1. Tipos de datos

String: Es un tipo de dato que representa texto. Por ejemplo: "Hola Mundo".

Number: Representa números, ya sean enteros o decimales. Por ejemplo: 42 o 3.14.

Boolean: Representa un valor verdadero o falso. Solo puede ser true o false.

Array: Es una lista ordenada de valores, que pueden ser de cualquier tipo. Por ejemplo: [1, 2, 3] o ["a", "b", "c"].

Object: Es una colección de pares clave-valor, donde cada clave es única. Por ejemplo: { nombre: "Juan", edad: 30 }.

Null: Representa la ausencia intencional de cualquier valor. Es un valor asignado explícitamente.

Undefined: Representa una variable que ha sido declarada pero no se le ha asignado ningún valor.

2. Operadores y estructuras de control

Operadores: Son símbolos que te permiten realizar operaciones con variables y valores. Por ejemplo:

Aritméticos: +, -, *, /

Comparación: ==, ===, !=, !==, >, <

Lógicos: &&, ||, !

Estructuras de control:

if

Permite ejecutar un bloque de código si una condición es verdadera. Por ejemplo:

```
if (condición1) {
  // Código si condición1 es verdadera
} else if (condición2) {
  // Código si condición2 es verdadera
} else {
  // Código si ninguna condición es verdadera
}
```

condiciones compuestas

Puedes combinar múltiples condiciones usando operadores lógicos como && (AND), || (OR) y ! (NOT).

```
if (edad >= 18 && tieneLicencia)
if (esEstudiante || esProfesor)

let esAdmin = false;
if (!esAdmin) {
  console.log("No tienes permisos de administrador.");
}
```

Aquí, como esAdmin es false, el ! lo convierte en true, por lo que se imprimirá "No tienes permisos de administrador.".

if anidados

```
if (edad >= 18) {
  if (tieneLicencia) {
  }
}
```

operador ternario

```
condición ? expresiónSiVerdadera : expresiónSiFalsa;

let edad = 18;
let mensaje = edad >= 18 ? "Mayor de edad" : "Menor de edad";
console.log(mensaje); // "Mayor de edad
```

Truthy y Falsy

En JavaScript, las condiciones no solo evalúan true o false, sino también valores "truthy" o "falsy".

Valores Falsy:

- false
- 0
- "" (cadena vacía)
- null
- undefined
- NaN

Valores Truthy: Cualquier valor que no sea falsy es truthy.

```
let nombre = "";

if (nombre) {
  console.log("Hola, " + nombre);
```

```
} else {
  console.log("No hay nombre.");
}
```

Aquí, como nombre es una cadena vacía (falsy), se imprimirá "No hay nombre.".

if con funciones

Puedes usar if para controlar el flujo de una función.

```
function esPositivo(numero) {
  if (numero > 0) {
    return "Positivo";
  } else if (numero < 0) {
    return "Negativo";
  } else {
    return "Cero";
  }
}

console.log(esPositivo(5)); // "Positivo"
```

if con objetos y arrays

Puedes usar if para verificar propiedades de objetos o elementos de arrays.

Ejemplo con objetos:

```
let usuario = {
  nombre: "Juan",
  edad: 25
};

if (usuario.edad >= 18) {
  console.log(usuario.nombre + " es mayor de edad.");
}
```

Ejemplo con arrays:

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

if (numeros.length > 0) {
  console.log("El array no está vacío.");
}
```

if avanzado: Cortocircuito

Puedes usar el cortocircuito para simplificar condiciones.

Si la primera expresión es falsy, no evalúa la segunda expresión porque el resultado ya es false.

Si la primera expresión es truthy, evalúa la segunda expresión y devuelve su valor.

Ejemplo con &&:

```
let usuario = { nombre: "Ana" };

if (usuario && usuario.nombre) {
  console.log("Hola, " + usuario.nombre);
}
```

Si la primera expresión es truthy, no evalúa la segunda expresión porque el resultado ya es true.

Si la primera expresión es falsy, evalúa la segunda expresión y devuelve su valor.

Ejemplo con ||:

```
let nombre = "";
let nombrePorDefecto = nombre || "Invitado";
console.log(nombrePorDefecto); // "Invitado"
```

early return

consiste en verificar primero las condiciones negativas y salir de la función o bloque de código inmediatamente si no se cumplen. Esto reduce la necesidad de anidar if y hace que el código sea más legible.

```
function verificarUsuario(usuario) {
  if (usuario) {
    if (usuario.edad >= 18) {
      if (usuario.tieneLicencia) {
        console.log("Puedes conducir.");
      } else {
        console.log("Necesitas una licencia.");
      }
    } else {
      console.log("Eres menor de edad.");
    }
  } else {
    console.log("Usuario no válido.");
  }
}

function verificarUsuario(usuario) {
```

```
  // Verificamos primero las condiciones negativas
  if (!usuario) {
    console.log("Usuario no válido.");
    return; // Salimos de la función
  }

  if (usuario.edad < 18) {
    console.log("Eres menor de edad.");
    return; // Salimos de la función
  }

  if (!usuario.tieneLicencia) {
    console.log("Necesitas una licencia.");
    return; // Salimos de la función
  }

  // Si todas las condiciones se cumplen
  console.log("Puedes conducir.");
}
```

Ventajas: Menos anidación: El código es más plano y fácil de leer.

Claridad: Cada condición se maneja por separado, lo que facilita entender el flujo del código.

Mantenimiento: Es más fácil agregar o modificar condiciones sin afectar el resto del código.

Sin early return:

```
function validarFormulario(formulario) {
  if (formulario.nombre) {
    if (formulario.email) {
      if (formulario.contraseña) {
        console.log("Formulario válido.");
      } else {
        console.log("La contraseña es requerida.");
      }
    } else {
      console.log("El email es requerido.");
    }
  } else {
    console.log("El nombre es requerido.");
  }
}
```

Con early return:

```
function validarFormulario(formulario) {
  if (!formulario.nombre) {
    console.log("El nombre es requerido.");
```

```
    return;
  }

  if (!formulario.email) {
    console.log("El email es requerido.");
    return;
  }

  if (!formulario.contraseña) {
    console.log("La contraseña es requerida.");
    return;
  }

  console.log("Formulario válido.");
}
```

Alternativas al early return

Si el "early return" no es suficiente o no se adapta a tu caso, puedes usar otras técnicas para reducir la anidación:

a) Usar switch: Útil cuando tienes múltiples condiciones basadas en el valor de una variable.

```
switch (usuario.rol) {
  case "admin":
    console.log("Eres administrador.");
    break;
  case "usuario":
    console.log("Eres usuario normal.");
    break;
  default:
    console.log("Rol no válido.");
}
```

b) Usar objetos o mapas: Útil para reemplazar múltiples if-else o switch.

```
const roles = {
  admin: "Eres administrador.",
  usuario: "Eres usuario normal.",
};

console.log(roles[usuario.rol] || "Rol no válido.");
```

c) Encadenamiento opcional (?.): Útil para evitar errores al acceder a propiedades de objetos anidados.

```
if (usuario?.direccion?.ciudad === "Madrid") {
  console.log("Vives en Madrid.");
```

```
}
```

switch:

Es una estructura que permite ejecutar diferentes bloques de código dependiendo del valor de una variable. Por ejemplo:

```
switch (dia) {
    case "Lunes":
        console.log("Es lunes");
        break;
    case "Martes":
        console.log("Es martes");
        break;
    default:
        console.log("No es lunes ni martes");
}
```

Puedes agrupar varios case para que ejecuten el mismo bloque de código. Esto es útil cuando varios valores deben tener el mismo comportamiento.

```
let mes = 2;
let estacion;

switch (mes) {
  case 12:
  case 1:
  case 2:
    estacion = "Invierno";
    break;
  case 3:
  case 4:
  case 5:
    estacion = "Primavera";
    break;
  case 6:
  case 7:
  case 8:
    estacion = "Verano";
    break;
  case 9:
  case 10:
  case 11:
    estacion = "Otoño";
    break;
  default:
    estacion = "Mes no válido";
}
console.log(estacion); // Salida: "Invierno"
```

for

Permite repetir un bloque de código un número específico de veces. Por ejemplo:

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

let frutas = ["Manzana", "Banana", "Cereza"];

for (let i = 0; i < frutas.length; i++) {
  console.log(frutas[i]);
}
```

for con múltiples variables

```


for (let i = 0, j = 10; i < j; i++, j--) {
  console.log(`i: ${i}, j: ${j}`);
}

Salida:

i: 0, j: 10
i: 1, j: 9
i: 2, j: 8
i: 3, j: 7
i: 4, j: 6
```

for con break y continue

- **break:** Termina el bucle inmediatamente.
- **continue:** Salta el resto del código en la iteración actual y pasa a la siguiente iteración.

```
for (let i = 0; i < 10; i++) {
  if (i === 5) {
    break;
  }
  console.log(i);//0,1,2,3,4
}

for (let i = 0; i < 5; i++) {
  if (i === 2) {
    continue;
  }
```

```
  console.log(i);//0,1,3,4
}
```

for...of (ES6)

El bucle for...of es una variante de for que permite recorrer elementos de objetos iterables como arrays, strings, etc.

```
let frutas = ["Manzana", "Banana", "Cereza"];

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

for...in

El bucle for...in se utiliza para recorrer las propiedades enumerables de un objeto.

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

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

//nombre: Juan
// edad: 30
// profesion: Desarrollador
```

for anidados

Puedes anidar bucles for para trabajar con estructuras de datos multidimensionales.

```
for (let i = 0; i < 3; i++) {
  for (let j = 0; j < 3; j++) {
    console.log(`i: ${i}, j: ${j}`);
  }
}
// i: 0, j: 0
// i: 0, j: 1
// i: 0, j: 2
// i: 1, j: 0
// i: 1, j: 1
// i: 1, j: 2
```

```
// i: 2, j: 0
// i: 2, j: 1
// i: 2, j: 2
```

for con funciones

Puedes usar funciones dentro de un bucle for para realizar operaciones más complejas.

```
function factorial(n) {
  let resultado = 1;
  for (let i = 1; i <= n; i++) {
    resultado *= i;
  }
  return resultado;
}

console.log(factorial(5)); // Salida: 120
```

while

Repite un bloque de código mientras una condición sea verdadera.

El bucle do...while es similar al while, pero con una diferencia clave: la condición se evalúa después de cada iteración. Esto garantiza que el bloque de código se ejecute al menos una vez, incluso si la condición es falsa desde el principio.Por ejemplo:

```
let contador = 10;

// while
while (contador < 5) {
  console.log("While: " + contador);
  contador++;
}

// do...while
do {
  console.log("Do...While: " + contador);
  contador++;
} while (contador < 5);
```

Uso de break y continue

Tanto en while como en do...while, puedes usar break para salir del bucle y continue para saltar a la siguiente iteración.

Ejemplo con break:

```
let contador = 0;

while (contador < 10) {
  if (contador === 5) {
    break; // Sale del bucle cuando contador es 5
  }
  console.log(contador);
  contador++;
}

Salida:

0
1
2
3
4
```

Ejemplo con continue:

```
let contador = 0;

while (contador < 5) {
  contador++;
  if (contador === 3) {
    continue; // Salta la iteración cuando contador es 3
  }
  console.log(contador);
}
Salida:


1
2
4
5
```

map: Es un método de los arrays que permite transformar cada elemento del array. Por ejemplo:

```
const numeros = [1, 2, 3];
const dobles = numeros.map(num => num * 2);
console.log(dobles); // [2, 4, 6]
```

filter: Filtra los elementos de un array que cumplen una condición. Por ejemplo:

```
const numeros = [1, 2, 3, 4];
const pares = numeros.filter(num => num % 2 === 0);
console.log(pares); // [2, 4]
```

reduce: Reduce un array a un único valor aplicando una función acumuladora. Por ejemplo:

```
const numeros = [1, 2, 3, 4];
const suma = numeros.reduce((acc, num) => acc + num, 0);
console.log(suma); // 10
```

3. Funciones y scope

Funciones

Declaración de función (Function Declaration)

Son bloques de código que puedes llamar para ejecutar una tarea específica. Por ejemplo:

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

Expresión de función (Function Expression)

Una función también puede ser asignada a una variable. Esto se llama expresión de función.

Sintaxis:

```
const nombreDeLaFuncion = function(parametro1, parametro2) {
  // Código a ejecutar
  return resultado; // Opcional
};

const sumar = (a, b) => a + b;
console.log(sumar(2, 3)); // Salida: 5
```

Parámetros y argumentos

Parámetros: Son las variables que se definen en la declaración de la función.

Argumentos: Son los valores que se pasan a la función cuando se invoca.

```
function saludar(nombre, edad) { // nombre y edad son parámetros
  console.log(`Hola, ${nombre}. Tienes ${edad} años.`);
}

saludar("Juan", 30); // "Juan" y 30 son argumentos

Salida:
Hola, Juan. Tienes 30 años.
```

Valores por defecto en parámetros

Puedes asignar valores por defecto a los parámetros en caso de que no se pasen argumentos.

Ejemplo:

```
function saludar(nombre = "Invitado", edad = 18) {
  console.log(`Hola, ${nombre}. Tienes ${edad} años.`);
}

saludar(); // Salida: "Hola, Invitado. Tienes 18 años."
```

Funciones que devuelven valores

Una función puede devolver un valor usando la palabra clave return. Si no se usa return, la función devuelve undefined.

Ejemplo:

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

let resultado = multiplicar(3, 4);
console.log(resultado); // Salida: 12
```

Funciones como ciudadanos de primera clase

En JavaScript, las funciones son ciudadanos de primera clase, lo que significa que pueden ser:

Asignadas a variables.

Pasadas como argumentos a otras funciones.

Devueltas por otras funciones.

Ejemplo de función como argumento:

```
function ejecutarFuncion(funcion) {
  funcion();
}

ejecutarFuncion(() => console.log("Hola")); // Salida: "Hola"
```

Ejemplo de función que devuelve otra función:

```
function crearSaludo(mensaje) {
  return function(nombre) {
    return `${mensaje}, ${nombre}`;
  };
}

const saludar = crearSaludo("Hola");
console.log(saludar("Juan")); // Salida: "Hola, Juan"
```

Funciones anónimas

Una función anónima es una función sin nombre. Se usa comúnmente como argumento de otra función o en expresiones de función.

Ejemplo:

```
setTimeout(function() {
  console.log("Esto es una función anónima");
}, 1000);
```

Funciones recursivas

Una función recursiva es una función que se llama a sí misma. Es útil para resolver problemas que pueden dividirse en subproblemas más pequeños.

Ejemplo (factorial):

```
function factorial(n) {
  if (n === 0 || n === 1) {
    return 1;
  }
  return n * factorial(n - 1);
```

```
}

console.log(factorial(5)); // Salida: 120
```

Funciones de orden superior

Una función de orden superior es una función que recibe otra función como argumento o devuelve una función.

Ejemplo (map):

```
const numeros = [1, 2, 3, 4];
const duplicados = numeros.map((numero) => numero * 2);

console.log(duplicados); // Salida: [2, 4, 6, 8]
```

Closures

Un closure es una función que "recuerda" el entorno en el que fue creada, incluso después de que ese entorno haya desaparecido.

Ejemplo:

```
function crearContador() {
  let contador = 0;
  return function() {
    contador++;
    return contador;
  };
}

const contador = crearContador();
console.log(contador()); // Salida: 1
console.log(contador()); // Salida: 2
```

Funciones generadoras (Generators)

Las funciones generadoras son funciones especiales que pueden pausar y reanudar su ejecución. Se definen con function* y usan yield para devolver valores.

Ejemplo:

```
function* generador() {
  yield 1;
```

```
  yield 2;
  yield 3;
}

const gen = generador();
console.log(gen.next().value); // Salida: 1
console.log(gen.next().value); // Salida: 2
console.log(gen.next().value); // Salida: 3
```

Funciones asíncronas (Async/Await)

Las funciones asíncronas permiten trabajar con código asíncrono de manera más legible. Se definen con async y usan await para esperar a que una promesa se resuelva.

Ejemplo:

```
async function obtenerDatos() {
  const respuesta = await fetch("https://api.ejemplo.com/datos");
  const datos = await respuesta.json();
  return datos;
}

obtenerDatos().then(datos => console.log(datos));
```

Arrow functions

Son una forma más corta de escribir funciones, introducidas en ES6. Por ejemplo:

```
const saludar = (nombre) => {
    console.log("Hola, " + nombre);
};
saludar("Juan"); // "Hola, Juan"
```

this en arrow functions

Uno de los aspectos más importantes de las arrow functions es cómo manejan el valor de this. A diferencia de las funciones tradicionales, las arrow functions no tienen su propio this. En su lugar, heredan el valor de this del contexto en el que fueron definidas (esto se conoce como lexical scoping).

Ejemplo comparativo:

```
// Función tradicional
const obj1 = {
  valor: 42,
  metodo: function() {
```

```
    console.log(this.valor); // `this` se refiere a obj1
  }
};

obj1.metodo(); // Salida: 42

// Arrow function
const obj2 = {
  valor: 42,
  metodo: () => {
    console.log(this.valor); // `this` no se refiere a obj2, sino al contexto
exterior
  }
};

obj2.metodo(); // Salida: undefined (si el contexto exterior no tiene `valor`)
```

En el caso de obj2, this no se refiere al objeto obj2, sino al contexto en el que se definió la arrow function (por ejemplo, el ámbito global o el de otra función).

Arrow functions y el objeto arguments

Las arrow functions no tienen su propio objeto arguments. Si intentas acceder a arguments dentro de una arrow function, se referirá al objeto arguments del contexto exterior (si existe).

Ejemplo:

```
function funcionTradicional() {
  console.log(arguments); // Muestra los argumentos pasados a la función
}

const arrowFunction = () => {
  console.log(arguments); // ReferenceError: arguments is not defined
};

funcionTradicional(1, 2, 3); // Salida: [1, 2, 3]
arrowFunction(1, 2, 3); // Error
```

Si necesitas acceder a los argumentos en una arrow function, puedes usar el rest parameter (...args):

```
const arrowFunction = (...args) => {
  console.log(args); // Muestra los argumentos como un array
};

arrowFunction(1, 2, 3); // Salida: [1, 2, 3]
```

Arrow functions como métodos de objetos

Aunque puedes usar arrow functions como métodos de objetos, no es recomendable debido a su manejo de this. Como mencionamos antes, las arrow functions no tienen su propio this, por lo que no pueden acceder a las propiedades del objeto correctamente.

Ejemplo:

```
const obj = {
  valor: 42,
  metodoTradicional: function() {
    console.log(this.valor); // `this` se refiere a obj
  },
  metodoArrow: () => {
    console.log(this.valor); // `this` no se refiere a obj
  }
};

obj.metodoTradicional(); // Salida: 42
obj.metodoArrow(); // Salida: undefined
```

Arrow functions y constructores

Las arrow functions no pueden ser usadas como constructores. Si intentas usar new con una arrow function, obtendrás un error.

Ejemplo:

```
const FuncionTradicional = function() {
  this.valor = 42;
};

const ArrowFunction = () => {
  this.valor = 42; // Error: Arrow functions no pueden ser constructores
};

const obj1 = new FuncionTradicional(); // Funciona
console.log(obj1.valor); // Salida: 42

const obj2 = new ArrowFunction(); // TypeError: ArrowFunction is not a constructor
```

Arrow functions y métodos de prototipos

Las arrow functions no son adecuadas para definir métodos en prototipos debido a su manejo de this. Si defines un método de prototipo con una arrow function, this no se referirá a la instancia del objeto.

Ejemplo:

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

```
}

// Método tradicional
Persona.prototype.saludarTradicional = function() {
  console.log(`Hola, soy ${this.nombre}`);
};

// Método con arrow function
Persona.prototype.saludarArrow = () => {
  console.log(`Hola, soy ${this.nombre}`); // `this` no se refiere a la instancia
};

const persona = new Persona("Juan");
persona.saludarTradicional(); // Salida: "Hola, soy Juan"
persona.saludarArrow(); // Salida: "Hola, soy undefined"
```

Arrow functions y call, apply, bind

Las arrow functions no pueden cambiar su valor de this usando call, apply o bind. Como this está ligado al contexto léxico, estos métodos no tienen efecto.

Ejemplo:

```
const obj = { valor: 42 };

const funcionTradicional = function() {
  console.log(this.valor);
};

const arrowFunction = () => {
  console.log(this.valor);
};

funcionTradicional.call(obj); // Salida: 42
arrowFunction.call(obj); // Salida: undefined (no cambia `this`)
```

Arrow functions y super

Las arrow functions no tienen su propio super. Si intentas usar super dentro de una arrow function, se referirá al super del contexto exterior.

Ejemplo:

```
class Padre {
  constructor() {
    this.valor = 42;
  }
}
```

```
class Hijo extends Padre {
  metodoTradicional() {
    console.log(super.valor); // Accede al `valor` del padre
  }

  metodoArrow = () => {
    console.log(super.valor); // Error: `super` no está definido
  };
}

const hijo = new Hijo();
hijo.metodoTradicional(); // Salida: 42
hijo.metodoArrow(); // Error
```

Arrow functions y yield

Las arrow functions no pueden ser usadas como generadoras. No puedes usar yield dentro de una arrow function.

Ejemplo:

```
function* generadoraTradicional() {
  yield 1;
  yield 2;
}

const generadoraArrow = () => {
  yield 1; // SyntaxError: Unexpected token 'yield'
};
```

Closures

Es una función que recuerda el entorno en el que fue creada, incluso si ese entorno ya no existe. Por ejemplo:

```
function crearContador() {
    let contador = 0;
    return function() {
        contador++;
        return contador;
    };
}
const contador = crearContador();
console.log(contador()); // 1
console.log(contador()); // 2
```

Un closure es una función que "recuerda" el entorno en el que fue creada, incluso después de que ese entorno haya desaparecido. En otras palabras, un closure permite que una función acceda a variables de su

ámbito léxico (el ámbito en el que fue definida) incluso cuando se ejecuta fuera de ese ámbito.

Ejemplo básico:

```
function crearContador() {
  let contador = 0; // Variable en el ámbito de crearContador

  return function() {
    contador++; // Accede a la variable contador
    return contador;
  };
}

const contador = crearContador();
console.log(contador()); // Salida: 1
console.log(contador()); // Salida: 2
```

En este ejemplo:

La función crearContador define una variable contador.

Devuelve una función que incrementa y devuelve contador.

Aunque crearContador ya terminó de ejecutarse, la función devuelta recuerda la variable contador y puede seguir accediendo a ella. Esto es un closure.

Los closures funcionan gracias a la cadena de ámbitos (scope chain) en JavaScript. Cuando una función se define, captura una referencia a su ámbito léxico (las variables disponibles en el lugar donde se definió). Esta referencia se mantiene incluso si la función se ejecuta en un ámbito diferente.

Ejemplo:

```
function exterior() {
  let mensaje = "Hola";

  function interior() {
    console.log(mensaje); // Accede a la variable mensaje del ámbito exterior
  }

  return interior;
}

const funcionInterior = exterior();
funcionInterior(); // Salida: "Hola"
```

Aquí, funcionInterior es un closure porque recuerda el ámbito de exterior y puede acceder a la variable mensaje.

- Closures y bucles Un error común es crear closures dentro de bucles sin tener en cuenta cómo se capturan las variables. Por ejemplo:

```
for (var i = 1; i <= 3; i++) {
  setTimeout(function() {
    console.log(i); // ¿Qué imprime?
  }, 1000);
}
Salida:
4
4
4
```

Esto ocurre porque var no tiene ámbito de bloque, y todas las funciones setTimeout comparten la misma variable i. Para solucionarlo, puedes usar let (que tiene ámbito de bloque) o crear un nuevo ámbito con una IIFE (Immediately Invoked Function Expression):

```
for (let i = 1; i <= 3; i++) {
  setTimeout(function() {
    console.log(i); // Ahora imprime 1, 2, 3
  }, 1000);
}
```

- Closures y memoria Los closures pueden causar fugas de memoria si no se usan correctamente, ya que mantienen referencias a variables que podrían no ser necesarias. Por ejemplo:

```
function crearClosure() {
  let datosGrandes = new Array(1000000).fill("Datos"); // Un array grande
  return function() {
    console.log("Closure ejecutado");
  };
}

const closure = crearClosure();
```

// Aunque no usamos datosGrandes, el closure mantiene una referencia a él Para evitar esto, puedes liberar manualmente las referencias cuando ya no sean necesarias:

```
closure = null; // Libera la referencia
```

- Closures y módulos Los closures son la base de los módulos en JavaScript. Puedes usar closures para crear patrones de módulos que encapsulan variables y funciones privadas.

Ejemplo:

```
const modulo = (function() {
  let contador = 0; // Variable privada

  return {
    incrementar: function() {
      contador++;
    },
    obtenerContador: function() {
      return contador;
    }
  };
})();

modulo.incrementar();
console.log(modulo.obtenerContador()); // Salida: 1
```

Aquí, contador es una variable privada a la que solo se puede acceder a través de los métodos incrementar y obtenerContador.

Los closures son extremadamente útiles en la programación. Aquí te explico algunos de sus usos más comunes:

- Manejo de estado privado Los closures permiten crear variables privadas que solo pueden ser accedidas o modificadas por funciones específicas.

Ejemplo:

```
function crearContador() {
  let contador = 0; // Variable privada

  return {
    incrementar: function() {
      contador++;
    },
    obtenerContador: function() {
      return contador;
    }
  };
}

const contador = crearContador();
contador.incrementar();
console.log(contador.obtenerContador()); // Salida: 1
```

- Funciones de orden superior Las funciones que devuelven otras funciones (funciones de orden superior) a menudo usan closures para recordar el estado.

Ejemplo:

```
function crearMultiplicador(factor) {
  return function(numero) {
    return numero * factor;
  };
}

const duplicar = crearMultiplicador(2);
console.log(duplicar(5)); // Salida: 10
```

- Callbacks y eventos Los closures son fundamentales en el manejo de callbacks y eventos, ya que permiten que las funciones recuerden el contexto en el que fueron creadas.

Ejemplo:

```
function configurarBoton() {
  let contador = 0;

  document.getElementById("miBoton").addEventListener("click", function() {
    contador++;
    console.log(`Botón clickeado ${contador} veces`);
  });
}

configurarBoton();
```

- Currying El currying es una técnica en la que una función con múltiples argumentos se transforma en una secuencia de funciones que toman un solo argumento. Los closures son esenciales para implementar currying.

Ejemplo:

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

const sumar5 = sumar(5);
console.log(sumar5(3)); // Salida: 8
```

- Memoización La memoización es una técnica de optimización que almacena los resultados de funciones costosas para evitar cálculos repetidos. Los closures son útiles para implementar memoización.

Ejemplo:

```
function memoizar(funcion) {
  const cache = {};

  return function(...args) {
    const clave = JSON.stringify(args);
    if (cache[clave]) {
      return cache[clave];
    }
    const resultado = funcion(...args);
    cache[clave] = resultado;
    return resultado;
  };
}

const factorial = memoizar(function(n) {
  if (n === 0 || n === 1) return 1;
  return n * factorial(n - 1);
});

console.log(factorial(5)); // Salida: 120
```

Scope

Se refiere al alcance de una variable. Las variables pueden ser globales (accesibles en todo el código) o locales (accesibles solo dentro de una función o bloque).

4. Manejo de objetos y arrays

Object.keys(): Devuelve un array con las claves de un objeto. Por ejemplo:

```
const persona = { nombre: "Juan", edad: 30 };
console.log(Object.keys(persona)); // ["nombre", "edad"]
```

Object.values(): Devuelve un array con los valores de un objeto. Por ejemplo:

```
const persona = { nombre: "Juan", edad: 30 };
console.log(Object.values(persona)); // ["Juan", 30]
```

Spread/Rest operator:

Spread: Permite expandir elementos de un array o propiedades de un objeto. Por ejemplo:

```
const numeros = [1, 2, 3];
const nuevosNumeros = [...numeros, 4, 5]; // [1, 2, 3, 4, 5]
```

Rest: Permite capturar múltiples argumentos en una función como un array. Por ejemplo:

```
function sumar(...numeros) {
    return numeros.reduce((acc, num) => acc + num, 0);
}
console.log(sumar(1, 2, 3)); // 6
```

1. Object.keys() ¿Qué es Object.keys()? Object.keys() es un método que devuelve un array con las claves (propiedades enumerables) de un objeto.

Sintaxis: javascript

Object.keys(objeto); Ejemplo básico: javascript

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

const claves = Object.keys(persona); console.log(claves); // Salida: ["nombre", "edad", "profesion"] Usos comunes: Recorrer las propiedades de un objeto:

javascript

claves.forEach(clave => { console.log(${clave}: ${persona[clave]}); }); Verificar si un objeto tiene propiedades:

javascript

if (Object.keys(persona).length === 0) { console.log("El objeto está vacío"); } 2. Object.values() ¿Qué es Object.values()? Object.values() es un método que devuelve un array con los valores de las propiedades enumerables de un objeto.

Sintaxis: javascript

Object.values(objeto); Ejemplo básico: javascript

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

const valores = Object.values(persona); console.log(valores); // Salida: ["Juan", 30, "Desarrollador"] Usos comunes: Recorrer los valores de un objeto:

javascript

valores.forEach(valor => { console.log(valor); }); Sumar valores numéricos de un objeto:

javascript

const total = Object.values(persona).reduce((acc, valor) => { if (typeof valor === "number") { return acc + valor; } return acc; }, 0); console.log(total); // Salida: 30 3. Operador Spread (...) ¿Qué es el operador spread? El operador spread (...) permite "expandir" un iterable (como un array o un objeto) en lugares donde se esperan múltiples elementos o pares clave-valor.

Sintaxis: javascript

...iterable; Ejemplos: a) Con arrays: Copiar un array:

javascript

const numeros = [1, 2, 3]; const copiaNumeros = [...numeros]; console.log(copiaNumeros); // Salida: [1, 2, 3] Concatenar arrays:

javascript

const masNumeros = [4, 5, 6]; const todosLosNumeros = [...numeros, ...masNumeros]; console.log(todosLosNumeros); // Salida: [1, 2, 3, 4, 5, 6] b) Con objetos: Copiar un objeto:

javascript

const persona = { nombre: "Juan", edad: 30 }; const copiaPersona = { ...persona }; console.log(copiaPersona); // Salida: { nombre: "Juan", edad: 30 } Combinar objetos:

javascript

const detalles = { profesion: "Desarrollador" }; const personaCompleta = { ...persona, ...detalles }; console.log(personaCompleta); // Salida: { nombre: "Juan", edad: 30, profesion: "Desarrollador" } c) En llamadas a funciones: Pasar elementos de un array como argumentos:

javascript

function sumar(a, b, c) { return a + b + c; } const numeros = [1, 2, 3]; console.log(sumar(...numeros)); // Salida: 6 4. Operador Rest (...) ¿Qué es el operador rest? El operador rest (...) permite representar un número indefinido de argumentos como un array. Se usa en la definición de funciones para capturar todos los argumentos restantes.

Sintaxis: javascript

function nombreFuncion(...args) { // args es un array con todos los argumentos } Ejemplos: a) En funciones: Capturar todos los argumentos:

javascript

function sumar(...numeros) { return numeros.reduce((acc, num) => acc + num, 0); } console.log(sumar(1, 2, 3, 4)); // Salida: 10 b) En desestructuración: Capturar elementos restantes de un array:

javascript

const [primero, segundo, ...resto] = [1, 2, 3, 4, 5]; console.log(primero); // Salida: 1 console.log(segundo); // Salida: 2 console.log(resto); // Salida: [3, 4, 5] Capturar propiedades restantes de un objeto:

javascript

const persona = { nombre: "Juan", edad: 30, profesion: "Desarrollador" }; const { nombre, ...detalles } = persona; console.log(nombre); // Salida: "Juan" console.log(detalles); // Salida: { edad: 30, profesion: "Desarrollador" } 5. Conceptos avanzados a) Object.keys() y Object.values() con objetos complejos: Puedes usar estos métodos con objetos anidados o que contienen funciones.

javascript

const objetoComplejo = { nombre: "Juan", edad: 30, saludar() { console.log("Hola"); } }; console.log(Object.keys(objetoComplejo)); // Salida: ["nombre", "edad", "saludar"] console.log(Object.values(objetoComplejo)); // Salida: ["Juan", 30, [Function: saludar]] b) Spread y rest con objetos dinámicos: Puedes combinar spread y rest para manipular objetos dinámicamente.

javascript

const persona = { nombre: "Juan", edad: 30 }; const nuevosDatos = { profesion: "Desarrollador", ciudad: "Madrid" }; const personaActualizada = { ...persona, ...nuevosDatos }; console.log(personaActualizada); // Salida: { nombre: "Juan", edad: 30, profesion: "Desarrollador", ciudad: "Madrid" } c) Spread y rest en funciones variádicas: Las funciones variádicas son aquellas que aceptan un número variable de argumentos. El operador rest es perfecto para esto.

javascript

function concatenar(...strings) { return strings.join(" "); } console.log(concatenar("Hola", "mundo", "!")); // Salida: "Hola mundo !" 6. Usos comunes a) Manipulación de objetos y arrays: Copiar, combinar o extraer propiedades de objetos y arrays.

Ejemplo:

javascript

const usuario = { nombre: "Ana", edad: 25 }; const usuarioConEmail = { ...usuario, email: "ana@example.com" }; console.log(usuarioConEmail); b) Funciones flexibles: Crear funciones que acepten un número variable de argumentos.

Ejemplo:

javascript

function logear(...mensajes) { mensajes.forEach(mensaje => console.log(mensaje)); } logear("Error", "Código 404", "Recurso no encontrado"); c) Desestructuración avanzada: Extraer valores específicos de arrays u objetos y capturar el resto.

Ejemplo:

javascript

const [primero, ...resto] = [1, 2, 3, 4]; console.log(primero); // 1 console.log(resto); // [2, 3, 4]

arrays

JavaScript proporciona una gran cantidad de métodos para trabajar con arrays. Aquí te explico algunos de los más comunes:

- **push y pop**

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

pop: Elimina y devuelve el último elemento del array.

Ejemplo:

```
let frutas = ["Manzana", "Banana"];
frutas.push("Cereza"); // Añade "Cereza" al final
console.log(frutas); // Salida: ["Manzana", "Banana", "Cereza"]

let ultimaFruta = frutas.pop(); // Elimina "Cereza"
console.log(ultimaFruta); // Salida: "Cereza"
```

- **shift y unshift**

shift: Elimina y devuelve el primer elemento del array.

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

Ejemplo:

```
let frutas = ["Manzana", "Banana"];
frutas.unshift("Cereza"); // Añade "Cereza" al principio
console.log(frutas); // Salida: ["Cereza", "Manzana", "Banana"]

let primeraFruta = frutas.shift(); // Elimina "Cereza"
console.log(primeraFruta); // Salida: "Cereza"
```

- **slice y splice**

slice: Devuelve una copia de una parte del array.

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

Ejemplo:

```
let frutas = ["Manzana", "Banana", "Cereza", "Durazno"];
let subArray = frutas.slice(1, 3); // Devuelve ["Banana", "Cereza"]
console.log(subArray);

frutas.splice(1, 2, "Kiwi", "Mango"); // Elimina 2 elementos desde el índice 1 y
```

```
añade "Kiwi" y "Mango"
console.log(frutas); // Salida: ["Manzana", "Kiwi", "Mango", "Durazno"]
```

- **concat**

Combina dos o más arrays.

Ejemplo:

```
let frutas1 = ["Manzana", "Banana"];
let frutas2 = ["Cereza", "Durazno"];
let todasLasFrutas = frutas1.concat(frutas2);
console.log(todasLasFrutas); // Salida: ["Manzana", "Banana", "Cereza", "Durazno"]
```

- **forEach**

Ejecuta una función para cada elemento del array.

Ejemplo:

```
let numeros = [1, 2, 3];
numeros.forEach(function(numero) {
  console.log(numero);
});
// Salida:
// 1
// 2
// 3
```

- **map**

Crea un nuevo array aplicando una función a cada elemento del array original.

Ejemplo:

```
let numeros = [1, 2, 3];
let cuadrados = numeros.map(function(numero) {
  return numero * numero;
});
console.log(cuadrados); // Salida: [1, 4, 9]
```

- **filter**

Crea un nuevo array con los elementos que cumplen una condición.

Ejemplo:

```
let numeros = [1, 2, 3, 4, 5];
let pares = numeros.filter(function(numero) {
  return numero % 2 === 0;
});
console.log(pares); // Salida: [2, 4]
```

- **reduce**

Reduce el array a un solo valor aplicando una función acumuladora.

Ejemplo:

```
let numeros = [1, 2, 3, 4];
let suma = numeros.reduce(function(acumulador, numero) {
  return acumulador + numero;
}, 0);
console.log(suma); // Salida: 10
```

Conceptos avanzados de arrays

- Arrays multidimensionales

Un array puede contener otros arrays, lo que permite crear estructuras multidimensionales.

Ejemplo:

```
let matriz = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]
];
console.log(matriz[1][2]); // Salida: 6
```

- Arrays dispersos

Un array disperso es un array en el que no todos los índices tienen valores asignados.

Ejemplo:

```
let disperso = [];
```

```
disperso[5] = "Hola";
console.log(disperso); // Salida: [ <5 empty items>, "Hola" ]
```

- Arrays y objetos iterables

Los arrays son objetos iterables, lo que significa que puedes usar bucles for...of para recorrerlos.

Ejemplo:

```
let frutas = ["Manzana", "Banana", "Cereza"];
for (let fruta of frutas) {
  console.log(fruta);
}
// Salida:
// Manzana
// Banana
// Cereza
```

- Desestructuración de arrays

La desestructuración permite extraer valores de un array en variables individuales.

Ejemplo:

```
let frutas = ["Manzana", "Banana", "Cereza"];
let [primera, segunda] = frutas;
console.log(primera); // Salida: "Manzana"
console.log(segunda); // Salida: "Banana"
```

- Operador spread (...)

El operador spread permite expandir un array en lugares donde se esperan múltiples elementos.

Ejemplo:

```
let frutas = ["Manzana", "Banana"];
let masFrutas = ["Cereza", ...frutas, "Durazno"];
console.log(masFrutas); // Salida: ["Cereza", "Manzana", "Banana", "Durazno"]
```

Usos comunes de arrays

Los arrays son fundamentales en la programación y se utilizan en una gran variedad de situaciones:

- Almacenamiento de datos

Los arrays son ideales para almacenar colecciones de datos, como listas de usuarios, productos, etc.

Ejemplo:

```
let usuarios = [
  { nombre: "Juan", edad: 30 },
  { nombre: "Ana", edad: 25 }
];
```

- Manipulación de datos

Puedes usar métodos como map, filter, y reduce para transformar y manipular datos.

Ejemplo:

```
let edades = usuarios.map(usuario => usuario.edad);
console.log(edades); // Salida: [30, 25]
```

- Iteración

Los arrays son perfectos para iterar sobre una colección de elementos.

Ejemplo:

```
usuarios.forEach(usuario => {
  console.log(`${usuario.nombre} tiene ${usuario.edad} años`);
});
```

- Implementación de estructuras de datos

Los arrays se utilizan para implementar otras estructuras de datos como pilas, colas y listas enlazadas.

Ejemplo de pila:

```
let pila = [];
pila.push(1); // Añadir
pila.push(2);
console.log(pila.pop()); // Eliminar y obtener el último elemento
```

- Algoritmos de ordenación y búsqueda

Los arrays son la base para algoritmos como la búsqueda binaria o la ordenación rápida (quicksort).

Ejemplo de ordenación:

```
let numeros = [3, 1, 4, 1, 5, 9];
numeros.sort((a, b) => a - b);
console.log(numeros); // Salida: [1, 1, 3, 4, 5, 9]
```

cortocircuito

- Acceso seguro a propiedades anidadas:

Supongamos que tienes un objeto usuario que puede no estar definido o no tener ciertas propiedades. Usamos && para evitar errores:

```
let usuario = { perfil: { nombre: "Carlos" } };

let nombreUsuario = usuario && usuario.perfil && usuario.perfil.nombre;
console.log(nombreUsuario); // "Carlos"
```

Con encadenamiento opcional (disponible en ES2020), esto se simplifica aún más:

```
let nombreUsuario = usuario?.perfil?.nombre;
console.log(nombreUsuario); // "Carlos"
```

- Asignación de valores por defecto:

Si tienes una variable que puede ser null, undefined o una cadena vacía, puedes usar || para asignar un valor por defecto:

```
let nombre = "";
let nombreAMostrar = nombre || "Invitado";
console.log(nombreAMostrar); // "Invitado"
```

- Ejecución condicional de funciones:

Puedes usar && para ejecutar una función solo si una condición es verdadera:

```
let usuario = { nombre: "Ana" };

usuario && usuario.nombre && console.log("Hola, " + usuario.nombre);
```

Aquí, console.log solo se ejecuta si usuario y usuario.nombre son truthy.

Ejemplos adicionales

- Uso de && para renderizado condicional (en frameworks como React):

```
let mostrarMensaje = true;

return (
  <div>
    {mostrarMensaje && <p>Este es un mensaje importante.</p>}
  </div>
);
```

Si mostrarMensaje es true, se renderiza el mensaje. Si es false, no se renderiza nada.

- Uso de || para valores por defecto en funciones:

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

saludar(); // "Hola, Invitado"
saludar("Ana"); // "Hola, Ana"
```

- Resumen

&&: Útil para verificar múltiples condiciones y acceder a propiedades de manera segura.

||: Útil para asignar valores por defecto cuando una variable es falsy.

Objetos y Clases (class)

Objetos en JavaScript

En JavaScript, los objetos son colecciones de pares clave-valor, donde las claves son strings (o símbolos) y los valores pueden ser cualquier tipo de dato, incluyendo funciones (métodos).

Ejemplo básico:

```
const persona = {
  nombre: "Juan",
  edad: 30,
```

```
  saludar() {
    console.log(`Hola, mi nombre es ${this.nombre}`);
  }
};

persona.saludar(); // Salida: "Hola, mi nombre es Juan"
```

Acceso a propiedades

Puedes acceder a las propiedades de un objeto de dos maneras:

Notación de punto:

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

Notación de corchetes:

```
console.log(persona["edad"]); // 30
let propiedad = "nombre";
console.log(persona[propiedad]); // "Juan"
```

Modificación y adición de propiedades Puedes modificar o agregar propiedades dinámicamente:

```
persona.edad = 31; // Modificar
persona.profesion = "Desarrollador"; // Agregar
console.log(persona); // { nombre: "Juan", edad: 31, saludar: [Function],
profesion: "Desarrollador" }
```

Eliminación de propiedades

Usa delete para eliminar una propiedad:

```
delete persona.edad;
console.log(persona.edad); // undefined
```

Métodos en objetos

Los métodos son funciones que están asociadas a un objeto. Puedes definirlos directamente en el objeto:

```
let coche = {
    marca: "Toyota",
    arrancar: function() {
```

```
        console.log("El coche está arrancando...");
    }
};
coche.arrancar(); // "El coche está arrancando..."
```

this en objetos

La palabra clave this hace referencia al objeto que está ejecutando el método. Es útil para acceder a las propiedades del objeto dentro de un método:

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

Objetos anidados

Los objetos pueden contener otros objetos:

```
let empresa = {
    nombre: "Tech Corp",
    direccion: {
        calle: "Calle Falsa 123",
        ciudad: "Madrid"
    }
};
console.log(empresa.direccion.ciudad); // "Madrid"
```

Recorrer propiedades de un objeto

Puedes usar un bucle for...in para recorrer las propiedades de un objeto:

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

Métodos estáticos de Object

JavaScript proporciona métodos útiles para trabajar con objetos:

Object.keys(obj): Devuelve un array con las claves del objeto.

```
console.log(Object.keys(persona)); // ["nombre", "edad", "saludar"]
Object.values(obj): Devuelve un array con los valores del objeto.
```

```
console.log(Object.values(persona)); // ["Juan", 30, [Function]]
Object.entries(obj): Devuelve un array de arrays, donde cada subarray es un par
clave-valor.
```

```
console.log(Object.entries(persona)); // [["nombre", "Juan"], ["edad", 30],
["saludar", [Function]]]
```

Prototipos y herencia

En JavaScript, los objetos pueden heredar propiedades y métodos de otros objetos a través de la cadena de prototipos (prototype chain). Esto es la base de la programación orientada a objetos en JavaScript.

Ejemplo:

```
let animal = {
    sonido: function() {
        console.log("Haciendo sonidos...");
    }
};

let perro = Object.create(animal);
perro.ladrar = function() {
    console.log("Guau guau!");
};

perro.sonido(); // "Haciendo sonidos..."
perro.ladrar(); // "Guau guau!"
```

Clases (ES6)

Las clases en JavaScript son una forma más moderna y clara de trabajar con objetos y herencia. Son una "sintaxis azucarada" sobre los prototipos.

Ejemplo:

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

```
    saludar() {
        console.log("Hola, soy " + this.nombre);
    }
}

let juan = new Persona("Juan", 30);
juan.saludar(); // "Hola, soy Juan"
```

Getters y Setters

Los getters y setters permiten definir cómo se accede y modifica una propiedad:

```
let rectangulo = {
    ancho: 10,
    alto: 5,
    get area() {
        return this.ancho * this.alto;
    },
    set nuevoAncho(valor) {
        this.ancho = valor;
    }
};

console.log(rectangulo.area); // 50
rectangulo.nuevoAncho = 20;
console.log(rectangulo.area); // 100
```

Objetos inmutables

Puedes crear objetos inmutables usando Object.freeze():

```
let objeto = { nombre: "Juan" };
Object.freeze(objeto);
objeto.nombre = "Ana"; // No tendrá efecto
console.log(objeto.nombre); // "Juan"
```

Desestructuración de objetos

La desestructuración permite extraer propiedades de un objeto en variables:

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

Métodos avanzados

Object.assign(): Copia propiedades de un objeto a otro.

```
let destino = { a: 1 };
let origen = { b: 2 };
Object.assign(destino, origen);
console.log(destino); // { a: 1, b: 2 }
```

Object.defineProperty(): Define una nueva propiedad con características específicas (como si es enumerable, configurable, etc.).

```
let obj = {};
Object.defineProperty(obj, "propiedad", {
    value: 42,
    writable: false
});
console.log(obj.propiedad); // 42
```

obj.propiedad = 100; // No tendrá efecto

Symbols como claves

Los símbolos son un tipo de dato único que se puede usar como clave en objetos:

```
let id = Symbol("id");
let usuario = {
    [id]: 123,
    nombre: "Pedro"
};
console.log(usuario[id]); // 123
```

Proxies

Los proxies permiten interceptar y redefinir operaciones fundamentales en objetos:

```
let objetivo = { nombre: "Juan" };
let manejador = {
    get: function(objeto, propiedad) {
        return propiedad in objeto ? objeto[propiedad] : "No existe";
    }
};
let proxy = new Proxy(objetivo, manejador);
console.log(proxy.nombre); // "Juan"
console.log(proxy.edad); // "No existe"
```

JSON y objetos

JavaScript permite convertir objetos a JSON y viceversa:

```
let objeto = { nombre: "Juan", edad: 30 };
let json = JSON.stringify(objeto); // Convertir a JSON
console.log(json); // {"nombre":"Juan","edad":30}

let nuevoObjeto = JSON.parse(json); // Convertir de JSON a objeto
console.log(nuevoObjeto); // { nombre: "Juan", edad: 30 }
```

Patrones de diseño con objetos

Factory Pattern: Crear objetos sin usar new.

Singleton Pattern: Garantizar una única instancia de un objeto.

Module Pattern: Encapsular funcionalidades en un objeto.

Ejemplo de Singleton:

```
let Singleton = (function() {
    let instancia;

    function crearInstancia() {
        return { nombre: "Instancia única" };
    }

    return {
        obtenerInstancia: function() {
            if (!instancia) {
                instancia = crearInstancia();
            }
            return instancia;
        }
    };
})();

let instancia1 = Singleton.obtenerInstancia();
let instancia2 = Singleton.obtenerInstancia();
console.log(instancia1 === instancia2); // true
```

Buenas prácticas

Usa const para objetos que no cambian.

Evita modificar objetos directamente; prefiere la inmutabilidad.

Usa clases para estructuras complejas.

Aprovecha la desestructuración para mejorar la legibilidad.

Clases en JavaScript

Las clases son una forma de crear objetos con una estructura y comportamiento definidos. Introducidas en ES6, las clases son una sintaxis más clara y orientada a objetos.

Sintaxis básica:

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

  saludar() {
    console.log(`Hola, mi nombre es ${this.nombre}`);
  }
}

const juan = new Persona("Juan", 30);
juan.saludar(); // Salida: "Hola, mi nombre es Juan"
```

constructor: Es un método especial que se ejecuta cuando se crea una instancia de la clase. Se usa para inicializar propiedades.

Métodos: Funciones definidas dentro de la clase.

Getters y Setters

Los getters y setters son métodos especiales que permiten controlar el acceso a las propiedades de un objeto.

Ejemplo:

```
class Persona {
  constructor(nombre, edad) {
    this._nombre = nombre;
    this._edad = edad;
  }

  // Getter para nombre
  get nombre() {
    return this._nombre.toUpperCase();
  }

  // Setter para nombre
  set nombre(nuevoNombre) {
```

```
    this._nombre = nuevoNombre.trim();
  }
}

const juan = new Persona("Juan", 30);
console.log(juan.nombre); // Salida: "JUAN" (usa el getter)
juan.nombre = "  Ana  "; // Usa el setter
console.log(juan.nombre); // Salida: "ANA"
```

Uso avanzado: Puedes usar getters y setters para validar datos, formatear valores o realizar acciones adicionales cuando se accede o modifica una propiedad.

Propiedades y métodos computados

Puedes definir propiedades y métodos cuyos nombres se calculan dinámicamente.

Ejemplo:

```
const metodo = "saludar";

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

  [metodo]() {
    console.log(`Hola, mi nombre es ${this.nombre}`);
  }
}

const juan = new Persona("Juan");
juan.saludar(); // Salida: "Hola, mi nombre es Juan"
```

Uso avanzado: Útil para crear métodos o propiedades dinámicos basados en condiciones o configuraciones.

Mixins

Los mixins son una forma de agregar funcionalidades a una clase sin usar herencia.

Ejemplo:

```
const SaludarMixin = {
  saludar() {
    console.log(`Hola, mi nombre es ${this.nombre}`);
  }
};

class Persona {
  constructor(nombre) {
```

```
    this.nombre = nombre;
  }
}

Object.assign(Persona.prototype, SaludarMixin);

const juan = new Persona("Juan");
juan.saludar(); // Salida: "Hola, mi nombre es Juan"
```

Uso avanzado: Permite reutilizar código en múltiples clases sin crear una jerarquía compleja.

Herencia y Polimorfismo (extends, super)

Herencia

La herencia permite crear una clase basada en otra clase existente. La clase hija hereda propiedades y métodos de la clase padre.

Ejemplo:

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

  hacerSonido() {
    console.log("Sonido genérico");
  }
}

class Perro extends Animal {
  constructor(nombre, raza) {
    super(nombre); // Llama al constructor de la clase padre
    this.raza = raza;
  }

  hacerSonido() {
    console.log("Guau guau");
  }
}

const miPerro = new Perro("Rex", "Labrador");
miPerro.hacerSonido(); // Salida: "Guau guau"
```

extends: Indica que una clase hereda de otra.

super: Llama al constructor o métodos de la clase padre.

Polimorfismo

El polimorfismo permite que una clase hija sobrescriba un método de la clase padre para proporcionar una implementación específica.

Ejemplo:

```
class Gato extends Animal {
  hacerSonido() {
    console.log("Miau miau");
  }
}

const miGato = new Gato("Mimi");
miGato.hacerSonido(); // Salida: "Miau miau"
```

Llamadas a super en métodos

Puedes usar super para llamar a métodos de la clase padre desde la clase hija.

Ejemplo:

```
class Animal {
  hacerSonido() {
    console.log("Sonido genérico");
  }
}

class Perro extends Animal {
  hacerSonido() {
    super.hacerSonido(); // Llama al método de la clase padre
    console.log("Guau guau");
  }
}

const miPerro = new Perro();
miPerro.hacerSonido();
// Salida:
// "Sonido genérico"
// "Guau guau"
```

Uso avanzado: Útil para extender el comportamiento de un método en lugar de reemplazarlo por completo.

Clases abstractas

En JavaScript, no hay soporte nativo para clases abstractas, pero puedes simularlas lanzando errores en métodos que deben ser implementados por las clases hijas.

Ejemplo:

```
class Animal {
  constructor() {
    if (new.target === Animal) {
      throw new Error("No se puede instanciar una clase abstracta");
    }
  }

  hacerSonido() {
    throw new Error("Método abstracto: debe ser implementado");
  }
}

class Perro extends Animal {
  hacerSonido() {
    console.log("Guau guau");
  }
}

// const animal = new Animal(); // Error: No se puede instanciar una clase
abstracta
const miPerro = new Perro();
miPerro.hacerSonido(); // Salida: "Guau guau"
```

Uso avanzado: Para definir una estructura común que las clases hijas deben seguir.

Encapsulación (private, public, #propiedadPrivada)

La encapsulación es un principio de la programación orientada a objetos que permite ocultar los detalles internos de una clase y exponer solo lo necesario.

Propiedades privadas (#propiedadPrivada)

En JavaScript, las propiedades privadas se definen usando el prefijo #. Estas propiedades solo son accesibles dentro de la clase.

Ejemplo:

```
class CuentaBancaria {
  #saldo; // Propiedad privada

  constructor(saldoInicial) {
    this.#saldo = saldoInicial;
  }

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

  obtenerSaldo() {
    return this.#saldo;
```

```
  }
}

const cuenta = new CuentaBancaria(1000);
cuenta.depositar(500);
console.log(cuenta.obtenerSaldo()); // Salida: 1500
// console.log(cuenta.#saldo); // Error: Propiedad privada no accesible
```

Métodos privados

Los métodos también pueden ser privados usando #.

Ejemplo:

```
class CuentaBancaria {
  #saldo;

  constructor(saldoInicial) {
    this.#saldo = saldoInicial;
  }

  #validarCantidad(cantidad) {
    return cantidad > 0;
  }

  depositar(cantidad) {
    if (this.#validarCantidad(cantidad)) {
      this.#saldo += cantidad;
    }
  }
}
```

Métodos Estáticos (static)

Los métodos estáticos son métodos que pertenecen a la clase en lugar de a una instancia de la clase. Se llaman directamente desde la clase, no desde un objeto.

Ejemplo:

```
class Matemáticas {
  static sumar(a, b) {
    return a + b;
  }

  static PI = 3.1416; // Propiedad estática
}
```

```
console.log(Matemáticas.sumar(2, 3)); // Salida: 5
console.log(Matemáticas.PI); // Salida: 3.1416
```

Uso común: Para funciones utilitarias o constantes relacionadas con la clase.

Propiedades privadas estáticas

Las propiedades privadas también pueden ser estáticas.

Ejemplo:

```
class Contador {
  static #contador = 0;

  static incrementar() {
    this.#contador++;
  }

  static obtenerContador() {
    return this.#contador;
  }
}

Contador.incrementar();
console.log(Contador.obtenerContador()); // Salida: 1
```

Uso avanzado: Para mantener un estado privado compartido entre todas las instancias de una clase.

Métodos privados estáticos

Los métodos privados también pueden ser estáticos.

Ejemplo:

```
class Matemáticas {
  static #validarNumero(numero) {
    return typeof numero === "number";
  }

  static sumar(a, b) {
    if (this.#validarNumero(a) && this.#validarNumero(b)) {
      return a + b;
    }
    throw new Error("Los argumentos deben ser números");
  }
}

console.log(Matemáticas.sumar(2, 3)); // Salida: 5
```

Uso avanzado: Para encapsular lógica interna en métodos estáticos.

Prototipos y Herencia Prototípica

Prototipos

En JavaScript, cada objeto tiene un prototipo, que es otro objeto del cual hereda propiedades y métodos. Esto forma una cadena de prototipos.

Ejemplo:

```
const animal = {
  hacerSonido() {
    console.log("Sonido genérico");
  }
};

const perro = Object.create(animal);
perro.hacerSonido(); // Salida: "Sonido genérico"
```

Object.create: Crea un nuevo objeto con un prototipo específico.

Herencia Prototípica

La herencia prototípica es el mecanismo por el cual un objeto hereda propiedades y métodos de otro objeto.

Ejemplo:

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

Animal.prototype.hacerSonido = function() {
  console.log("Sonido genérico");
};

function Perro(nombre, raza) {
  Animal.call(this, nombre); // Llama al constructor de Animal
  this.raza = raza;
}

Perro.prototype = Object.create(Animal.prototype); // Hereda de Animal
Perro.prototype.constructor = Perro; // Corrige el constructor

Perro.prototype.hacerSonido = function() {
  console.log("Guau guau");
};
```

```
const miPerro = new Perro("Rex", "Labrador");
miPerro.hacerSonido(); // Salida: "Guau guau"
```

prototype: Es el objeto desde el cual se heredan propiedades y métodos.

Object.create: Crea un nuevo objeto con un prototipo específico.

Herencia de métodos estáticos

Los métodos estáticos también se heredan.

Ejemplo:

```
class Animal {
  static describir() {
    return "Soy un animal";
  }
}

class Perro extends Animal {}

console.log(Perro.describir()); // Salida: "Soy un animal"
```

Uso avanzado: Para compartir funcionalidades estáticas entre clases relacionadas.

Acceso a propiedades estáticas desde métodos estáticos

Puedes acceder a propiedades estáticas dentro de métodos estáticos usando this.

Ejemplo:

```
class Configuracion {
  static nombreAplicacion = "Mi App";

  static obtenerNombre() {
    return this.nombreAplicacion;
  }
}

console.log(Configuracion.obtenerNombre()); // Salida: "Mi App"
```

Uso avanzado: Para mantener configuraciones globales en una clase.

Modificación de prototipos nativos

Puedes extender los prototipos de objetos nativos (como Array o String), pero esto se considera una mala práctica en la mayoría de los casos.

Ejemplo (no recomendado):

```
Array.prototype.ultimo = function() {
  return this[this.length - 1];
};

const numeros = [1, 2, 3];
console.log(numeros.ultimo()); // Salida: 3
```

Uso avanzado: Solo en casos muy específicos y con mucho cuidado, ya que puede causar conflictos.

Composición sobre herencia

En lugar de usar herencia, puedes usar composición para construir objetos combinando funcionalidades.

Ejemplo:

```
const Nadador = {
  nadar() {
    console.log("Nadando...");
  }
};

const Volador = {
  volar() {
    console.log("Volando...");
  }
};

class Pato {
  constructor() {
    Object.assign(this, Nadador, Volador);
  }
}

const pato = new Pato();
pato.nadar(); // Salida: "Nadando..."
pato.volar(); // Salida: "Volando..."
```

Uso avanzado: Para evitar jerarquías complejas y favorecer la reutilización de código.

Seleccionar elementos

getElementById()

Este método selecciona un elemento del DOM por su atributo id. Es uno de los métodos más rápidos y específicos para seleccionar elementos.

Sintaxis:

```
let elemento = document.getElementById('idDelElemento');
```

Ejemplo:

```
<div id="miDiv">Hola Mundo</div>
```

```
let div = document.getElementById('miDiv');
console.log(div.textContent); // "Hola Mundo"
```

Notas avanzadas:

Solo puede seleccionar un elemento, ya que los id deben ser únicos en el documento.

Si no encuentra el elemento, devuelve null.

querySelector()

Este método es más versátil y permite seleccionar elementos usando selectores CSS. Puede seleccionar el primer elemento que coincida con el selector.

Sintaxis:

```
let elemento = document.querySelector('selectorCSS');
```

Ejemplos:

Seleccionar por id:

```
let div = document.querySelector('#miDiv');
```

Seleccionar por clase:

```
let elemento = document.querySelector('.miClase');
```

Seleccionar por etiqueta:

```
let parrafo = document.querySelector('p');
```

Seleccionar por atributo:

```
let input = document.querySelector('input[type="text"]');
```

Notas avanzadas:

Si no encuentra ningún elemento, devuelve null.

Puedes usar cualquier selector CSS válido.

Para seleccionar todos los elementos que coincidan con un selector, usa querySelectorAll():

```
let elementos = document.querySelectorAll('.miClase');
```

Modificar contenido

.textContent

Esta propiedad permite obtener o establecer el texto contenido dentro de un elemento. Ignora las etiquetas HTML y solo devuelve el texto.

Sintaxis:

```
elemento.textContent = 'Nuevo texto';
```

Ejemplo:

```
<p id="miParrafo">Este es un <strong>párrafo</strong>.</p>
```

```
let parrafo = document.getElementById('miParrafo');
console.log(parrafo.textContent); // "Este es un párrafo."
parrafo.textContent = 'Texto cambiado';
console.log(parrafo.textContent); // "Texto cambiado"
```

Notas avanzadas:

Es seguro contra ataques XSS (Cross-Site Scripting), ya que no interpreta HTML.

Si asignas HTML como texto, este no se renderizará, sino que se mostrará como texto plano.

.innerHTML

Esta propiedad permite obtener o establecer el contenido HTML dentro de un elemento. A diferencia de .textContent, sí interpreta las etiquetas HTML.

Sintaxis:

```
elemento.innerHTML = 'Nuevo contenido HTML';
```

Ejemplo:

```
<div id="miDiv">Hola Mundo</div>
```

```
let div = document.getElementById('miDiv');
div.innerHTML = '<strong>Hola</strong> Mundo';
console.log(div.innerHTML); // "<strong>Hola</strong> Mundo"
```

Notas avanzadas:

Es útil para insertar contenido dinámico con etiquetas HTML.

¡Cuidado con XSS! Si insertas contenido no confiable, podrías exponer tu aplicación a ataques de inyección de código.

Para evitar XSS, sanitiza el contenido o usa .textContent cuando no necesites HTML.

.value

Esta propiedad se usa principalmente para obtener o establecer el valor de elementos de formulario, como

```
<input>, <textarea>, y <select>.
```

Sintaxis:

```
elemento.value = 'Nuevo valor';
```

Ejemplo:

```
<input type="text" id="miInput" value="Texto inicial">
```

```
let input = document.getElementById('miInput');
console.log(input.value); // "Texto inicial"
input.value = 'Nuevo texto';
console.log(input.value); // "Nuevo texto"
```

Notas avanzadas:

Solo funciona con elementos de formulario.

Para elementos como `<select>`, devuelve el valor de la opción seleccionada.

Para `<input type="checkbox">` o `<input type="radio">`, usa .checked en lugar de .value.

Ejemplo avanzado combinado

```
<div id="contenedor">
  <p class="texto">Este es un párrafo.</p>
  <input type="text" id="entrada" value="Escribe algo">
  <button id="boton">Cambiar contenido</button>
</div>
```

```
// Seleccionar elementos
let contenedor = document.getElementById('contenedor');
let parrafo = document.querySelector('.texto');
let input = document.getElementById('entrada');
let boton = document.getElementById('boton');

// Modificar contenido al hacer clic en el botón
boton.addEventListener('click', function () {
  // Cambiar el texto del párrafo
  parrafo.textContent = 'Texto cambiado con JavaScript';

  // Insertar HTML dinámico
  contenedor.innerHTML += '<p>Nuevo párrafo añadido</p>';

  // Obtener el valor del input
  console.log(input.value);
});
```

Resumen

Seleccionar elementos:

`getElementById()`: Selecciona por id.

`querySelector()`: Selecciona usando selectores CSS.

Modificar contenido:

`.textContent`: Para texto plano.

`.innerHTML`: Para contenido HTML.

`.value`: Para valores de formularios.

Modificar estilos

.style

La propiedad .style permite modificar los estilos CSS de un elemento directamente desde JavaScript. Puedes acceder a las propiedades CSS como propiedades del objeto .style.

Sintaxis:

```
elemento.style.propiedadCSS = 'valor';
```

Ejemplo:

```
<div id="miDiv">Hola Mundo</div>
```

```
let div = document.getElementById('miDiv');
div.style.color = 'red'; // Cambia el color del texto a rojo
div.style.backgroundColor = 'yellow'; // Cambia el fondo a amarillo
div.style.fontSize = '20px'; // Cambia el tamaño de la fuente
```

Notas avanzadas:

Las propiedades CSS que tienen guiones (como background-color) se convierten en camelCase en JavaScript (backgroundColor).

Los valores deben ser cadenas de texto (por ejemplo, '20px', '#ff0000').

Este método es útil para cambios de estilo puntuales, pero no es recomendable para estilos complejos (mejor usar clases CSS).

.classList.add()

Este método añade una o más clases a un elemento. Es útil para aplicar estilos predefinidos en CSS.

Sintaxis:

```
elemento.classList.add('clase1', 'clase2', ...);
```

Ejemplo:

```
<div id="miDiv">Hola Mundo</div>
```

```
.destacado {
  color: red;
  font-weight: bold;
}
```

```
let div = document.getElementById('miDiv');
div.classList.add('destacado'); // Añade la clase "destacado"
```

Notas avanzadas:

Puedes añadir múltiples clases separándolas por comas.

Si la clase ya existe, no se duplica.

.classList.remove()

Este método elimina una o más clases de un elemento.

Sintaxis:

```
elemento.classList.remove('clase1', 'clase2', ...);
```

Ejemplo:

```
let div = document.getElementById('miDiv');
div.classList.remove('destacado'); // Elimina la clase "destacado"
```

Notas avanzadas:

Si la clase no existe, no ocurre ningún error.

Puedes eliminar múltiples clases a la vez.

.classList.toggle() (Extra)

Este método alterna (añade o elimina) una clase. Si la clase existe, la elimina; si no existe, la añade.

Sintaxis:

```
elemento.classList.toggle('clase');
```

Ejemplo:

```
let div = document.getElementById('miDiv');
div.classList.toggle('destacado'); // Alterna la clase "destacado"
```

Crear y eliminar elementos

createElement()

Este método crea un nuevo elemento HTML. Sin embargo, el elemento no se añade automáticamente al DOM; debes insertarlo manualmente.

Sintaxis:

```
let nuevoElemento = document.createElement('etiqueta');
```

Ejemplo:

```
let nuevoParrafo = document.createElement('p');
nuevoParrafo.textContent = 'Este es un nuevo párrafo.';
```

Notas avanzadas:

Puedes crear cualquier elemento HTML válido (por ejemplo, 'div', 'span', 'img', etc.).

Después de crearlo, debes insertarlo en el DOM usando métodos como appendChild() o insertBefore().

appendChild()

Este método añade un elemento como último hijo de otro elemento.

Sintaxis:

```
elementoPadre.appendChild(nuevoElemento);
```

Ejemplo:

```
<div id="contenedor"></div>
```

```
let contenedor = document.getElementById('contenedor');
let nuevoParrafo = document.createElement('p');
nuevoParrafo.textContent = 'Este es un nuevo párrafo.';
contenedor.appendChild(nuevoParrafo);
```

Notas avanzadas:

Si el elemento ya existe en el DOM, se moverá a la nueva posición.

Puedes usarlo para mover elementos dentro del DOM.

removeChild()

Este método elimina un elemento hijo de su elemento padre.

Sintaxis:

```
elementoPadre.removeChild(elementoHijo);
```

Ejemplo:

```
<div id="contenedor">
  <p id="parrafo">Este párrafo será eliminado.</p>
</div>
```

```
let contenedor = document.getElementById('contenedor');
let parrafo = document.getElementById('parrafo');
contenedor.removeChild(parrafo); // Elimina el párrafo
```

Notas avanzadas:

Si no estás seguro de si el elemento existe, verifica primero:

```
if (parrafo) {
  contenedor.removeChild(parrafo);
}
```

También puedes usar elemento.remove() para eliminar un elemento directamente:

```
parrafo.remove(); // Elimina el párrafo sin necesidad de referenciar al padre
```

Ejemplo avanzado combinado

```
<div id="contenedor">
  <p class="texto">Este es un párrafo.</p>
  <button id="boton">Cambiar estilo y contenido</button>
</div>
```

```
// Seleccionar elementos
let contenedor = document.getElementById('contenedor');
let parrafo = document.querySelector('.texto');
let boton = document.getElementById('boton');

// Modificar estilos y contenido al hacer clic en el botón
boton.addEventListener('click', function () {
  // Cambiar estilos con .style
  parrafo.style.color = 'blue';
  parrafo.style.fontSize = '24px';

  // Añadir una clase con .classList.add()
  parrafo.classList.add('destacado');

  // Crear un nuevo elemento
  let nuevoParrafo = document.createElement('p');
  nuevoParrafo.textContent = 'Este es un nuevo párrafo añadido dinámicamente.';

  // Añadir el nuevo elemento al contenedor
  contenedor.appendChild(nuevoParrafo);

  // Eliminar el párrafo original después de 3 segundos
  setTimeout(() => {
    contenedor.removeChild(parrafo);
  }, 3000);
});
```

Resumen

Modificar estilos:

.style: Para estilos en línea.

.classList.add(): Para añadir clases.

.classList.remove(): Para eliminar clases.

Crear y eliminar elementos:

createElement(): Para crear nuevos elementos.

appendChild(): Para añadir elementos al DOM.

removeChild(): Para eliminar elementos del DOM.

addEventListener()

El método addEventListener() permite asociar un evento a un elemento del DOM. Cuando el evento ocurre, se ejecuta una función (llamada "manejador de eventos" o "event handler").

Sintaxis:

```
elemento.addEventListener('evento', funcionManejadora);
```

Parámetros:

evento: El nombre del evento (por ejemplo, 'click', 'change', 'keyup').

funcionManejadora: La función que se ejecutará cuando ocurra el evento.

Ejemplo básico:

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

```
let boton = document.getElementById('miBoton');
boton.addEventListener('click', function () {
  alert('¡Hiciste clic en el botón!');
});
```

Notas avanzadas:

Puedes añadir múltiples eventos al mismo elemento.

Puedes usar funciones anónimas (como en el ejemplo) o funciones declaradas.

Para eliminar un evento, usa removeEventListener().

Eventos comunes

click

El evento click se dispara cuando el usuario hace clic en un elemento.

Ejemplo:

```
<button id="miBoton">Haz clic</button>
<p id="mensaje"></p>
```

```
let boton = document.getElementById('miBoton');
let mensaje = document.getElementById('mensaje');

boton.addEventListener('click', function () {
  mensaje.textContent = '¡Hiciste clic en el botón!';
});
```

Notas avanzadas:

Funciona en cualquier elemento, no solo en botones.

Puedes detectar clics en imágenes, enlaces, divs, etc.

change

El evento change se dispara cuando el valor de un elemento de formulario cambia y pierde el foco (por ejemplo, en un `<input>, <select>, o <textarea>`).

Ejemplo:

```
<input type="text" id="miInput" placeholder="Escribe algo">
<p id="resultado"></p>
```

```
let input = document.getElementById('miInput');
let resultado = document.getElementById('resultado');

input.addEventListener('change', function () {
  resultado.textContent = `Escribiste: ${input.value}`;
});
```

Notas avanzadas:

Para `<input type="text">`, el evento se dispara cuando el campo pierde el foco.

Para `<select>`, se dispara cuando se selecciona una opción.

Si quieres detectar cambios en tiempo real (sin perder el foco), usa el evento input.

keyup

El evento keyup se dispara cuando el usuario suelta una tecla después de presionarla. Es útil para detectar entradas de texto en tiempo real.

Ejemplo:

```
<input type="text" id="miInput" placeholder="Escribe algo">
<p id="resultado"></p>
```

```
let input = document.getElementById('miInput');
let resultado = document.getElementById('resultado');

input.addEventListener('keyup', function () {
  resultado.textContent = `Escribiendo: ${input.value}`;
});
```

Notas avanzadas:

También existen eventos relacionados como keydown (cuando se presiona una tecla) y keypress (cuando se mantiene presionada una tecla).

Puedes acceder a la tecla presionada usando el objeto event:

```
input.addEventListener('keyup', function (event) {
  console.log(`Tecla presionada: ${event.key}`);
});
```

Objeto event

Cuando se ejecuta un manejador de eventos, recibe un objeto event que contiene información sobre el evento. Este objeto es útil para acceder a detalles como la tecla presionada, las coordenadas del clic, etc.

Propiedades comunes del objeto event:

event.target: El elemento que disparó el evento.

event.type: El tipo de evento (por ejemplo, 'click', 'keyup').

event.key: La tecla presionada (para eventos de teclado).

event.clientX y event.clientY: Las coordenadas del clic (para eventos de ratón).

Ejemplo:

```
document.addEventListener('click', function (event) {
  console.log(`Clic en: (${event.clientX}, ${event.clientY})`);
});
```

Ejemplo avanzado combinado

```
<input type="text" id="miInput" placeholder="Escribe algo">
<button id="miBoton">Haz clic</button>
<p id="resultado"></p>
```

```
// Seleccionar elementos
let input = document.getElementById('miInput');
let boton = document.getElementById('miBoton');
let resultado = document.getElementById('resultado');

// Evento keyup para el input
input.addEventListener('keyup', function (event) {
  resultado.textContent = `Escribiendo: ${input.value}`;
  console.log(`Tecla presionada: ${event.key}`);
});

// Evento click para el botón
boton.addEventListener('click', function () {
  resultado.textContent = `Texto final: ${input.value}`;
});

// Evento change para el input
input.addEventListener('change', function () {
  console.log('El valor del input cambió.');
});
```

Resumen

addEventListener(): Para registrar eventos en elementos.

Eventos comunes:

click: Para clics del ratón.

change: Para cambios en elementos de formulario.

keyup: Para detectar teclas presionadas.

Objeto event: Proporciona información detallada sobre el evento.

setTimeout y setInterval

setTimeout

¿Qué hace? Ejecuta una función después de un retraso especificado en milisegundos.

Sintaxis:

```
setTimeout(callback, delay, arg1, arg2, ...);
```

callback: La función que se ejecutará.

delay: El tiempo de espera en milisegundos antes de ejecutar la función.

arg1, arg2, ...: Argumentos opcionales que se pasan al callback.

Ejemplo básico:

```
setTimeout(() => {
    console.log("Este mensaje se muestra después de 2 segundos");
}, 2000);
```

setInterval

¿Qué hace? Ejecuta una función repetidamente, con un intervalo fijo entre cada ejecución.

Sintaxis:

```
setInterval(callback, interval, arg1, arg2, ...);
```

callback: La función que se ejecutará repetidamente.

interval: El tiempo en milisegundos entre cada ejecución.

arg1, arg2, ...: Argumentos opcionales que se pasan al callback.

Ejemplo básico:

```
setInterval(() => {
    console.log("Este mensaje se muestra cada 3 segundos");
}, 3000);
```

Uso Avanzado de setTimeout y setInterval

Cancelar setTimeout y setInterval

clearTimeout: Cancela la ejecución de un setTimeout.

```
const timeoutId = setTimeout(() => {
    console.log("Este mensaje nunca se mostrará");
}, 2000);

clearTimeout(timeoutId); // Cancela el timeout
```

clearInterval: Detiene la ejecución repetida de un setInterval.

```
const intervalId = setInterval(() => {
    console.log("Este mensaje se muestra cada 1 segundo");
}, 1000);

setTimeout(() => {
    clearInterval(intervalId); // Detiene el intervalo después de 5 segundos
}, 5000);
```

Anidar setTimeout para crear intervalos dinámicos

En lugar de usar setInterval, puedes usar setTimeout de manera recursiva para crear intervalos dinámicos o controlar mejor el flujo de ejecución.

```
function ejecutarConRetardo() {
    console.log("Ejecutado con retardo");
    setTimeout(ejecutarConRetardo, 1000); // Se llama a sí mismo después de 1 segundo
}
ejecutarConRetardo();
```

Problemas Comunes con setInterval

Desbordamiento de llamadas: Si el código dentro del setInterval tarda más en ejecutarse que el intervalo especificado, las llamadas se acumulan.

Solución: Usar setTimeout recursivo para garantizar que la siguiente ejecución solo comience después de que la anterior haya terminado.

Falta de limpieza: Si no se limpia un setInterval, puede seguir ejecutándose incluso cuando ya no es necesario, lo que lleva a fugas de memoria.

Solución: Siempre usar clearInterval cuando el intervalo ya no sea necesario.

Aplicaciones en Programación Empresarial

Ejecución Diferida

Escenario: En una aplicación empresarial, es común retrasar la ejecución de ciertas tareas, como la validación de formularios o la carga de datos.

```
let timeoutId;
document.getElementById('inputField').addEventListener('input', () => {
    clearTimeout(timeoutId); // Cancela el timeout anterior
    timeoutId = setTimeout(() => {
```

```
        console.log("Validando entrada...");
        // Lógica de validación aquí
    }, 500); // Espera 500 ms después de la última entrada
});
```

Actualización Periódica de Datos

Escenario: En aplicaciones que consumen APIs, es común actualizar datos periódicamente (por ejemplo, para mostrar información en tiempo real).

```
let intervalId = setInterval(async () => {
    try {
        const response = await fetch('https://api.ejemplo.com/datos');
        const data = await response.json();
        console.log("Datos actualizados:", data);
    } catch (error) {
        console.error("Error al actualizar datos:", error);
        clearInterval(intervalId); // Detiene el intervalo si hay un error
    }
}, 10000); // Actualiza cada 10 segundos
```

Problemas Frecuentes para Programadores Junior

Callback Hell con setTimeout

Problema: Anidar múltiples setTimeout puede llevar a un código difícil de leer y mantener.

```
setTimeout(() => {
    console.log("Primera tarea");
    setTimeout(() => {
        console.log("Segunda tarea");
        setTimeout(() => {
            console.log("Tercera tarea");
        }, 1000);
    }, 1000);
}, 1000);
```

Solución: Usar promesas o async/await para manejar la asincronía de manera más limpia.

No Limpiar Intervarlos o Timeouts

Problema: Si no se limpian los intervalos o timeouts, pueden seguir ejecutándose incluso cuando ya no son necesarios.

```
const intervalId = setInterval(() => {
    console.log("Ejecutando...");
}, 1000);

// Olvidar clearInterval(intervalId) puede llevar a fugas de memoria.
```

Solución: Siempre limpiar los intervalos y timeouts cuando ya no sean necesarios.

Uso Excesivo de setInterval

Problema: Usar setInterval para tareas que no requieren una ejecución estrictamente periódica puede llevar a un alto consumo de recursos.

Solución: Evaluar si es necesario un intervalo o si se puede usar un enfoque más eficiente, como setTimeout recursivo.

Buenas Prácticas

Evitar el uso excesivo de setInterval: Prefiere setTimeout recursivo para tareas que no requieren una ejecución estrictamente periódica.

Limpiar intervalos y timeouts: Siempre usa clearInterval y clearTimeout cuando ya no sean necesarios.

Manejar errores: Asegúrate de manejar errores en tareas asincrónicas para evitar comportamientos inesperados.

Optimizar el rendimiento: Evita ejecutar código costoso en intervalos cortos.

Conceptos Básicos de Promesas y async/await

Promesas (Promise)

¿Qué es una promesa? Una promesa es un objeto que representa un valor que puede estar disponible ahora, en el futuro o nunca. Es una forma más limpia y poderosa de manejar operaciones asincrónicas en comparación con los callbacks.

Estados de una promesa:

- Pending (Pendiente): Estado inicial, la promesa no se ha cumplido ni rechazado.
- Fulfilled (Cumplida): La operación se completó con éxito.
- Rejected (Rechazada): La operación falló.

Sintaxis básica:

```
const promesa = new Promise((resolve, reject) => {
    // Lógica asincrónica
```

```
    if (éxito) {
        resolve("Éxito"); // La promesa se cumple
    } else {
        reject("Error"); // La promesa se rechaza
    }
});

promesa
    .then((resultado) => {
        console.log(resultado); // "Éxito"
    })
    .catch((error) => {
        console.error(error); // "Error"
    });
```

async/await

¿Qué es async/await? Es una sintaxis más moderna y legible para trabajar con promesas. Permite escribir código asincrónico de manera similar al código síncrono.

Sintaxis básica:

```
async function obtenerDatos() {
    try {
        const resultado = await promesa; // Espera a que la promesa se resuelva
        console.log(resultado); // "Éxito"
    } catch (error) {
        console.error(error); // "Error"
    }
}
obtenerDatos();
```

Uso Avanzado de Promesas y async/await

Encadenamiento de Promesas Las promesas se pueden encadenar para ejecutar operaciones asincrónicas en secuencia.

```
function tarea1() {
    return new Promise((resolve) => {
        setTimeout(() => resolve("Tarea 1 completada"), 1000);
    });
}

function tarea2() {
    return new Promise((resolve) => {
        setTimeout(() => resolve("Tarea 2 completada"), 1000);
    });
}
```

```
tarea1()
    .then((resultado1) => {
        console.log(resultado1);
        return tarea2();
    })
    .then((resultado2) => {
        console.log(resultado2);
    })
    .catch((error) => {
        console.error(error);
    });
```

Manejo de Múltiples Promesas

Promise.all: Ejecuta múltiples promesas en paralelo y espera a que todas se resuelvan.

```
Promise.all([tarea1(), tarea2()])
    .then((resultados) => {
        console.log("Todas las tareas completadas:", resultados);
    })
    .catch((error) => {
        console.error("Al menos una tarea falló:", error);
    });
```

Promise.race: Devuelve la primera promesa que se resuelva (ya sea cumplida o rechazada).

```
Promise.race([tarea1(), tarea2()])
    .then((resultado) => {
        console.log("La primera tarea en completarse:", resultado);
    })
    .catch((error) => {
        console.error("La primera tarea en fallar:", error);
    });
```

async/await con Bucles

Puedes usar async/await dentro de bucles para manejar operaciones asincrónicas en secuencia.

```
async function ejecutarTareas() {
    const tareas = [tarea1, tarea2];
    for (const tarea of tareas) {
        const resultado = await tarea();
        console.log(resultado);
    }
}
ejecutarTareas();
```

Aplicaciones en Programación Empresarial

Consumo de APIs

Escenario: En aplicaciones empresariales, es común consumir APIs para obtener datos.

```
async function obtenerDatosDeAPI() {
    try {
        const response = await fetch('https://api.ejemplo.com/datos');
        const data = await response.json();
        console.log("Datos obtenidos:", data);
    } catch (error) {
        console.error("Error al obtener datos:", error);
    }
}
obtenerDatosDeAPI();
```

Validación de Datos en Formularios

Escenario: Validar datos de un formulario antes de enviarlos al servidor.

```
async function validarYEnviarFormulario(datos) {
    try {
        await validarDatos(datos); // Validación asincrónica
        const respuesta = await enviarDatos(datos); // Envío asincrónico
        console.log("Formulario enviado con éxito:", respuesta);
    } catch (error) {
        console.error("Error en el formulario:", error);
    }
}
```

Procesamiento en Paralelo

Escenario: Procesar múltiples tareas en paralelo para mejorar el rendimiento.

```
async function procesarTareas() {
    try {
        const [resultado1, resultado2] = await Promise.all([tarea1(), tarea2()]);
        console.log("Resultados:", resultado1, resultado2);
    } catch (error) {
        console.error("Error en el procesamiento:", error);
    }
}
procesarTareas();
```

Problemas Frecuentes para Programadores Junior

No Manejar Errores Adecuadamente

Problema: Olvidar el catch o try...catch puede llevar a errores no manejados.

```
promesa
    .then((resultado) => {
        console.log(resultado);
    });
    // Falta .catch para manejar errores
```

Solución: Siempre incluir un bloque catch o try...catch para manejar errores.

Uso Incorrecto de await

Problema: Usar await en lugares innecesarios puede bloquear la ejecución del código.

```
async function ejecutar() {
    const resultado1 = await tarea1(); // Espera aquí
    const resultado2 = await tarea2(); // Espera aquí
    console.log(resultado1, resultado2);
}
```

Solución: Usar Promise.all para ejecutar tareas en paralelo cuando sea posible.

Callback Hell con Promesas

Problema: Anidar múltiples promesas puede llevar a un código difícil de leer.

```
tarea1()
    .then((resultado1) => {
        tarea2(resultado1)
            .then((resultado2) => {
                tarea3(resultado2)
                    .then((resultado3) => {
                        console.log(resultado3);
                    });
            });
    });
```

Solución: Usar async/await para escribir código más limpio y legible.

Buenas Prácticas

Manejar errores siempre: Usa catch o try...catch para evitar errores no manejados.

Evitar el anidamiento excesivo: Usa async/await para mejorar la legibilidad del código.

Usar Promise.all para tareas paralelas: Mejora el rendimiento ejecutando tareas en paralelo cuando sea posible.

Evitar el uso innecesario de await: No uses await si no es necesario, ya que puede bloquear la ejecución del código.

Conceptos Básicos de AJAX y Fetch API

¿Qué es AJAX?

AJAX (Asynchronous JavaScript and XML):

Es una técnica que permite realizar solicitudes HTTP asincrónicas desde el navegador sin necesidad de recargar la página. Aunque el nombre incluye "XML", hoy en día se usa principalmente JSON.

¿Qué es Fetch API?

Fetch API: Es una interfaz moderna y nativa de JavaScript para realizar solicitudes HTTP. Es más poderosa y flexible que el antiguo XMLHttpRequest.

Uso Básico de Fetch API

Sintaxis Básica de Fetch fetch(): Realiza una solicitud HTTP y devuelve una promesa que se resuelve en la respuesta.

```
fetch('https://api.ejemplo.com/datos')
    .then(response => response.json()) // Convierte la respuesta a JSON
    .then(data => console.log(data))  // Muestra los datos
    .catch(error => console.error('Error:', error)); // Maneja errores
```

Métodos HTTP Comunes

GET: Obtener datos.

```
fetch('https://api.ejemplo.com/datos')
    .then(response => response.json())
    .then(data => console.log(data))
    .catch(error => console.error('Error:', error));
```

POST: Enviar datos.

```
fetch('https://api.ejemplo.com/datos', {
    method: 'POST',
```

```
    headers: {
        'Content-Type': 'application/json',
    },
    body: JSON.stringify({ clave: 'valor' }), // Datos a enviar
})
    .then(response => response.json())
    .then(data => console.log(data))
    .catch(error => console.error('Error:', error));
```

PUT y DELETE: Actualizar y eliminar datos.

```
// PUT
fetch('https://api.ejemplo.com/datos/1', {
    method: 'PUT',
    headers: {
        'Content-Type': 'application/json',
    },
    body: JSON.stringify({ clave: 'nuevo valor' }),
});

// DELETE
fetch('https://api.ejemplo.com/datos/1', {
    method: 'DELETE',
});
```

Uso Avanzado de Fetch API

Manejo de Headers

Puedes agregar cabeceras personalizadas a tus solicitudes.

```
fetch('https://api.ejemplo.com/datos', {
    headers: {
        'Authorization': 'Bearer token',
        'Content-Type': 'application/json',
    },
});
```

Manejo de Errores

Verificar el estado de la respuesta:

No todas las respuestas con estado HTTP 200-299 son válidas. Es importante verificar response.ok.

```
fetch('https://api.ejemplo.com/datos')
    .then(response => {
```

```
        if (!response.ok) {
            throw new Error('Error en la solicitud');
        }
        return response.json();
    })
    .then(data => console.log(data))
    .catch(error => console.error('Error:', error));
```

Uso de async/await con Fetch

async/await hace que el código sea más legible y fácil de manejar.

```
async function obtenerDatos() {
    try {
        const response = await fetch('https://api.ejemplo.com/datos');
        if (!response.ok) {
            throw new Error('Error en la solicitud');
        }
        const data = await response.json();
        console.log(data);
    } catch (error) {
        console.error('Error:', error);
    }
}
obtenerDatos();
```

Cancelación de Solicitudes

AbortController: Permite cancelar una solicitud fetch.

```
const controller = new AbortController();
const signal = controller.signal;

fetch('https://api.ejemplo.com/datos', { signal })
    .then(response => response.json())
    .then(data => console.log(data))
    .catch(error => {
        if (error.name === 'AbortError') {
            console.log('Solicitud cancelada');
        } else {
            console.error('Error:', error);
        }
    });

// Cancelar la solicitud después de 1 segundo
setTimeout(() => controller.abort(), 1000);
```

Aplicaciones en Programación Empresarial

Consumo de APIs Externas

Escenario: Obtener datos de una API externa para mostrar en una aplicación.

```
async function obtenerDatosDeAPI() {
    try {
        const response = await fetch('https://api.ejemplo.com/datos');
        if (!response.ok) {
            throw new Error('Error en la solicitud');
        }
        const data = await response.json();
        console.log("Datos obtenidos:", data);
    } catch (error) {
        console.error("Error al obtener datos:", error);
    }
}
obtenerDatosDeAPI();
```

Autenticación y Autorización

Escenario: Enviar un token de autenticación en las cabeceras.

```
async function obtenerDatosProtegidos() {
    try {
        const token = 'tu_token';
        const response = await fetch('https://api.ejemplo.com/datos-protegidos', {
            headers: {
                'Authorization': `Bearer ${token}`,
            },
        });
        if (!response.ok) {
            throw new Error('Error en la solicitud');
        }
        const data = await response.json();
        console.log("Datos protegidos:", data);
    } catch (error) {
        console.error("Error al obtener datos protegidos:", error);
    }
}
obtenerDatosProtegidos();
```

Subida de Archivos

Escenario: Subir un archivo a un servidor.

```
async function subirArchivo(archivo) {
    const formData = new FormData();
    formData.append('archivo', archivo);

    try {
        const response = await fetch('https://api.ejemplo.com/subir', {
            method: 'POST',
            body: formData,
        });
        if (!response.ok) {
            throw new Error('Error en la subida');
        }
        const data = await response.json();
        console.log("Archivo subido:", data);
    } catch (error) {
        console.error("Error al subir el archivo:", error);
    }
}
```

Problemas Frecuentes para Programadores Junior

No Verificar el Estado de la Respuesta

Problema: Asumir que todas las respuestas son válidas sin verificar response.ok.

```
fetch('https://api.ejemplo.com/datos')
    .then(response => response.json())
    .then(data => console.log(data))
    .catch(error => console.error('Error:', error));
```

Solución: Siempre verificar response.ok antes de procesar la respuesta.

No Manejar Errores de Red

Problema: Ignorar errores de red o respuestas no válidas.

```
fetch('https://api.ejemplo.com/datos')
    .then(response => response.json())
    .then(data => console.log(data));
```

Solución: Usar try...catch o .catch() para manejar errores.

Uso Incorrecto de async/await

Problema: Usar await en lugares innecesarios, bloqueando la ejecución del código.

```
async function obtenerDatos() {
    const response = await fetch('https://api.ejemplo.com/datos'); // Espera aquí
    const data = await response.json(); // Espera aquí
    console.log(data);
}
```

Solución: Usar Promise.all para ejecutar tareas en paralelo cuando sea posible.

Buenas Prácticas

- Verificar el estado de la respuesta: Siempre verifica response.ok antes de procesar la respuesta.
- Manejar errores adecuadamente: Usa try...catch o .catch() para manejar errores de red o respuestas no válidas.
- Usar async/await para mejorar la legibilidad: Escribe código más limpio y fácil de mantener.
- Optimizar el rendimiento: Usa AbortController para cancelar solicitudes innecesarias.

Conceptos Básicos de Almacenamiento Local

¿Qué es localStorage?

localStorage: Es un mecanismo de almacenamiento persistente en el navegador. Los datos almacenados no tienen fecha de expiración y permanecen incluso después de cerrar el navegador.

¿Qué es sessionStorage?

sessionStorage: Es un mecanismo de almacenamiento temporal en el navegador. Los datos almacenados se eliminan cuando se cierra la pestaña o el navegador.

Uso Básico de localStorage y sessionStorage

Sintaxis Básica Guardar datos:

```
localStorage.setItem('clave', 'valor'); // Almacena en localStorage
sessionStorage.setItem('clave', 'valor'); // Almacena en sessionStorage
```

Obtener datos:

```
const valorLocal = localStorage.getItem('clave'); // Obtiene de localStorage
const valorSession = sessionStorage.getItem('clave'); // Obtiene de sessionStorage
```

Eliminar datos:

```
```

```
localStorage.removeItem('clave'); // Elimina de localStorage
sessionStorage.removeItem('clave'); // Elimina de sessionStorage
```

Limpiar todo el almacenamiento:

```
localStorage.clear(); // Limpia todo el localStorage
sessionStorage.clear(); // Limpia todo el sessionStorage
```

Ejemplo Básico

```
// Guardar datos
localStorage.setItem('nombre', 'Juan');
sessionStorage.setItem('sesion', 'activa');

// Obtener datos
console.log(localStorage.getItem('nombre')); // "Juan"
console.log(sessionStorage.getItem('sesion')); // "activa"

// Eliminar datos
localStorage.removeItem('nombre');
sessionStorage.removeItem('sesion');

// Limpiar todo
localStorage.clear();
sessionStorage.clear();
```

Uso Avanzado de localStorage y sessionStorage

Almacenamiento de Objetos

Problema: localStorage y sessionStorage solo pueden almacenar cadenas de texto (strings).

Solución: Usar JSON.stringify para convertir objetos a cadenas y JSON.parse para convertirlos de vuelta a objetos.

```
const usuario = { nombre: 'Juan', edad: 30 };

// Guardar objeto
localStorage.setItem('usuario', JSON.stringify(usuario));

// Obtener objeto
const usuarioGuardado = JSON.parse(localStorage.getItem('usuario'));
console.log(usuarioGuardado); // { nombre: 'Juan', edad: 30 }
```

Eventos de Almacenamiento

Evento storage: Se dispara cuando se modifica el localStorage o sessionStorage desde otra pestaña o ventana.

```
window.addEventListener('storage', (event) => {
    console.log('Cambio en el almacenamiento:', event.key, event.newValue);
});
```

Limitaciones de Almacenamiento

Tamaño máximo: Aproximadamente 5 MB por dominio (depende del navegador).

Problema: Almacenar demasiados datos puede afectar el rendimiento del navegador.

Solución: Usar el almacenamiento local solo para datos pequeños y necesarios.

Aplicaciones en Programación Empresarial

Guardar Preferencias del Usuario

Escenario: Guardar preferencias como el tema (claro/oscuro) o el idioma seleccionado.

```
// Guardar preferencia
localStorage.setItem('tema', 'oscuro');

// Obtener preferencia
const tema = localStorage.getItem('tema') || 'claro'; // Valor por defecto
console.log("Tema seleccionado:", tema);
```

Mantener el Estado de la Aplicación

Escenario: Guardar el estado de una aplicación (por ejemplo, un carrito de compras) para que persista incluso después de recargar la página.

```
const carrito = [{ id: 1, nombre: 'Producto 1', cantidad: 2 }];

// Guardar carrito
localStorage.setItem('carrito', JSON.stringify(carrito));

// Obtener carrito
const carritoGuardado = JSON.parse(localStorage.getItem('carrito')) || [];
console.log("Carrito:", carritoGuardado);
```

Autenticación y Tokens

Escenario: Guardar un token de autenticación para mantener la sesión del usuario.

```
// Guardar token
localStorage.setItem('token', 'tu_token');

// Obtener token
const token = localStorage.getItem('token');
if (token) {
    console.log("Usuario autenticado");
} else {
    console.log("Usuario no autenticado");
}
```

Problemas Frecuentes para Programadores Junior

No Manejar Datos No Existentes

Problema: Intentar acceder a datos que no existen puede llevar a errores.

```
const datos = localStorage.getItem('datos');
console.log(datos.nombre); // Error si datos es null
```

Solución: Verificar si los datos existen antes de usarlos.

```
const datos = JSON.parse(localStorage.getItem('datos')) || {};
console.log(datos.nombre); // undefined (pero no genera error)
```

Almacenar Datos Sensibles

Problema: Almacenar información sensible (como contraseñas o tokens sin cifrar) en localStorage o sessionStorage puede ser inseguro.

Solución: Evitar almacenar datos sensibles en el almacenamiento local. Usar cookies seguras o almacenamiento en el servidor.

No Limpiar Datos Obsoletos

Problema: Acumular datos obsoletos en el almacenamiento local puede llevar a un uso excesivo de memoria.

Solución: Limpiar regularmente los datos que ya no son necesarios.

```
localStorage.removeItem('datos_obsoletos');
```

Uso Excesivo de Almacenamiento Local

Problema: Almacenar grandes cantidades de datos puede afectar el rendimiento del navegador.

Solución: Usar el almacenamiento local solo para datos pequeños y necesarios. Para grandes volúmenes de datos, considerar el uso de bases de datos en el servidor.

Buenas Prácticas

- Usar JSON para objetos: Convertir objetos a cadenas con JSON.stringify y viceversa con JSON.parse.
- Verificar datos existentes: Siempre verificar si los datos existen antes de usarlos.
- Evitar datos sensibles: No almacenar información sensible en el almacenamiento local.
- Limpiar datos obsoletos: Eliminar regularmente los datos que ya no son necesarios.
- Optimizar el uso: Usar el almacenamiento local solo para datos pequeños y necesarios.

Conceptos Básicos de Manejo de Errores

¿Por qué es importante manejar errores?

El manejo de errores es crucial para evitar que una aplicación falle de manera inesperada y para proporcionar una experiencia de usuario más robusta y confiable.

En entornos empresariales, un manejo adecuado de errores puede prevenir pérdidas de datos, mejorar la depuración y facilitar el mantenimiento del código.

Errores en JavaScript

Errores comunes:

- Errores de sintaxis (SyntaxError).
- Errores de tipo (TypeError).
- Errores de referencia (ReferenceError).
- Errores personalizados (throw new Error()).

Uso Básico de try...catch

Sintaxis de try...catch

- try: Bloque de código que puede generar un error.
- catch: Bloque que se ejecuta si ocurre un error en el try.
- finally: Bloque opcional que se ejecuta siempre, independientemente de si hubo un error o no.

```
try {
    // Código que puede generar un error
    const resultado = operacionRiesgosa();
    console.log(resultado);
```

```
} catch (error) {
    // Manejo del error
    console.error("Ocurrió un error:", error.message);
} finally {
    // Código que siempre se ejecuta
    console.log("Finalizado");
}
```

Ejemplo Básico

```
function dividir(a, b) {
    if (b === 0) {
        throw new Error("División por cero no permitida");
    }
    return a / b;
}

try {
    const resultado = dividir(10, 0);
    console.log(resultado);
} catch (error) {
    console.error("Error:", error.message); // "División por cero no permitida"
}
```

Uso Avanzado de try...catch y throw new Error()

Tipos de Errores Personalizados

Puedes crear errores personalizados para manejar situaciones específicas.

```
class ErrorPersonalizado extends Error {
    constructor(mensaje) {
        super(mensaje);
        this.name = "ErrorPersonalizado";
    }
}

try {
    throw new ErrorPersonalizado("Este es un error personalizado");
} catch (error) {
    console.error(error.name); // "ErrorPersonalizado"
    console.error(error.message); // "Este es un error personalizado"
}
```

Manejo de Errores en Funciones Asincrónicas

Promesas: Usar .catch() para manejar errores.

```
function operacionAsincrona() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            reject(new Error("Error en operación asincrónica"));
        }, 1000);
    });
}

operacionAsincrona()
    .then(resultado => console.log(resultado))
    .catch(error => console.error(error.message)); // "Error en operación
asincrónica"
```

async/await: Usar try...catch para manejar errores.

```
async function ejecutarOperacion() {
    try {
        const resultado = await operacionAsincrona();
        console.log(resultado);
    } catch (error) {
        console.error(error.message); // "Error en operación asincrónica"
    }
}
ejecutarOperacion();
```

Manejo de Errores en Bucles

Puedes manejar errores dentro de bucles para evitar que un error detenga toda la ejecución.

```
const tareas = [tarea1, tarea2, tarea3];

for (const tarea of tareas) {
    try {
        tarea();
    } catch (error) {
        console.error("Error en tarea:", error.message);
    }
}
```

Aplicaciones en Programación Empresarial

Validación de Datos

Escenario: Validar datos de entrada antes de procesarlos.

```
```

```
function validarDatos(datos) {
    if (!datos.nombre) {
        throw new Error("El nombre es requerido");
    }
    if (!datos.edad || datos.edad < 18) {
        throw new Error("La edad debe ser mayor o igual a 18");
    }
}

try {
    validarDatos({ nombre: "", edad: 15 });
} catch (error) {
    console.error("Error de validación:", error.message);
}
```

Consumo de APIs

Escenario: Manejar errores al consumir una API.

```
async function obtenerDatosDeAPI() {
    try {
        const response = await fetch('https://api.ejemplo.com/datos');
        if (!response.ok) {
            throw new Error("Error al obtener datos");
        }
        const data = await response.json();
        console.log(data);
    } catch (error) {
        console.error("Error en la API:", error.message);
    }
}
obtenerDatosDeAPI();
```

Procesamiento de Archivos

Escenario: Manejar errores al leer o procesar archivos.

```
async function procesarArchivo(archivo) {
    try {
        const contenido = await leerArchivo(archivo);
        console.log(contenido);
    } catch (error) {
        console.error("Error al procesar el archivo:", error.message);
    }
}
procesarArchivo('archivo.txt');
```

Problemas Frecuentes para Programadores Junior

No Manejar Errores Adecuadamente

Problema: Ignorar errores puede llevar a comportamientos inesperados.

```
function operacionRiesgosa() {
    throw new Error("Algo salió mal");
}

operacionRiesgosa(); // Error no manejado
```

Solución: Siempre usar try...catch o .catch() para manejar errores.

Errores Silenciosos

Problema: No registrar o notificar errores dificulta la depuración.

```
try {
    operacionRiesgosa();
} catch (error) {
    // No se registra el error
}
```

Solución: Registrar errores en la consola o enviarlos a un servicio de monitoreo.

Uso Incorrecto de throw

Problema: Lanzar errores sin un mensaje claro dificulta la identificación del problema.

```
throw "Algo salió mal"; // Error sin detalles
```

Solución: Usar throw new Error("Mensaje descriptivo") para proporcionar detalles útiles.

Buenas Prácticas

- Manejar todos los errores: Usa try...catch o .catch() para evitar errores no manejados.
- Registrar errores: Siempre registra errores en la consola o en un servicio de monitoreo.
- Proporcionar mensajes claros: Usa mensajes descriptivos en throw new Error().
- Evitar errores silenciosos: No ignores errores, incluso si parecen menores.
- Usar errores personalizados: Crea errores personalizados para manejar situaciones específicas.