Contenido

CONSTANTES

```
// Declarar una constante llamada PI

const PI = 3.14159;

// Intentar reasignar un valor a la constante PI

// Esto generará un error ya que las constantes no pueden ser reasignadas

PI = 3.14; // Esto generará un error

// Declarar una constante de tipo objeto

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

// Modificar una propiedad del objeto

persona.edad = 31;

// Intentar reasignar el objeto a otra cosa

// Esto generará un error, pero las propiedades del objeto todavía pueden ser modificadas

persona = { nombre: "María", edad: 25 }; // Esto generará un error
```

En JavaScript, aunque no puedes declarar una constante con un tipo específico como en otros lenguajes de programación estáticamente tipados, puedes lograr un comportamiento similar utilizando TypeScript, que es un superconjunto de JavaScript que agrega tipado estático opcional al lenguaje.

En TypeScript, puedes declarar una constante con un tipo específico utilizando la sintaxis de TypeScript.

```
// Declarar una constante con un tipo específico en TypeScript

const miNumero: number = 42;

// Intentar reasignar un valor a la constante generará un error

// miNumero = 3.14; // Esto generará un error, ya que miNumero es una constante y no se puede reasignar

console.log(miNumero); // Imprimirá: 42
```

Para utilizar TypeScript, necesitarás un compilador TypeScript que traduzca tu código TypeScript a JavaScript estándar antes de ejecutarlo en un entorno de ejecución de JavaScript.

TypeScript es un lenguaje de programación desarrollado por Microsoft que es un superconjunto tipado de JavaScript. Esto significa que TypeScript extiende y añade funcionalidades a JavaScript al mismo tiempo que proporciona la capacidad de agregar tipado estático opcional al código.

VARIABLES

Las variables en JavaScript son contenedores para almacenar datos. Pueden contener cualquier tipo de datos, como números, cadenas, objetos, funciones, etc. En JavaScript, hay tres palabras clave para declarar variables: **var**, **let** y **const**.

1. **var**: Era la forma tradicional de declarar variables en JavaScript antes de la introducción de **let** y **const**. Las variables declaradas con **var** tienen un ámbito de función, lo que significa que están disponibles en toda la función en la que se declaran.

```
var x = 10;

console.log(x); // Imprime 10
```

2. **let**: Introducido en ECMAScript 6 (ES6), **let** permite declarar variables con un ámbito de bloque, lo que significa que están disponibles solo dentro del bloque en el que se declaran.

```
let y = 20;

console.log(y); // Imprime 20
```

3. **const**: **const** también fue introducido en ES6 y se utiliza para declarar constantes, es decir, variables cuyo valor no puede ser reasignado una vez que se les ha asignado un valor inicial. Las constantes deben inicializarse en el momento de la declaración y no pueden ser redeclaradas.

```
const z = 30;

console.log(z); // Imprime 30
```

TIPOS DE DATOS MÁS COMUNES

En JavaScript, existen varios tipos de datos que se utilizan para almacenar diferentes tipos de valores. Aquí tienes una descripción de los tipos de datos más comunes:

1. **Number (Número)**: Se utiliza para representar números, ya sean enteros o de punto flotante. Ejemplos: `10`, `3.14`, `-7`.

```
// Definir números enteros y de punto flotante

let entero = 10;

let flotante = 3.14;

// Realizar operaciones aritméticas

let suma = entero + flotante; // Suma

let resta = entero - flotante; // Resta

let multiplicacion = entero * flotante; // Multiplicación

let division = entero / flotante; // División

// Imprimir los resultados

console.log("Suma:", suma); // Imprimirá: Suma: 13.14

console.log("Resta:", resta); // Imprimirá: Resta: 6.86

console.log("Multiplicación:", multiplicacion); // Imprimirá: Multiplicación: 31.4

console.log("División:", division); // Imprimirá: División: 3.1847133757961785
```

2. **String (Cadena):** Se utiliza para representar texto. Las cadenas se pueden definir utilizando comillas simples (`'`) o dobles (`"`). Ejemplos: `'Hola'`, `"Mundo"`.

```
// Definir cadenas de texto

let saludo = '¡Hola, mundo!';

let nombre = "Juan";

// Concatenar cadenas

let mensaje = saludo + ' Mi nombre es ' + nombre;

// Imprimir el mensaje

console.log(mensaje); // Imprimirá: ¡Hola, mundo! Mi nombre es Juan
```

3. **Boolean (Booleano)**: Representa un valor de verdadero o falso. Los valores son `true` y `false`.

```
// Definir variables booleanas

let esVerdadero = true;

let esFalso = false;

// Realizar operaciones lógicas

let resultadoY = esVerdadero && esFalso; // AND lógico

let resultadoO = esVerdadero || esFalso; // OR lógico

let resultadoNegacion = !esVerdadero; // Negación lógica

// Imprimir los resultados

console.log("Resultado AND:", resultadoY); // Imprimirá: Resultado AND: false

console.log("Resultado OR:", resultadoO); // Imprimirá: Resultado OR: true

console.log("Negación:", resultadoNegacion); // Imprimirá: Negación: false
```

4. **Null:** Representa la ausencia intencional de cualquier valor o referencia a un objeto. En JavaScript, `null` es un valor primitivo.

```
// Definir una variable inicializada con null

let miVariable = null;

// Realizar una verificación de null

if (miVariable === null) {

   console.log("La variable está inicializada como null.");

} else {

   console.log("La variable no está inicializada como null.");

}
```

5. **Undefined:** Representa el valor de una variable que no ha sido asignada. Si una variable se declara pero no se le asigna ningún valor, su valor predeterminado será `undefined`.

```
// Definir una variable sin asignarle un valor

let variableIndefinida;

// Imprimir el valor de la variable
```

```
console.log("Valor de la variable:", variableIndefinida); // Imprimirá: Valor de la variable: undefined

// Verificar si la variable está definida

if (typeof variableIndefinida === 'undefined') {

   console.log("La variable está indefinida.");

} else {

   console.log("La variable está definida.");

}
```

6. **Object (Objeto):** Se utiliza para almacenar colecciones de datos clave-valor. Los objetos en JavaScript son contenedores de propiedades, donde cada propiedad tiene un nombre único y un valor asociado. Ejemplo:

```
  // Crear un objeto representando una persona

let persona = {

   nombre: "Juan",

   edad: 30,

   ciudad: "Madrid",

   saludar: function() {

      console.log("¡Hola! Mi nombre es " + this.nombre + ", tengo " + this.edad + " años y vivo en " + this.ciudad + ".");

   }

};

// Acceder a las propiedades del objeto

console.log("Nombre:", persona.nombre); // Imprimirá: Nombre: Juan

console.log("Edad:", persona.edad); // Imprimirá: Edad: 30

console.log("Ciudad:", persona.ciudad); // Imprimirá: Ciudad: Madrid

// Llamar al método del objeto

persona.saludar(); // Imprimirá: ¡Hola! Mi nombre es Juan, tengo 30 años y vivo en Madrid.
```

7. **Array (Arreglo):** Se utiliza para almacenar una colección ordenada de valores. Los elementos de un array se acceden mediante un índice numérico, y el primer elemento tiene el índice `0`. Ejemplo:

```
  var colores = ["rojo", "verde", "azul"];

// Crear un array de números

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

// Acceder a los elementos del array

console.log("Primer elemento:", numeros[0]); // Imprimirá: Primer elemento: 1

console.log("Segundo elemento:", numeros[1]); // Imprimirá: Segundo elemento: 2

// Iterar sobre los elementos del array

console.log("Elementos del array:");

numeros.forEach(function(numero) {

   console.log(numero);

});

// Añadir un elemento al final del array

numeros.push(6);

console.log("Array después de añadir un elemento:", numeros); // Imprimirá: Array después de añadir un elemento: [1, 2, 3, 4, 5, 6]

// Eliminar el último elemento del array

let ultimoElemento = numeros.pop();

console.log("Último elemento eliminado:", ultimoElemento); // Imprimirá: Último elemento eliminado: 6

console.log("Array después de eliminar el último elemento:", numeros); // Imprimirá: Array después de eliminar el último elemento: [1, 2, 3, 4, 5]
```

8. **Symbol (Símbolo):** Introducido en ECMAScript 6, representa un valor único e inmutable que se puede utilizar como clave de una propiedad de un objeto. Los símbolos se crean utilizando la función `Symbol()`.

```
// Crear un símbolo con descripción

const miSimbolo = Symbol('Descripción de mi símbolo');

// Crear un objeto con una propiedad de tipo símbolo

const persona = {

  nombre: 'Juan',

  edad: 30,

  [miSimbolo]: 'Valor secreto'
```

```
};

// Acceder a la propiedad de tipo símbolo utilizando la notación de corchetes

console.log(persona[miSimbolo]); // Imprimirá: Valor secreto
```

Estos son los tipos de datos principales en JavaScript. Es importante tener en cuenta que JavaScript es un lenguaje de tipado dinámico, lo que significa que una variable puede contener diferentes tipos de datos en momentos diferentes durante la ejecución del programa.

TIPOS DE DATOS MÁS ESPECIALIZADOS

En JavaScript, además de los tipos de datos que mencioné anteriormente, hay algunos otros tipos que son menos comunes o especializados. Aquí te los presento:

1. **BigInt:** Introducido en ECMAScript 2020, permite representar números enteros arbitrariamente grandes. Se crea agregando `n` al final de un número entero literal o utilizando el constructor `BigInt()`. Ejemplo:

```
const valorGrande = 1234567890123456789012345678901234567890n;
```

2. **Function (Función):** En JavaScript, las funciones son de hecho objetos, pero se pueden considerar un tipo de datos separado. Las funciones se utilizan para encapsular un bloque de código que se puede ejecutar cuando se invoca. Ejemplo:

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

3. **Date (Fecha):** Se utiliza para trabajar con fechas y horas en JavaScript. Se crea utilizando el constructor `Date()`. Ejemplo:

```
const fechaActual = new Date();
```

Crear un objeto Date con la fecha y hora actuales:

```
const fechaActual = new Date();
console.log("Fecha actual:", fechaActual);
```

Crear un objeto Date con una fecha específica:

```
const miCumpleaños = new Date(1990, 6, 15); // 15 de julio de 1990
console.log("Mi cumpleaños:", miCumpleaños);
```

Obtener componentes individuales de una fecha:

```
const fecha = new Date();
const año = fecha.getFullYear();
const mes = fecha.getMonth(); // Los meses comienzan desde 0 (enero) hasta 11 (diciembre)
const dia = fecha.getDate();
const hora = fecha.getHours();
const minutos = fecha.getMinutes();
```

```
const segundos = fecha.getSeconds();

console.log(`Fecha actual: ${dia}/${mes + 1}/${año}`);

console.log(`Hora actual: ${hora}:${minutos}:${segundos}`);
```

Formatear una fecha como una cadena de texto:

```
const fecha = new Date();

const opciones = { weekday: 'long', year: 'numeric', month: 'long', day: 'numeric' };

const fechaFormateada = fecha.toLocaleDateString('es-ES', opciones);

console.log("Fecha formateada:", fechaFormateada);
```

Operaciones con fechas:

```
const fechaActual = new Date();

const fechaFutura = new Date(fechaActual);

fechaFutura.setDate(fechaActual.getDate() + 7); // Sumar 7 días

console.log("Fecha dentro de una semana:", fechaFutura);

const milisegundosEnUnDia = 24 * 60 * 60 * 1000;

const fechaPasada = new Date(fechaActual.getTime() - milisegundosEnUnDia); // Restar 1 día

console.log("Fecha de ayer:", fechaPasada);
```

4. **RegExp (Expresión Regular):** Se utiliza para trabajar con patrones de búsqueda y manipulación de texto. Las expresiones regulares se crean utilizando el constructor `RegExp()` o mediante una notación literal. Ejemplo:

```
const expresionRegular = /[a-z]+/;

// Definir una expresión regular para buscar números de teléfono en formato (XXX) XXX-XXXX

const expresionRegular = /\(\d{3}\) \d{3}-\d{4}/;

// Definir una cadena de texto para buscar el número de teléfono

const texto = "Mi número de teléfono es (123) 456-7890. Llámame si tienes alguna pregunta.";

// Buscar coincidencias utilizando la expresión regular

const coincidencias = texto.match(expresionRegular);

// Imprimir las coincidencias encontradas
```

console.log("Coincidencias encontradas:", coincidencias);

Notación Literal:

\(y \) coinciden con los paréntesis literalmente.

\d coincide con un dígito numérico.

{3} indica que el patrón anterior (en este caso, \d) debe repetirse exactamente tres veces.

El espacio y el guion (y -) coinciden con esos caracteres literalmente.

La barra diagonal (/) que se utiliza al principio y al final de la expresión regular es simplemente parte de la sintaxis de la notación literal de expresiones regulares en JavaScript. Sirve para delimitar el inicio y el final de la expresión regular.

5. **Map:** Se utiliza para almacenar pares clave-valor donde tanto las claves como los valores pueden ser de cualquier tipo. Ejemplo:

Map es una estructura de datos en JavaScript que representa un mapa de claves y valores.

La estructura Map en JavaScript es similar a un objeto, pero a diferencia de los objetos, las claves pueden ser de cualquier tipo de datos, no solo cadenas de texto o símbolos. Además, en un Map, el orden de los elementos está garantizado, lo que significa que los elementos se almacenan en el orden en que se insertaron.

```
  // Crear un nuevo Map

const miMapa = new Map();

// Agregar elementos al Map

miMapa.set('clave1', 'valor1');

miMapa.set('clave2', 'valor2');

// Obtener el valor de una clave

console.log(miMapa.get('clave1')); // Imprimirá: valor1

// Verificar si una clave existe en el Map

console.log(miMapa.has('clave2')); // Imprimirá: true

// Iterar sobre los elementos del Map

miMapa.forEach(function(valor, clave) {

   console.log(`Clave: ${clave}, Valor: ${valor}`);

});

// Eliminar un elemento del Map
```

```
miMapa.delete('clave1');
```

6. **Set:** Se utiliza para almacenar valores únicos de cualquier tipo, tanto valores primitivos como objetos. La función new Set() se utiliza para crear un nuevo objeto Set, que es una estructura de datos que te permite almacenar valores únicos de cualquier tipo, ya sean valores primitivos o referencias a objetos. En un Set, no se permiten duplicados, lo que significa que cada elemento puede aparecer solo una vez en la colección.

```
// Crear un nuevo Set

const miSet = new Set();

// Agregar elementos al Set

miSet.add(1);

miSet.add(2);

miSet.add(3);

miSet.add(1); // Duplicado, será ignorado

// Verificar si un elemento está en el Set

console.log(miSet.has(2)); // Imprimirá: true

console.log(miSet.has(4)); // Imprimirá: false

// Obtener el tamaño del Set

console.log("Tamaño del Set:", miSet.size); // Imprimirá: 3

// Iterar sobre los elementos del Set

miSet.forEach(function(valor) {

   console.log("Elemento:", valor);

});

// Eliminar un elemento del Set

miSet.delete(2);

// Convertir el Set a un array

const arrayDesdeSet = Array.from(miSet);

console.log("Array desde Set:", arrayDesdeSet); // Imprimirá: [1, 3]
```

Estos son algunos de los tipos de datos menos comunes o más especializados en JavaScript. Cada uno tiene su propia utilidad y se utiliza en situaciones específicas.

7. **Typed Arrays (Arrays Tipados):** Introducidos en ECMAScript 2015, los Typed Arrays son arrays que tienen un tipo de dato específico para sus elementos, como enteros de 8 bits sin signo, enteros de 16 bits con signo, flotantes de 32 bits, etc. Estos arrays proporcionan una forma eficiente de trabajar con datos binarios en el navegador o en entornos de servidor como Node.js. Ejemplo:

```
  const arrayEnteros = new Int32Array(3); // Array de 3 enteros de 32 bits con signo

  arrayEnteros[0] = 1;

  arrayEnteros[1] = 2;

  arrayEnteros[2] = 3;

// Crear un "Typed Array" de 8 bits sin signo con una longitud de 4 elementos

const typedArray = new Uint8Array(4);

// Asignar valores a los elementos del "Typed Array"

typedArray[0] = 10;

typedArray[1] = 20;

typedArray[2] = 30;

typedArray[3] = 40;

// Acceder a los elementos del "Typed Array" e imprimirlos

console.log("Elemento 1:", typedArray[0]); // Imprimirá: 10

console.log("Elemento 2:", typedArray[1]); // Imprimirá: 20

console.log("Elemento 3:", typedArray[2]); // Imprimirá: 30

console.log("Elemento 4:", typedArray[3]); // Imprimirá: 40

// Iterar sobre los elementos del "Typed Array"

console.log("Todos los elementos:");

for (let i = 0; i < typedArray.length; i++) {

   console.log("Elemento", i + 1, ":", typedArray[i]);

}
```

Array tipado con caracteres

```
// Crear un "Typed Array" de 8 bits sin signo con una longitud de 5 elementos
const caracteresArray = new Uint8Array(5);
// Asignar valores numéricos que representan caracteres ASCII
caracteresArray[0] = 72; // 'H'
caracteresArray[1] = 101; // 'e'
caracteresArray[2] = 108; // 'l'
caracteresArray[3] = 108; // 'l'
caracteresArray[4] = 111; // 'o'
// Iterar sobre los elementos del "Typed Array" y convertirlos a caracteres
console.log("Elementos como caracteres:");
for (let i = 0; i < caracteresArray.length; i++) {
   const caracter = String.fromCharCode(caracteresArray[i]);
   console.log("Elemento", i + 1, ":", caracter);
}
```

Los Typed Arrays son especialmente útiles cuando se trabaja con operaciones que involucran manipulación de datos binarios, como procesamiento de imágenes, audio, o cuando se necesita interactuar con APIs que operan en este nivel de detalle.

Con estos siete tipos de datos, hemos cubierto prácticamente todos los tipos de datos principales en JavaScript, cada uno con sus propias características y usos específicos.

OPERADORES

En JavaScript, los operadores son símbolos especiales que se utilizan para realizar operaciones en operandos. Los operadores en JavaScript se dividen en varios tipos, que incluyen:

1. **Operadores aritméticos:** Se utilizan para realizar operaciones matemáticas en operandos numéricos.

```
// Ejemplos de operadores aritméticos

let suma = 5 + 3; // suma = 8

let resta = 10 - 4; // resta = 6

let multiplicacion = 2 * 6; // multiplicacion = 12

let division = 20 / 4; // division = 5

let modulo = 11 % 3; // modulo = 2 (resto de la división)
```

Los operadores de incremento (++) y decremento (--) se utilizan para aumentar o disminuir el valor de una variable en 1, respectivamente. Estos operadores pueden aplicarse de dos formas diferentes: prefijo y sufijo.

```
let x = 5;

let resultado = ++x; // Incrementa x en 1 y luego asigna el nuevo valor a resultado

console.log(resultado); // Imprime 6
```

Más ejemplos:

```
let y = 10;

console.log(--y); // Imprime 9 (decrementa y en 1 y luego devuelve el nuevo valor)

let a = 3;

let b = a++; // Asigna el valor actual de a a b y luego incrementa a en 1

console.log(b); // Imprime 3

console.log(a); // Imprime 4

let c = 8;

console.log(c--); // Imprime 8 (devuelve el valor actual de c y luego decrementa c en 1)

console.log(c); // Imprime 7
```

2. **Operadores de asignación:** Se utilizan para asignar valores a variables.

```
// Ejemplos de operadores de asignación

let x = 10; // Asignación simple
```

```
  x += 5; // Asignación con adición (equivalente a x = x + 5)

let x = 10;

let y = 5;

// Operador de asignación simple (=)

let z = x + y; // z = 15

// Operadores de asignación compuesta

z += 5; // z = z + 5 (z = 15 + 5)

console.log("Después de z += 5:", z); // Imprime 20

z -= 3; // z = z - 3 (z = 20 - 3)

console.log("Después de z -= 3:", z); // Imprime 17

z *= 2; // z = z * 2 (z = 17 * 2)

console.log("Después de z *= 2:", z); // Imprime 34

z /= 4; // z = z / 4 (z = 34 / 4)

console.log("Después de z /= 4:", z); // Imprime 8.5

z %= 3; // z = z % 3 (z = 8.5 % 3)

console.log("Después de z %= 3:", z); // Imprime 2.5

// Operadores de asignación de incremento/decremento

let a = 5;

a++; // Incremento en 1 (a = a + 1)

console.log("Después de a++:", a); // Imprime 6

let b = 8;

b--; // Decremento en 1 (b = b - 1)

console.log("Después de b--:", b); // Imprime 7
```

3. **Operadores de comparación:** Se utilizan para comparar valores y devolver un valor booleano (verdadero o falso).

```
  // Ejemplos de operadores de comparación

  let a = 5;

  let b = 10;
```

```
  console.log(a === b); // Igualdad estricta (false)
  console.log(a !== b); // Desigualdad estricta (true)
```

Igualdad

```
console.log(5 == 5); // true
console.log("5" == 5); // true (se realiza una conversión de tipo)
console.log(5 == "5"); // true (se realiza una conversión de tipo)
console.log(5 == 6); // false
```

Igualdad estricta

```
console.log(5 === 5); // true
console.log("5" === 5); // false
console.log(5 === "5"); // false
console.log(5 === 6); // false
```

Desigualdad

```
console.log(5 != 6); // true
console.log(5 != "6"); // false (se realiza una conversión de tipo)
```

Desigualdad estricta

```
console.log(5 !== 6); // true
console.log(5 !== "6"); // true
```

Mayor que, menor que

```
console.log(5 > 3); // true
console.log(5 < 3); // false
```

Mayor o igual que, menor o igual que

```
console.log(5 >= 3); // true
console.log(5 <= 3); // false
```

4. **Operadores lógicos:** Se utilizan para realizar operaciones lógicas en valores booleanos y devolver un valor booleano.

```
// Ejemplos de operadores lógicos
let c = true;
let d = false;
console.log(c && d); // AND lógico (false)
console.log(c || d); // OR lógico (true)
console.log(!c); // NOT lógico (false)
```

Otro ejemplo:

```
let x = 5;
let y = 10;
let z = 15;
// Operador lógico AND (&&)
if (x < y && y < z) {
   console.log("x es menor que y y y es menor que z");
} else {
   console.log("Al menos una de las condiciones no se cumple");
}
// Operador lógico OR (||)
if (x < y || y > z) {
   console.log("x es menor que y o y es mayor que z");
} else {
   console.log("Ninguna de las condiciones se cumple");
}
// Operador lógico NOT (!)
let esMayor = !(x < y);
console.log("¿x es mayor que y?", esMayor);
```

No hay XOR, hay que simularlo:

```
// XOR utilizando operadores lógicos
function xor(a, b) {
   return (a || b) && !(a && b);
}
// Ejemplos de uso
console.log(xor(true, true));   // false
console.log(xor(true, false));  // true
console.log(xor(false, true));  // true
console.log(xor(false, false)); // false
```

5. **Operadores de concatenación:** Se utilizan para concatenar cadenas de texto.

```
  // Ejemplo de operador de concatenación
  let nombre = "Juan";
  let apellido = "Pérez";
  let nombreCompleto = nombre + " " + apellido; // nombreCompleto = "Juan Pérez"
```

```
let nombre = "Juan";
let apellido = "Pérez";
let edad = 30;
let ciudad = "Madrid";

// Concatenación de cadenas de texto
let nombreCompleto = nombre + " " + apellido;
console.log("Nombre completo:", nombreCompleto);
// Concatenación de cadena y número
let mensajeEdad = "Tengo " + edad + " años";
```

```
console.log("Mensaje de edad:", mensajeEdad);

// Concatenación de múltiples cadenas de texto

let direccion = "Vivo en " + ciudad + ", España";

console.log("Dirección:", direccion);
```

6. **Operadores de tipo:** Se utilizan para determinar el tipo de datos de un valor.

```
  // Ejemplo de operador de tipo

  let tipoDeDato = typeof 5; // tipoDeDato = "number"
```

Otro ejemplo:

```
let nombre = "Juan";

let edad = 30;

let esCasado = false;

let lista = [1, 2, 3];

let persona = { nombre: "María", edad: 25 };

console.log(typeof nombre); // Imprime "string"

console.log(typeof edad); // Imprime "number"

console.log(typeof esCasado); // Imprime "boolean"

console.log(typeof lista); // Imprime "object" (array es un tipo de objeto en JavaScript)

console.log(typeof persona); // Imprime "object"
```

Estos son solo algunos ejemplos de los tipos de operadores que se utilizan comúnmente en JavaScript. Los operadores son fundamentales en la programación y se utilizan para realizar una variedad de tareas, desde cálculos matemáticos hasta comparaciones y manipulación de datos.

CONDICIONALES

Los condicionales en JavaScript son estructuras de control que te permiten ejecutar cierto bloque de código solo si se cumple una condición específica. Los condicionales más comunes en JavaScript son el `if`, `else if` y `else`. Los condicionales son fundamentales en la programación, ya que permiten que el flujo del programa se adapte a diferentes situaciones según las condiciones dadas.

1. **if:** Se utiliza para ejecutar un bloque de código si la condición dada es verdadera.

```
let edad = 18;

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

2. **else if:** Se utiliza para especificar una nueva condición si la primera condición es falsa.

```
let hora = 14;

if (hora < 12) {

   console.log("Buenos días");

} else if (hora < 18) {

   console.log("Buenas tardes");

} else {

   console.log("Buenas noches");

}
```

3. **else:** Se utiliza para ejecutar un bloque de código si la condición del `if` es falsa.

```
let hora = 20;

if (hora < 12) {

   console.log("Buenos días");

} else {

   console.log("Buenas noches");

}
```

Además de estos condicionales básicos, JavaScript también tiene el **operador ternario** (`condición ? expresión1 : expresión2`), que es una forma abreviada de escribir condicionales simples en una sola línea.

```
let edad = 20;

let mensaje = (edad >= 18) ? "Eres mayor de edad" : "Eres menor de edad";

console.log(mensaje);
```

BUCLES

En JavaScript, los bucles se utilizan para ejecutar una serie de instrucciones repetidamente hasta que se cumpla una condición específica. Los bucles más comunes en JavaScript son el `for`, `while` y `do...while`.

1. **for:** Se utiliza cuando conoces el número exacto de iteraciones que quieres realizar.

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

2. **while:** Se utiliza cuando no conoces el número exacto de iteraciones y la condición de salida se evalúa antes de ejecutar el bloque de código.

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

3. **do...while:** Similar al `while`, pero la condición se evalúa después de ejecutar el bloque de código, lo que garantiza que el bloque de código se ejecute al menos una vez.

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

En cada uno de estos bucles, el código dentro del bloque de bucle se ejecuta repetidamente hasta que la condición especificada se evalúa como falsa. Es importante tener cuidado con bucles infinitos, donde la condición nunca se vuelve falsa y el bucle sigue ejecutándose indefinidamente, lo que puede bloquear el navegador o hacer que el script se ejecute de forma ineficiente.

SWITCH

`switch` es otra estructura de control en JavaScript que se utiliza para tomar decisiones basadas en el valor de una expresión. Es una forma más compacta y legible de escribir condicionales cuando se necesita evaluar una expresión contra múltiples valores posibles.

La estructura básica de un `switch` en JavaScript es la siguiente:

```
switch (expresion) {
  case valor1:
    // código a ejecutar si la expresion coincide con valor1
    break;
  case valor2:
    // código a ejecutar si la expresion coincide con valor2
    break;
  // Puedes tener más casos aquí
  default:
    // código a ejecutar si la expresion no coincide con ningún caso anterior
}
```

Aquí hay un ejemplo de cómo se utiliza `switch` en JavaScript:

```
let dia = 3;
let nombreDia;
switch (dia) {
  case 1:
    nombreDia = "Lunes";
    break;
  case 2:
    nombreDia = "Martes";
    break;
  case 3:
    nombreDia = "Miércoles";
```

```
    break;
  case 4:
    nombreDia = "Jueves";
    break;
  case 5:
    nombreDia = "Viernes";
    break;
  case 6:
    nombreDia = "Sábado";
    break;
  case 7:
    nombreDia = "Domingo";
    break;
  default:
    nombreDia = "Día no válido";
}
console.log("Hoy es " + nombreDia);
```

ANIDAMIENTOS

Ejemplos de sentencias anidadas:

Bucles anidados

```
for (let i = 0; i < 3; i++) {
   for (let j = 0; j < 3; j++) {
      console.log(`i: ${i}, j: ${j}`);
   }
}
```

Condicionales anidados

```
let x = 10;
if (x > 0) {
   if (x < 20) {
      console.log("x es un número positivo menor que 20");
   }
}
```

Funciones anidadas

```
function exterior() {
   function interior() {
      console.log("Función interior ejecutada");
   }
   interior();
}
exterior();
```

COMENTARIOS

```
// Este es un comentario de una sola línea

let x = 5; // Asignamos el valor 5 a la variable x


/*

Este es un comentario

de múltiples líneas.

Se utiliza para explicar

bloques más grandes de código.

*/

let y = 10;
```

FUNCIONES

Las funciones en JavaScript son bloques de código reutilizables que pueden ser llamados y ejecutados en cualquier momento. Permiten encapsular una pieza de código para realizar una tarea específica y pueden aceptar parámetros (datos de entrada) y devolver un valor de salida opcional. Aquí tienes un ejemplo básico de cómo se define y se utiliza una función en JavaScript:

```
// Definición de una función llamada "saludar"

function saludar(nombre) {

   console.log("¡Hola, " + nombre + "!");

}

// Llamada a la función "saludar" con un argumento
```

En este ejemplo:

- Se define una función llamada `saludar` que acepta un parámetro llamado `nombre`.

- Dentro del cuerpo de la función, se utiliza `console.log` para imprimir un mensaje de saludo utilizando el nombre proporcionado como argumento.

- Luego, se llama a la función `saludar` pasando `"Juan"` como argumento.

Las funciones en JavaScript pueden ser mucho más complejas, pueden tener múltiples parámetros, realizar cálculos, contener condicionales y bucles, y pueden incluso devolver un valor utilizando la palabra clave `return`.

Aquí tienes otro ejemplo que devuelve el cuadrado de un número:

```
// Definición de una función llamada "calcularCuadrado"

function calcularCuadrado(numero) {

   return numero * numero; // Devuelve el cuadrado del número

}

// Llamada a la función "calcularCuadrado" con un argumento

let resultado = calcularCuadrado(5);

console.log("El cuadrado de 5 es:", resultado); // Imprime "El cuadrado de 5 es: 25"
```

En este ejemplo, la función `calcularCuadrado` toma un parámetro `numero`, calcula el cuadrado de ese número y devuelve el resultado utilizando la palabra clave `return`. Luego, se almacena el resultado devuelto en la variable `resultado` y se imprime en la consola.

VAR Y LET EN UNA FUNCIÓN

Sí, se puede usar la palabra clave `var` para declarar variables dentro de una función en JavaScript. Sin embargo, es importante comprender las diferencias entre `var`, `let` y `const`, especialmente en lo que respecta al ámbito de las variables.

Antes de la introducción de `let` y `const` en ECMAScript 6 (ES6), `var` era la única forma de declarar variables en JavaScript. Aunque `var` todavía funciona en versiones actuales de JavaScript, su ámbito es diferente al de `let` y `const`.

La diferencia clave radica en el ámbito de las variables:

- Con `var`, la variable está limitada al ámbito de la función en la que se declara o, si se declara fuera de cualquier función, tiene un ámbito global.

- Con `let` y `const`, la variable está limitada al ámbito del bloque en el que se declara. Esto significa que una variable declarada con `let` o `const` dentro de una función solo estará disponible dentro de esa función, no fuera de ella.

Aquí tienes un ejemplo que ilustra cómo se puede usar `var` dentro de una función:

```
function ejemplo() {
  var x = 10;
  console.log("Dentro de la función:", x);
}
ejemplo();
console.log("Fuera de la función:", x); // Esto generará un error porque x no está definido fuera de la función
```

En este ejemplo, `x` está declarada con `var` dentro de la función `ejemplo()`. Esto significa que `x` es local al ámbito de la función `ejemplo()` y no está disponible fuera de ella. Si intentas acceder a `x` fuera de la función, generará un error porque `x` no está definido en ese ámbito.

En resumen, sí, puedes usar `var` dentro de una función en JavaScript, pero debes tener en cuenta las implicaciones del ámbito de las variables al hacerlo. En la mayoría de los casos, es preferible usar `let` o `const` en lugar de `var`, especialmente para evitar problemas relacionados con el ámbito.

EJEMPLOS VARIABLES LOCALES Y GLOBALES

Ejemplo

```
// Variable global
let variableGlobal = "Soy una variable global";
function ejemplo() {
    // Variable local
    let variableLocal = "Soy una variable local";
   console.log("Dentro de la función:");
    console.log("Variable local:", variableLocal);
    console.log("Variable global:", variableGlobal);
}
ejemplo();
console.log("Fuera de la función:");
console.log("Variable local:", typeof variableLocal !== 'undefined' ? variableLocal : "Variable no definida en este ámbito");
console.log("Variable global:", variableGlobal);
```

Ejemplo

```
// Variable global
let nombre = "Juan";
function saludar() {
    // Variable local
    let apellido = "Pérez";
    console.log("Dentro de la función:");
    console.log("Hola,", nombre, apellido);
}
function despedirse() {
    // Usando la variable global
```

```
    console.log("Dentro de otra función:");

    console.log("Adiós,", nombre);

}

saludar();

despedirse();

// Modificando la variable global

nombre = "María";

saludar();

despedirse();

// Intentando acceder a la variable local fuera de su ámbito

console.log("Fuera de las funciones:");

console.log("Apellido:", typeof apellido !== 'undefined' ? apellido : "Variable no definida en este ámbito");
```

En este ejemplo:

- **nombre** es una variable global que almacena el nombre "Juan". Esta variable se puede acceder y modificar desde cualquier parte del código.
- **apellido** es una variable local que almacena el apellido "Pérez". Esta variable solo es accesible dentro de la función **saludar()** donde se declara.

Dentro de las funciones **saludar()** y **despedirse()**, se muestra cómo se pueden usar las variables locales y globales respectivamente. Además, se ilustra cómo se puede modificar la variable global **nombre** y cómo la variable local **apellido** no está disponible fuera de su función.

Este ejemplo muestra cómo las variables locales y globales pueden utilizarse en diferentes partes del código y cómo su ámbito afecta su accesibilidad y visibilidad.

Ejemplo que incluye var let y const

```
// Variable global (se usa var)

var variableGlobalVar = "Soy una variable global (var)";

function ejemplo() {

    // Variable local (se usa let)

    let variableLocalLet = "Soy una variable local (let)";
```

```
    // Constante local (se usa const)
    const PI = 3.1416;
    console.log("Dentro de la función:");
    console.log("Variable global (var):", variableGlobalVar);
    console.log("Variable local (let):", variableLocalLet);
    console.log("Constante local (const):", PI);
}

ejemplo();

// Intentando acceder a las variables locales y constantes fuera de la función

console.log("Fuera de la función:");

console.log("Variable global (var):", variableGlobalVar);

console.log("Variable local (let):", typeof variableLocalLet !== 'undefined' ? variableLocalLet : "Variable no definida en este ámbito");

console.log("Constante local (const):", typeof PI !== 'undefined' ? PI : "Constante no definida en este ámbito");
```

En este ejemplo:

- Se declara **variableGlobalVar** con **var**, lo que la convierte en una variable global.
- Dentro de la función **ejemplo()**, se declara **variableLocalLet** con **let**, lo que la convierte en una variable local a la función.
- También dentro de la función **ejemplo()**, se declara **PI** con **const**, lo que la convierte en una constante local a la función.

Al ejecutar la función **ejemplo()**, se muestra el valor de cada una de las variables y constantes dentro de la función. Fuera de la función, intentamos acceder a las variables locales y a la constante, lo que nos mostrará que **variableGlobalVar** es accesible globalmente, **variableLocalLet** es local a la función y **PI** también es local a la función pero es una constante y no puede ser modificada fuera de la función.

BIBLIOTECAS DE FUNCIONES

Las bibliotecas de funciones en JavaScript son conjuntos de funciones predefinidas que están diseñadas para realizar tareas comunes y que pueden ser reutilizadas en múltiples proyectos. Estas bibliotecas pueden ser creadas por desarrolladores individuales, equipos de desarrollo o por la comunidad en general. Aquí tienes algunos ejemplos de bibliotecas de funciones populares en JavaScript:

1. **jQuery:** Es una biblioteca de JavaScript rápida, pequeña y rica en funciones que simplifica la manipulación del DOM, manejo de eventos, animaciones y llamadas AJAX en JavaScript. Se utiliza ampliamente en el desarrollo web para simplificar el trabajo con JavaScript en múltiples navegadores.

2. **Lodash:** Es una biblioteca de utilidades de JavaScript que proporciona funciones de utilidad para manejar y manipular datos, como arreglos, objetos, cadenas, números, etc. Lodash es conocido por su rendimiento y por su amplia gama de funciones útiles que van más allá de lo que proporciona el estándar de JavaScript.

3. **Moment.js:** Es una biblioteca de JavaScript para manejar, analizar y formatear fechas y horas de una manera sencilla. Moment.js facilita la manipulación de fechas y horas, incluyendo operaciones como el cálculo de diferencia entre fechas, el formato de fechas y la conversión entre formatos de fecha.

4. **React:** Aunque no es una biblioteca de funciones en sí misma, React es una biblioteca de JavaScript para construir interfaces de usuario interactivas y dinámicas. React proporciona un modelo de programación basado en componentes que simplifica el desarrollo de aplicaciones web complejas y escalables.

5. **Express.js:** Es un marco de aplicación web para Node.js que simplifica el proceso de creación de aplicaciones web y APIs. Express.js proporciona una capa de abstracción sobre Node.js que facilita la definición de rutas, manejo de solicitudes HTTP, middleware y otras funcionalidades comunes en el desarrollo web.

Estas son solo algunas de las bibliotecas de funciones populares en JavaScript, pero hay muchas más disponibles para una amplia gama de casos de uso, desde el desarrollo web hasta el procesamiento de datos y más allá. Al utilizar bibliotecas de funciones, los desarrolladores pueden acelerar el desarrollo de aplicaciones, mejorar la calidad del código y reducir la duplicación de esfuerzos al reutilizar código existente.

MANIPULACION DE TEXTO

Las funciones de manipulación de texto en JavaScript son útiles para realizar operaciones comunes en cadenas de caracteres, como buscar, reemplazar, dividir, concatenar, convertir a mayúsculas o minúsculas, entre otras.

INDICES E INMUTABILIDAD

Los índices de cadenas de texto en JavaScript son fundamentales para acceder y manipular caracteres dentro de una cadena.

1. **Índices:**

- En JavaScript, los índices de las cadenas de texto comienzan desde 0. Es decir, el primer carácter de una cadena tiene un índice de 0, el segundo tiene un índice de 1, y así sucesivamente.

- También puedes acceder a los caracteres desde el final de la cadena utilizando índices negativos. El último carácter tiene un índice de -1, el penúltimo tiene un índice de -2, y así sucesivamente.

2. **Acceso a caracteres:**

- Para acceder a un carácter específico dentro de una cadena, se utiliza la notación de corchetes `[]`, seguida del índice del carácter que deseas obtener.

```
const str = "Hola";

console.log(str[0]); // Salida: "H"

console.log(str[1]); // Salida: "o"

console.log(str[-1]); // Salida: undefined

console.log(str[str.length - 1]); // Salida: "a" (último carácter)
```

3. **Longitud de la cadena:**

- La propiedad `length` se utiliza para obtener la longitud de una cadena de texto, es decir, el número de caracteres que contiene.

```
const str = "Hola";

console.log(str.length); // Salida: 4
```

4. **Inmutabilidad:**

- Las cadenas de texto en JavaScript son inmutables, lo que significa que no se pueden modificar directamente. Sin embargo, puedes acceder a los caracteres individuales y crear nuevas cadenas a partir de ellos.

```
let str = "Hola";

// Intentar modificar un carácter
```

```
str[0] = "h";

console.log(str); // Salida: "Hola" (no se ha modificado)
```

En resumen, los índices de cadenas de texto en JavaScript te permiten acceder a caracteres individuales y realizar operaciones basadas en ellos. Son una parte fundamental de la manipulación de cadenas en JavaScript.

FUNCIONES DE MANIPULACIÓN DE CADENAS DE CARACTERES

Aquí tienes algunas funciones de manipulación de texto incorporadas en JavaScript:

1. **`length`:** Devuelve la longitud de una cadena de texto.

```
let texto = "Hola mundo";

console.log(texto.length); // Imprime 10
```

2. **`charAt(index)`:** Devuelve el carácter en la posición especificada de una cadena de texto.

```
let texto = "Hola mundo";

console.log(texto.charAt(0)); // Imprime "H"
```

3. **`substring(startIndex, endIndex)`:** Devuelve una subcadena de la cadena de texto, comenzando desde `startIndex` hasta `endIndex` (opcional).

```
let texto = "Hola mundo";

console.log(texto.substring(0, 4)); // Imprime "Hola"
```

4. **`indexOf(substring)`:** Devuelve la posición de la primera ocurrencia de `substring` dentro de la cadena de texto, o -1 si no se encuentra.

```
let texto = "Hola mundo";

console.log(texto.indexOf("mundo")); // Imprime 5
```

5. **`replace(searchValue, replaceValue)`:** Reemplaza una subcadena de texto (`searchValue`) con otra (`replaceValue`).

```
let texto = "Hola mundo";

console.log(texto.replace("mundo", "amigo")); // Imprime "Hola amigo"
```

6. **`toUpperCase()` y `toLowerCase()`:** Convierten una cadena de texto a mayúsculas o minúsculas.

```
let texto = "Hola mundo";

console.log(texto.toUpperCase()); // Imprime "HOLA MUNDO"
```

```
console.log(texto.toLowerCase()); // Imprime "hola mundo"
```

Estas son solo algunas de las funciones de manipulación de texto disponibles en JavaScript. Puedes combinar estas funciones para realizar tareas más complejas, como validar entradas de usuario, formatear texto o realizar búsquedas y reemplazos avanzados.

Algunas funciones adicionales de manipulación de texto en JavaScript:

7. **`trim()`:** Elimina los espacios en blanco al principio y al final de una cadena de texto.

```
let texto = "   Hola mundo   ";

console.log(texto.trim()); // Imprime "Hola mundo"
```

8. **`split(separator)`:** Divide una cadena de texto en un array de subcadenas basadas en un separador especificado.

```
let texto = "Hola,mundo,cómo,estás";

let array = texto.split(",");

console.log(array); // Imprime ["Hola", "mundo", "cómo", "estás"]
```

9. **`concat(string1, string2, ...)`:** Combina dos o más cadenas de texto y devuelve una nueva cadena.

```
let str1 = "Hola";

let str2 = "mundo";

console.log(str1.concat(" ", str2)); // Imprime "Hola mundo"
```

10. **`startsWith(searchString)` y `endsWith(searchString)`:** Comprueban si una cadena de texto comienza o termina con una subcadena especificada, respectivamente.

```
let texto = "Hola mundo";

console.log(texto.startsWith("Hola")); // true

console.log(texto.endsWith("mundo")); // true
```

11. **`slice(startIndex, endIndex)`:** Extrae una sección de una cadena de texto y devuelve una nueva cadena, desde `startIndex` hasta `endIndex` (opcional).

```
let texto = "Hola mundo";

console.log(texto.slice(0, 4)); // Imprime "Hola"

const str = 'JavaScript es genial';

const subStr1 = str.slice(0, 10); // Extrae desde el índice 0 hasta el 9
```

```
console.log(subStr1); // Salida: "JavaScript"

const subStr2 = str.slice(11); // Extrae desde el índice 11 hasta el final

console.log(subStr2); // Salida: "es genial"

const subStr3 = str.slice(-6); // Extrae los últimos 6 caracteres

console.log(subStr3); // Salida: "genial"

const subStr4 = str.slice(4, -3); // Extrae desde el índice 4 hasta el 3 desde el final

console.log(subStr4); // Salida: "Script es ge"
```

12. **`repeat(count)`:** Devuelve una nueva cadena que contiene la cadena original repetida `count` veces.

```
let str = "Hola ";

console.log(str.repeat(3)); // Imprime "Hola Hola Hola "
```

Estas funciones ofrecen una variedad de formas de manipular y trabajar con cadenas de texto en JavaScript, permitiéndote realizar tareas como división, concatenación, búsqueda, validación y formateo de texto de manera efectiva.

ARRAYS

Los arrays en JavaScript son estructuras de datos que permiten almacenar múltiples valores en una sola variable. Los elementos de un array pueden ser de cualquier tipo, incluyendo números, cadenas, booleanos, objetos e incluso otros arrays. Aquí tienes un ejemplo básico de cómo crear y trabajar con arrays en JavaScript:

```
// Crear un array vacío

let miArray = [];

// Crear un array con elementos

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

let colores = ["rojo", "verde", "azul"];

// Acceder a elementos de un array (indexados desde 0)

console.log(numeros[0]); // Imprime 1

console.log(colores[1]); // Imprime "verde"

// Modificar elementos de un array

numeros[2] = 10;

console.log(numeros); // Imprime [1, 2, 10, 4, 5]

// Longitud de un array

console.log(colores.length); // Imprime 3

// Iterar sobre un array usando un bucle for

for (let i = 0; i < numeros.length; i++) {

console.log(numeros[i]);

}

// Añadir elementos al final de un array

miArray.push("elemento1");

miArray.push("elemento2");

console.log(miArray); // Imprime ["elemento1", "elemento2"]

// Eliminar el último elemento de un array

miArray.pop();

console.log(miArray); // Imprime ["elemento1"] 36
```

Además de estas operaciones básicas, los arrays en JavaScript también admiten una variedad de métodos integrados, como `push()`, `pop()`, `shift()`, `unshift()`, `splice()`, `slice()`, `concat()`, `join()`, `reverse()`, `sort()`, entre otros, que te permiten realizar diversas operaciones de manipulación y transformación en los arrays.

Aquí tienes un ejemplo que utiliza la mayoría de los métodos comunes de los arrays en JavaScript:

// Crear un array

let numeros = [3, 1, 4, 1, 5, 9];

// Mostrar el array original

console.log("Array original:", numeros);

// Método push: Añade elementos al final del array

numeros.push(2);

console.log("Después de push(2):", numeros);

// Método pop: Elimina el último elemento del array

numeros.pop();

console.log("Después de pop():", numeros);

// Método shift: Elimina el primer elemento del array

numeros.shift();

console.log("Después de shift():", numeros);

// Método unshift: Añade elementos al principio del array

numeros.unshift(6, 5);

console.log("Después de unshift(6, 5):", numeros);

// Método splice: Cambia el contenido de un array eliminando elementos existentes y/o añadiendo nuevos elementos

numeros.splice(2, 1, 8);

// Comienza en el índice 2 del array números, elimina 1 elemento a partir de ese índice.

// Luego, agrega el elemento 8 en la posición del índice 2.

console.log("Después de splice(2, 1, 8):", numeros);

// Método slice: Devuelve una copia superficial de una porción del array en un nuevo array

let subArray = numeros.slice(1, 4);

console.log("Resultado de slice(1, 4):", subArray); 37

// Método concat: Combina dos o más arrays

let otrosNumeros = [7, 9, 0];

let numerosConcatenados = numeros.concat(otrosNumeros);

console.log("Resultado de concat(otrosNumeros):", numerosConcatenados);

// Método join: Une todos los elementos de un array en una cadena

let cadena = numeros.join("-");

console.log("Resultado de join('-'):", cadena);

// Método reverse: Invierte el orden de los elementos del array

numeros.reverse();

console.log("Después de reverse():", numeros);

// Método sort: Ordena los elementos de un array

numeros.sort();

console.log("Después de sort():", numeros);

// Método indexOf: Devuelve el primer índice en el que se encuentra un elemento específico en el array

let indice = numeros.indexOf(5);

console.log("Índice de 5:", indice);

// Método lastIndexOf: Devuelve el último índice en el que se encuentra un elemento específico en el array

let ultimoIndice = numeros.lastIndexOf(5);

console.log("Último índice de 5:", ultimoIndice);

// Método includes: Determina si un array incluye un determinado elemento

let incluido = numeros.includes(8);

console.log("¿Está incluido el 8?:", incluido);

// Método forEach: Ejecuta una función proporcionada una vez por cada elemento del array

console.log("Recorriendo el array con forEach():");

```
numeros.forEach(function(elemento) {

console.log(elemento);

}); 38
```

// Método map: Crea un nuevo array con los resultados de la llamada a la función indicada aplicados a cada uno de sus elementos

```
let duplicados = numeros.map(function(elemento) {

return elemento * 2;

});

console.log("Resultado de map(elemento * 2):", duplicados);
```

// Método filter: Crea un nuevo array con todos los elementos que pasen la prueba implementada por la función proporcionada

```
let impares = numeros.filter(function(elemento) {

return elemento % 2 !== 0;

});

console.log("Resultado de filter(elemento % 2 !== 0):", impares);
```

// Método reduce: Aplica una función a un acumulador y a cada valor de un array (de izquierda a derecha) para reducirlo a un único valor

```
let suma = numeros.reduce(function(acumulador, elemento) {

return acumulador + elemento;

}, 0);

console.log("Resultado de reduce(suma):", suma);
```

Este ejemplo muestra cómo utilizar la mayoría de los métodos comunes de los arrays en JavaScript para realizar diversas operaciones, como añadir y eliminar elementos, modificar el contenido, buscar elementos, transformar el array y realizar cálculos sobre sus elementos.

ARRAYS MULTIDIMENSIONALES

Los arrays multidimensionales en JavaScript son arrays que contienen otros arrays como elementos. Esto permite crear estructuras de datos más complejas, como matrices o tablas. Cada elemento de un array multidimensional puede ser, a su vez, otro array, formando así una "matriz" con filas y columnas.

Aquí tienes un ejemplo de cómo se puede crear y trabajar con un array bidimensional en JavaScript:

```
// Crear un array bidimensional (matriz)

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

// Acceder a elementos de la matriz

console.log(matriz[0][0]); // Imprime el primer elemento de la primera fila (1)

console.log(matriz[1][2]); // Imprime el tercer elemento de la segunda fila (6)

// Iterar sobre una matriz bidimensional usando bucles for anidados

for (let i = 0; i < matriz.length; i++) {

for (let j = 0; j < matriz[i].length; j++) {

console.log(matriz[i][j]);

}

}
```

En este ejemplo, `matriz` es un array bidimensional que contiene tres arrays internos, cada uno representando una fila de la matriz. Puedes acceder a los elementos de la matriz utilizando dos índices: el primero para seleccionar la fila y el segundo para seleccionar la columna.

Los arrays multidimensionales son útiles para representar datos estructurados en forma de tablas o cuadrículas, como datos de una hoja de cálculo, píxeles de una imagen o elementos de un juego de mesa.

Ejemplo de array multidimensional. Este ejemplo muestra cómo crear, acceder, modificar, iterar, realizar cálculos y realizar modificaciones en una matriz bidimensional en JavaScript.

```
// Definir una matriz (array bidimensional)

let matriz = [

[1, 2, 3],

[4, 5, 6],

[7, 8, 9]

];

// Mostrar la matriz original
```

```
console.log("Matriz original:"); 40
console.log(matriz);
// Acceder a un elemento específico de la matriz
console.log("Elemento en la fila 2, columna 3:", matriz[1][2]); // Imprime 6
// Modificar un elemento de la matriz
matriz[0][0] = 10;
console.log("Matriz después de modificar el elemento en la fila 1, columna 1:");
console.log(matriz);
// Iterar sobre la matriz y mostrar todos los elementos
console.log("Mostrando todos los elementos de la matriz:");
for (let i = 0; i < matriz.length; i++) {
for (let j = 0; j < matriz[i].length; j++) {
console.log("Elemento en la fila " + (i + 1) + ", columna " + (j + 1) + ":", matriz[i][j]);
}
}
// Calcular la suma de todos los elementos de la matriz
let suma = 0;
for (let i = 0; i < matriz.length; i++) {
for (let j = 0; j < matriz[i].length; j++) {
suma += matriz[i][j];
} }
console.log("La suma de todos los elementos de la matriz es:", suma);
// Añadir una nueva fila a la matriz
let nuevaFila = [10, 11, 12];
matriz.push(nuevaFila);
console.log("Matriz después de añadir una nueva fila:");
console.log(matriz);
```

CLASES Y OBJETOS

1. **Definición de clase:**

- En JavaScript, una clase es una plantilla para crear objetos. Define las propiedades y métodos que los objetos de esa clase tendrán.

2. **Sintaxis:**

- La sintaxis para definir una clase en JavaScript es la siguiente:

```
class NombreDeLaClase {
  constructor(parametros) {
    // Código para inicializar el objeto
  }
  metodo1() {
    // Código del método 1
  }
  metodo2() {
    // Código del método 2
  }
  // Otros métodos...
}
```

3. **Constructor:**

- El método `constructor` es un método especial que se llama automáticamente cuando se crea un nuevo objeto de la clase. Se utiliza para inicializar las propiedades del objeto.

- Puedes pasar parámetros al constructor para configurar las propiedades del objeto durante la creación.

4. **Métodos:**

- Los métodos son funciones asociadas a la clase que pueden ser llamadas en los objetos creados a partir de ella.

- Puedes definir cualquier cantidad de métodos en una clase para encapsular la lógica relacionada con el comportamiento de los objetos.

5. **Creación de objetos:**

- Para crear un nuevo objeto a partir de una clase, utilizamos la palabra clave `new` seguida del nombre de la clase y los argumentos necesarios para el constructor.

- Por ejemplo:

```
const objeto = new NombreDeLaClase(argumentos);
```

6. **Acceso a propiedades y métodos:**

- Las propiedades y métodos de un objeto se acceden utilizando la notación de punto (`.`).

- Por ejemplo:

objeto.metodo();

let valor = objeto.propiedad;

7. **Herencia:**

- JavaScript admite la herencia de clases, lo que significa que una clase puede heredar propiedades y métodos de otra clase.

- Esto se logra utilizando la palabra clave `extends`.

- Por ejemplo:

class SubClase extends ClaseBase {

// Métodos y propiedades adicionales de la subclase

}

EJEMPLO DE CLASE

Ejemplo simple de una clase en JavaScript que representa un `Rectángulo`:

```
class Rectangulo {
  constructor(ancho, alto) {
    this.ancho = ancho;
    this.alto = alto;
  }
  // Método para calcular el área del rectángulo
  calcularArea() {
    return this.ancho * this.alto;
  }
  // Método para calcular el perímetro del rectángulo
  calcularPerimetro() {
    return 2 * (this.ancho + this.alto);
  }
}
// Crear un objeto de la clase Rectángulo
const rectangulo = new Rectangulo(5, 7);
// Acceder a propiedades y métodos del objeto
console.log("Ancho:", rectangulo.ancho); // Salida: 5
console.log("Alto:", rectangulo.alto); // Salida: 7
console.log("Área:", rectangulo.calcularArea()); // Salida: 35
console.log("Perímetro:", rectangulo.calcularPerimetro()); // Salida: 24
```

En este ejemplo, la clase `Rectangulo` tiene un constructor que toma dos parámetros, `ancho` y `alto`, y los asigna a propiedades de instancia. También tiene dos métodos, `calcularArea()` y `calcularPerimetro()`, que calculan el área y el perímetro del rectángulo, respectivamente.

Puedes crear múltiples objetos de la clase `Rectangulo`, cada uno con sus propios valores de `ancho` y `alto`, y utilizar los métodos para calcular propiedades específicas de cada objeto. Esto muestra cómo las clases en JavaScript pueden encapsular datos y funcionalidades relacionadas en un solo objeto.

HERENCIA

Vamos a crear una clase base `Figura` y luego dos subclases, `Rectangulo` y `Circulo`, que heredan de ella:

```
class Figura {
  constructor(color) {
    this.color = color;
  }
  // Método para obtener el color de la figura
  getColor() {
    return this.color;
  }
  // Método abstracto para calcular el área (será implementado en las subclases)
  calcularArea() {
    throw new Error("El método calcularArea() debe ser implementado en las subclases.");
  }
}
// Subclase Rectangulo que hereda de Figura
class Rectangulo extends Figura {
  constructor(color, ancho, alto) {
    super(color);
    this.ancho = ancho;
    this.alto = alto;
  }
  // Implementación del método calcularArea para Rectangulo
  calcularArea() {
    return this.ancho * this.alto;
  }
}
// Subclase Circulo que hereda de Figura
class Circulo extends Figura {
  constructor(color, radio) {
    super(color);
    this.radio = radio;
  }
  // Implementación del método calcularArea para Circulo
  calcularArea() {
    return Math.PI * this.radio * this.radio;
```

```
  }
}
// Crear objetos de las subclases
const rectangulo = new Rectangulo("rojo", 5, 7);
const circulo = new Circulo("azul", 3);
// Acceder a propiedades y métodos de los objetos
console.log("Color del rectángulo:", rectangulo.getColor()); // Salida: rojo
console.log("Área del rectángulo:", rectangulo.calcularArea()); // Salida: 35
console.log("Color del círculo:", circulo.getColor()); // Salida: azul
console.log("Área del círculo:", circulo.calcularArea().toFixed(2)); // Salida: 28.27
```

En este ejemplo, la clase base `Figura` tiene un constructor que inicializa el color de la figura y un método `getColor()` para obtener ese color. También tiene un método `calcularArea()` que se define como abstracto (es decir, no tiene implementación) y debe ser implementado en las subclases.

Las subclases `Rectangulo` y `Circulo` heredan de `Figura` utilizando la palabra clave `extends`. Cada subclase tiene su propio constructor y una implementación del método `calcularArea()` que calcula el área específica para esa figura.

Este ejemplo ilustra cómo puedes usar la herencia en JavaScript para crear una jerarquía de clases donde las subclases heredan propiedades y métodos de una clase base, y también pueden tener sus propias implementaciones específicas.

POLIMORFISMO

El polimorfismo es un concepto fundamental en la programación orientada a objetos que permite que los objetos de diferentes clases respondan al mismo mensaje (llamada a un método) de manera específica para cada clase. En JavaScript, aunque no tiene una implementación directa como en lenguajes orientados a objetos tradicionales como Java o C++, puedes lograr un tipo de polimorfismo mediante el uso de clases y métodos en JavaScript. Aquí tienes una explicación sobre cómo funciona:

1. **Polimorfismo en JavaScript:**

- En JavaScript, puedes lograr el polimorfismo utilizando la herencia y la sobrescritura de métodos. Cuando una subclase hereda de una clase base, puede sobrescribir (redefinir) los métodos de la clase base para adaptar su comportamiento a las necesidades específicas de la subclase.

- Esto significa que un método con el mismo nombre puede tener diferentes implementaciones en diferentes clases.

2. **Ejemplo:**

Supongamos que tienes una clase base `Animal` con un método `hacerSonido()`, y dos subclases `Perro` y `Gato` que heredan de la clase base y sobrescriben el método `hacerSonido()` para producir diferentes sonidos:

```
class Animal {
  hacerSonido() {
    console.log("Haciendo algún sonido genérico...");
  }
}
class Perro extends Animal {
  hacerSonido() {
    console.log("Guau!");
  }
}
class Gato extends Animal {
  hacerSonido() {
    console.log("Miau!");
  }
}
const perro = new Perro();
const gato = new Gato();
perro.hacerSonido(); // Salida: "Guau!"
gato.hacerSonido(); // Salida: "Miau!"
```

En este ejemplo, tanto `Perro` como `Gato` heredan el método `hacerSonido()` de la clase `Animal`, pero cada uno lo sobrescribe con su propia implementación específica.

3. **Beneficios del polimorfismo:**

- El polimorfismo permite escribir código más flexible y modular, ya que puedes tratar objetos de diferentes clases de manera uniforme si responden al mismo mensaje (es decir, tienen el mismo método).

- Facilita la extensión del código, ya que puedes agregar nuevas clases que implementen un comportamiento específico sin tener que modificar el código existente.

En resumen, aunque JavaScript no implementa el polimorfismo de la misma manera que otros lenguajes orientados a objetos, puedes lograr un tipo de polimorfismo utilizando la herencia y la sobrescritura de métodos. Esto te permite escribir código más flexible y modular, lo que facilita la extensión y el mantenimiento del código.

INTERFACES DE CLASES

En JavaScript, a diferencia de otros lenguajes como TypeScript, no hay una implementación nativa de interfaces de clases. Sin embargo, puedes simular interfaces mediante la definición de contratos de clases utilizando comentarios o convenciones de nomenclatura. Estos contratos pueden ayudar a definir la estructura esperada de una clase, incluyendo qué métodos y propiedades deberían estar presentes en ella.

Aquí tienes un ejemplo de cómo podrías simular una interfaz de clase en JavaScript utilizando comentarios:

```
/**
 * @interface Figura
 * @property {string} color - Color de la figura.
 * @method calcularArea - Método para calcular el área de la figura.
 */
class Rectangulo {
  /**
   * Crea un rectángulo.
   * @param {number} ancho - Ancho del rectángulo.
   * @param {number} alto - Alto del rectángulo.
   * @param {string} color - Color del rectángulo.
   */
  constructor(ancho, alto, color) {
    this.ancho = ancho;
    this.alto = alto;
    this.color = color;
  }
  /**
   * Calcula el área del rectángulo.
   * @returns {number} El área del rectángulo.
   */
  calcularArea() {
    return this.ancho * this.alto;
  }
}
// Implementación de la interfaz Figura en la clase Rectangulo
/** @implements {Figura} */
```

En este ejemplo, se define una interfaz de clase llamada `Figura` utilizando comentarios JSDoc sobre la clase `Rectangulo`. La interfaz `Figura` especifica que debe tener una propiedad `color` y un método `calcularArea()`. Luego, la clase `Rectangulo` implementa esta interfaz, asegurándose de que tenga las propiedades y métodos requeridos.

Es importante destacar que esto es una simulación de interfaces y no hay un chequeo de tipo estático en tiempo de compilación como lo haría un lenguaje como TypeScript. Sin embargo, seguir estas convenciones puede ayudar a documentar y entender mejor la estructura de las clases en tu código JavaScript.

EJEMPLO DE UN INTERFAZ DE CLASE EN JAVA PARA QUE SE ENTIENDAN LAS INTERFACES

Java por ejemplo sí implementa los interfaces de clases.

En Java las interfaces son una parte fundamental de la programación orientada a objetos. Te proporcionaré un ejemplo básico de cómo se define e implementa una interfaz de clase en Java:

Supongamos que tenemos una interfaz llamada `Figura` que define un método `calcularArea()` para calcular el área de una figura geométrica. Aquí está cómo se vería:

```
// Definición de la interfaz Figura

public interface Figura {

   double calcularArea();

}
```

En este ejemplo, `Figura` es una interfaz que define un único método `calcularArea()`. No proporciona ninguna implementación para este método; solo declara su firma.

Ahora, supongamos que tenemos una clase `Rectangulo` que implementa la interfaz `Figura` y proporciona su propia implementación del método `calcularArea()`:

```
// Implementación de la clase Rectangulo que implementa la interfaz Figura
public class Rectangulo implements Figura {
   private double ancho;
   private double alto;
   // Constructor
   public Rectangulo(double ancho, double alto) {
      this.ancho = ancho;
      this.alto = alto;
   }
   // Implementación del método calcularArea() para el Rectangulo
   @Override
   public double calcularArea() {
      return ancho * alto;
   }
```

}
En este ejemplo, `Rectangulo` implementa la interfaz `Figura` utilizando la palabra clave `implements`. Esto significa que `Rectangulo` debe proporcionar una implementación del método `calcularArea()`. La implementación del método `calcularArea()` en `Rectangulo` calcula el área del rectángulo multiplicando su ancho por su alto.

En resumen, en Java una interfaz de clase define un contrato que las clases concretas pueden implementar. Esto permite que las clases proporcionen funcionalidad específica mientras se aseguran de que cumplan con ciertas especificaciones definidas por la interfaz.

SECUENCIAS DE ESCAPE PAG 119

Las secuencias de escape en JavaScript son combinaciones de caracteres que se utilizan para representar caracteres que de otra manera serían difíciles o imposibles de escribir en un programa. Aquí tienes algunas secuencias de escape comunes en JavaScript:

1. **\n**: Representa un salto de línea.

2. **\t**: Representa un tabulador.

3. **\r**: Representa un retorno de carro.

4. ****: Representa un carácter de barra invertida.

5. **\'**: Representa una comilla simple.

6. **\"**: Representa una comilla doble.

7. **\b**: Representa un retroceso (backspace).

8. **\f**: Representa un avance de página.

Estas secuencias de escape son útiles cuando necesitas incluir caracteres especiales dentro de cadenas de texto en JavaScript. Por ejemplo:

```
console.log("Hola\nMundo"); // Imprime "Hola" en una línea y "Mundo" en la siguiente

console.log("Texto con una\ttabulación"); // Imprime "Texto con una    tabulación"

console.log("Comilla simple: \' y comilla doble: \" "); // Imprime "Comilla simple: ' y comilla doble: " "
```

Es importante tener en cuenta estas secuencias de escape al trabajar con cadenas de texto en JavaScript para asegurarte de que los caracteres especiales se interpreten correctamente.

Aquí tienes un ejemplo que utiliza todas las secuencias de escape comunes en una sola cadena de texto:

```
console.log("Hola\\nMundo\t\'Esto es una\\b prueba\\f de secuencias\\rescape\'\"");

// Imprime:

// Hola\nMundo    'Esto es una\b prueba  de secuencias

// escape'"
```

Este ejemplo muestra cómo usar cada una de las secuencias de escape comunes en una cadena de texto. Cada secuencia de escape se representa correctamente dentro de la cadena, y la salida en la consola muestra cómo se interpretan estas secuencias al imprimir la cadena.

FORMATOS ESTANDAR DE ALMACENAMIENTO DE DATOS pag 167

OBJETOS GLOBALES JAVASCRIPT pag 127

Libro: Math Document Window

Lista de algunos objetos globales en JavaScript:

1. **Object**: Proporciona métodos y propiedades para trabajar con objetos en JavaScript.

2. **Function**: Proporciona métodos y propiedades para trabajar con funciones en JavaScript.

3. **Array**: Proporciona métodos para trabajar con matrices en JavaScript.

4. **String**: Proporciona métodos para trabajar con cadenas de texto en JavaScript.

5. **Boolean**: Proporciona métodos y propiedades para trabajar con valores booleanos en JavaScript.

6. **Number**: Proporciona métodos y propiedades para trabajar con valores numéricos en JavaScript.

7. **Date**: Proporciona métodos para trabajar con fechas y horas en JavaScript.

8. **Math**: Proporciona métodos y constantes para operaciones matemáticas en JavaScript.

9. **RegExp**: Proporciona funcionalidades para trabajar con expresiones regulares en JavaScript.

10. **Error**: Proporciona objetos de error que representan errores en el código JavaScript.

11. **JSON**: Proporciona métodos para trabajar con datos en formato JSON (JavaScript Object Notation).

12. **Promise**: Proporciona una forma de realizar operaciones asincrónicas y manejar sus resultados en JavaScript.

13. **Map**: Proporciona una colección de pares clave-valor en JavaScript.

14. **Set**: Proporciona una colección de valores únicos en JavaScript.

15. **Symbol**: Proporciona una forma de crear identificadores únicos en JavaScript.

16. **WeakMap**: Proporciona una colección de pares clave-valor débiles en JavaScript.

17. **WeakSet**: Proporciona una colección de valores débiles en JavaScript.

18. **Proxy**: Proporciona una forma de interceptar y personalizar operaciones en objetos en JavaScript.

19. **Reflect**: Proporciona métodos estáticos para realizar operaciones sobre objetos en JavaScript.

20. **Intl**: Proporciona funcionalidades para manejar formatos de fechas, números y cadenas de texto internacionalizados en JavaScript.

21. **Iterator y Iterable**: Proporciona un protocolo para definir y utilizar iteradores en JavaScript.

22. **ArrayBuffer**: Proporciona una forma de crear búferes de datos binarios en JavaScript.

23. **SharedArrayBuffer**: Proporciona una forma de compartir datos binarios entre procesos en JavaScript (restringido en navegadores por razones de seguridad).

24. **Atomics**: Proporciona operaciones atómicas sobre búferes de datos compartidos en JavaScript (restringido en navegadores por razones de seguridad).

25. **Document**: Proporciona métodos y propiedades para interactuar con el DOM (Document Object Model) en JavaScript.

26. **Window**: Representa la ventana del navegador y proporciona métodos y propiedades para controlarla y manipularla, así como para interactuar con el entorno del navegador.

27. **Navigator**: Proporciona información sobre el navegador del usuario, como el nombre, la versión y las características.

28. **Location**: Proporciona información sobre la URL actual y permite redirigir a otras páginas web.

29. **History**: Permite navegar hacia adelante y hacia atrás a través del historial del navegador.

30. **XMLHttpRequest**: Proporciona funcionalidades para realizar peticiones HTTP asíncronas desde JavaScript, comúnmente utilizado para realizar solicitudes AJAX.

31. **Blob**: Representa datos binarios inmutables, que pueden ser utilizados para manipular archivos en el navegador.

32. **FormData**: Proporciona una forma fácil de construir un conjunto de pares clave/valor que representan campos y valores de un formulario HTML.

33. **Console**: Proporciona métodos para enviar mensajes a la consola del navegador o de Node.js para propósitos de depuración.

Estos son algunos de los objetos globales más comunes y útiles en JavaScript, cada uno proporcionando funcionalidades específicas para diferentes aspectos de la programación en JavaScript. Cada uno de ellos tiene su propio conjunto de métodos y propiedades que proporcionan funcionalidades útiles para diferentes aspectos de la programación en JavaScript.