentales

Aquí hay algunas de las convenciones más importantes para empezar a escribir código JavaScript de alta calidad:

1. **Nombres de Variables y Funciones:**
    - **Usa `camelCase`:** `let nombreDeUsuario;`, `function calcularTotal() {}`.
    - **Sé descriptivo:** Usa `let usuariosActivos = 10;` en lugar de `let u = 10;`. El nombre debe revelar la intención.
    - **Evita abreviaturas confusas:** `nombreUsuario` es mejor que `nomUsr`.
2. **Indentación y Espaciado:**
    - **Indentación:** Usa 2 o 4 espacios para indentar bloques de código (dentro de `if`, `for`, funciones, etc.). Sé consistente. La mayoría de los editores de código lo hacen automáticamente.
    - **Espacios alrededor de operadores:** `let suma = a + b;` es más legible que `let suma=a+b;`.
3. **Comentarios:**
    - **Comenta el "porqué", no el "qué":** El código debe explicar por sí mismo *qué* está haciendo. Usa comentarios para explicar *por qué* tomaste una decisión de implementación compleja o inusual.
    - **Evita comentarios obvios:**

```
// Mal:
// Incrementa i en 1
i++;

// Bueno:
// Usamos el algoritmo de Fisher-Yates para barajar el arreglo
// porque ofrece una distribución uniforme y es eficiente.
...código complejo...
```