Objetos y Estructuras Anidadas

JS

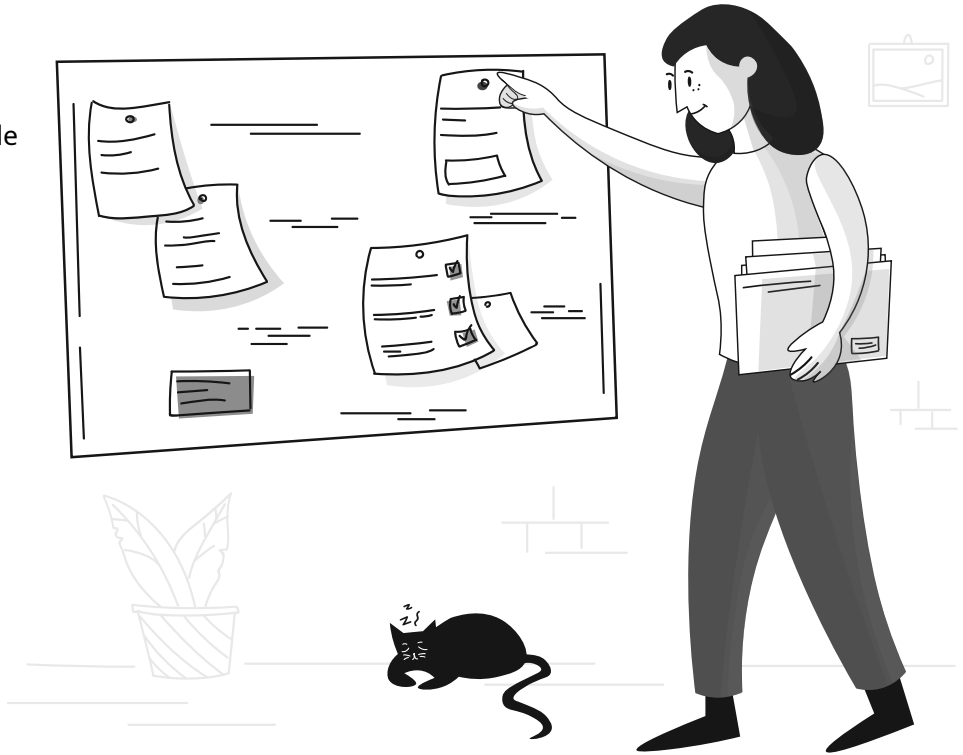
Introducción a los objetos en JavaScript

Un objeto en JavaScript es una colección de datos y funcionalidades relacionados, organizados en pares clave-valor llamados propiedades y métodos. Permite encapsular información para acceder y manipularla de forma sencilla e intuitiva.

Los objetos se definen usando llaves {} y pares clave:valor.

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

objeto persona sin funciones aun



1ObjetoBase.htm

Diferencias entre . y []

La notación de punto (obj.propiedad) es más legible y se usa cuando la clave es un identificador estático válido (sin espacios, caracteres especiales o números al inicio) y conocido de antemano.

La notación de corchetes (obj["propiedad"] o obj[variable]) permite acceder a propiedades con nombres dinámicos, que pueden contener espacios, caracteres especiales, o cuyo nombre se determina en tiempo de ejecución mediante variables o expresiones

Clase Completa

```
class Persona {
        #nombre;
        #edad;

        constructor(nombre, edad) {
            this.#nombre = nombre;
            this.#edad = edad;
        }
    obtenerNombre() {
        return this.#nombre;
    }
    obtenerEdad() {
        return this.#edad;
    }
    cumplirAnios() {
        this.#edad += 1;
      }
}
```

3ClaseVariablesOcultas.html

Estructura anidada

Definición de estructuras anidadas

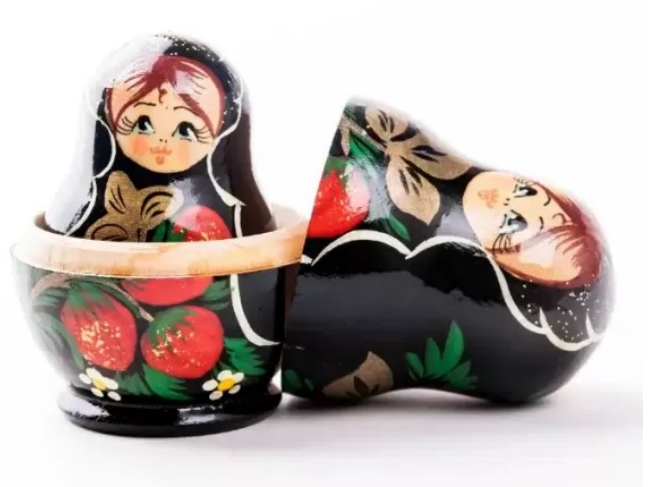
- En JavaScript, una estructura de datos anidada es una organización de información donde se combinan objetos y arrays dentro de otros objetos o arrays, permitiendo representar datos jerárquicos o relacionales de manera flexible y compleja
- Estas estructuras son esenciales para modelar información realista, como listas de usuarios, catálogos de productos, o configuraciones de aplicaciones.

```
let usuario = {
  nombre: "Ana",
  dirección: {
    calle: "Gran Vía",
    ciudad: "Madrid",
    codigoPostal: "28013"
  }
};
```

4ClasesAnidadas.html

Objetos con objeto interno

```
let usuario = {
  nombre: "Ana",
  direccion: {
    calle: "Gran Vía",
    ciudad: "Madrid",
    codigoPostal: "28013"
  }
};
```



```
console.log(usuario.nombre); // "Ana"

console.log(usuario.direccion.calle); // "Gran Vía"
console.log(usuario.direccion.ciudad); // "Madrid"
console.log(usuario.direccion.codigoPostal); // "28013"
```

Objetos con array interno

```
let estudiante = {
  nombre: "Carlos",
  notas: [8, 7.5, 9, 6]
};
```

```
console.log(estudiante.nombre);      // "Carlos"

console.log(estudiante.notas[0]);    // 8 (primera nota)

console.log(estudiante.notas.length); // 4 (cantidad de notas)
```

Sintaxis básica de desestructuración

- La desestructuración permite extraer valores de objetos o arrays y asignarlos a variables de forma concisa, usando llaves {} para objetos y corchetes [] para arrays

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


const [primero, segundo] = colores;


console.log(primero); // "rojo"
console.log(segundo); // "verde"
```

```
const persona = {
  nombre: "Goku",
  edad: 9001,
};


// Desestructuramos así:
const { nombre, edad } = persona;


// Ahora podemos usar "nombre" y "edad" directamente
console.log(nombre); // "Goku"
console.log(edad);   // 9001
```

6desestructurar.html

Recorrer datos con for...in

```
const persona = {
        nombre: "Laura",
        edad: 30,
        ciudad: "Madrid"
};

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

7ForIn.html

Desventajas for ... in

El uso de for...in en JavaScript tiene varias desventajas importantes, especialmente cuando se aplica sobre arrays o estructuras donde no es la opción más adecuada:

Itera sobre todas las propiedades enumerables, incluyendo las heredadas del prototipo. Esto puede provocar resultados inesperados si el objeto o array tiene propiedades agregadas manualmente o por librerías, ya que no solo recorre los índices numéricos, sino también cualquier método o propiedad extra que se haya añadido al objeto o prototipo.

No garantiza el orden de iteración. En los arrays, el orden de los índices no está asegurado al usar for...in, lo que puede ser problemático si necesitas procesar los elementos en orden secuencial.

Rendimiento subóptimo. Al recorrer arrays grandes, for...in suele ser más lento que otros métodos como for tradicional, for...of o métodos de array como forEach, ya que debe verificar todas las propiedades enumerables.

Propenso a errores. Es fácil cometer errores al usar for...in con arrays, ya que puedes terminar procesando propiedades que no son elementos del array, lo que puede llevar a bugs difíciles de detectar.

Object.keys(), Object.values() y Object.entries()

Object.keys(obj)

Devuelve un array con las claves (propiedades propias y enumerables) del objeto.

Object.values(obj)

Devuelve un array con los valores de esas propiedades.

Object.entries(obj)

Devuelve un array de pares [clave, valor] para cada propiedad.

De Objeto a Array

```
const persona = { nombre: "Lucía", edad: 28 };

const array = Object.entries(persona);

console.log(array); //  [["nombre", "Lucía"], ["edad", 28]]
```

De Array a Objeto

```
const array = [["nombre", "Lucía"], ["edad", 28]];

const objeto = Object.fromEntries(array);

console.log(objeto); // { nombre: "Lucía", edad: 28 }
```

De Objeto a Array

find: Devuelve el primer elemento del array que cumple con una condición dada.

findIndex: Devuelve el índice del primer elemento que cumple la condición, o -1 si no existe.

filter: Devuelve un nuevo array con todos los elementos que cumplen la condición especificada.

map: Crea un nuevo array con los resultados de aplicar una función a cada elemento.

reduce: Reduce el array a un solo valor, aplicando una función acumuladora sobre cada elemento.

Consejos finales

Elige estructuras de datos que sean claras y eficientes para el uso que necesitas, evitando anidamientos excesivos y priorizando la facilidad de acceso y modificación

Evitar errores comunes al acceder a niveles profundos

Antes de acceder a propiedades muy anidadas, valida la existencia de cada nivel usando encadenamiento opcional o comprobaciones lógicas.

Evita acceder directamente a propiedades profundas sin comprobar, ya que puede causar errores si alguna parte no existe:

```
// Incorrecto: puede lanzar error si dirección no existe

const codigoPostal = usuario.direccion.codigoPostal;

// Correcto: seguro ante propiedades faltantes

const codigoPostal = usuario.direccion?.codigoPostal ?? "No disponible";

Mantén las estructuras lo más planas posible cuando sea viable, para reducir la complejidad y los posibles errores de acceso
```