💻 Serialización y Deserialización de Datos

La **serialización** es el proceso de convertir un objeto JavaScript (como un Array u Object) en una cadena de texto para poder almacenarlo o transmitirlo (por ejemplo, a través de una red o al localStorage). La **deserialización** es el proceso inverso, tomando esa cadena de texto y volviéndola a convertir en el objeto original de JavaScript.

1. Serialización y Deserialización de JSON

JSON (JavaScript Object Notation) es el formato de intercambio de datos más común en JavaScript.

Concepto	Método	Descripción
Serialización	JSON.stringify(objeto)	Convierte un valor de JavaScript a una **cadena JSON**.
Deserialización	JSON.parse(cadena_json)	Convierte una **cadena JSON** a un valor de JavaScript.

Ejemplo con JSON

JavaScript

```javascript
// 1. Objeto JavaScript original
const usuario = {
  id: 1,
  nombre: "Alice",
  activo: true,
  roles: ["admin", "editor"]
};

// 2. Serialización: Objeto a cadena de texto JSON
const usuarioJSON = JSON.stringify(usuario);
console.log(typeof usuarioJSON);  // Salida: string
console.log(usuarioJSON);
// Salida: {"id":1,"nombre":"Alice","activo":true,"roles":["admin","editor"]}

// 3. Deserialización: Cadena de texto JSON a objeto JavaScript
const usuarioOriginal = JSON.parse(usuarioJSON);
console.log(typeof usuarioOriginal); // Salida: object
console.log(usuarioOriginal.nombre); // Salida: Alice

// 4. Navegar y obtener un valor
// Se navega directamente después de la deserialización, ya que es un objeto JS normal.
const primerRol = usuarioOriginal.roles[0];
console.log(`El primer rol es: ${primerRol}`); // Salida: El primer rol es: admin
```

---

2. Serialización y Deserialización de XML

JavaScript no tiene métodos nativos directos y sencillos como JSON.stringify para manejar XML, ya que es un formato menos común en el ecosistema JS moderno.

- **Serialización a XML:** Generalmente requiere construir la cadena XML manualmente o usar librerías externas (como xmlbuilder).
- **Deserialización de XML:** Generalmente se utiliza el objeto DOMParser para convertir una cadena XML en un **objeto Document** que puede ser navegado.

Ejemplo con XML (Deserialización y Navegación)

JavaScript

```
/*** PRUEBA EN NAVEGADOR WEB ***/
const xmlString = `
<libro id="101">
  <titulo>Cien años de soledad</titulo>
  <autor>Gabriel García Márquez</autor>
</libro>`;

// 1. Deserialización: Cadena XML a objeto Document
const parser = new DOMParser();
```

```
const xmlDoc = parser.parseFromString(xmlString, "text/xml");

// 2. Navegar y obtener un valor
// Usamos métodos del DOM (Document Object Model) para navegar.
const tituloElemento = xmlDoc.getElementsByTagName("titulo")[0];
const autorElemento = xmlDoc.getElementsByTagName("autor")[0];

const titulo = tituloElemento ? tituloElemento.textContent : 'No encontrado';
const idLibro = xmlDoc.documentElement.getAttribute('id');

console.log(`Título: ${titulo}`);        // Salida: Título: Cien años de soledad
console.log(`ID del libro: ${idLibro}`);  // Salida: ID del libro: 101
```

🛠️ Métodos de Array de Orden Superior

Los métodos filter() y reduce() son esenciales en la programación funcional de JavaScript, permitiendo manipular Arrays de forma concisa y declarativa.

1. Método Array.prototype.filter(): Filtrado de elementos

El método filter() crea un **nuevo array** con todos los elementos que pasan la prueba implementada por la función proporcionada. No modifica el array original.

- **Sintaxis:** array.filter(callback(elemento, indice, array))

- **Callback:** Debe devolver true (para incluir el elemento en el nuevo array) o false (para excluirlo).

Ejemplo con filter()

JavaScript

```
const numeros = [10, 5, 25, 8, 15, 30];

// Filtrar números mayores a 12
const mayoresA12 = numeros.filter(num => num > 12);

console.log(mayoresA12); // Salida: [25, 15, 30]
console.log(numeros);    // Salida: [10, 5, 25, 8, 15, 30] (El original no se modifica)
```

---

2. Método Array.prototype.reduce(): Reducción de elementos a un único valor

El método reduce() ejecuta una función **reductora** sobre cada elemento del array, resultando en un único valor de retorno.

- **Sintaxis:** array.reduce(callback(acumulador, elemento, indice, array), valorInicial)
- **Acumulador:** Es el valor que se va acumulando y es el valor de retorno final.
- **Valor Inicial:** (Opcional) El valor con el que se comienza el acumulador. Si se omite, el primer elemento del array se usa como

valor inicial y la iteración comienza en el segundo elemento.

Ejemplo con reduce() (Suma)

JavaScript

```
const precios = [2.5, 5.0, 1.75, 10.0];

// Reducir: Sumar todos los precios
const total = precios.reduce((suma, precioActual) => {
  return suma + precioActual;
}, 0); // El 0 es el valor inicial de la 'suma'

console.log(total); // Salida: 19.25 (2.5 + 5.0 + 1.75 + 10.0)
```

Ejemplo con reduce() (Contador de ocurrencias)

JavaScript

```
const frutas = ['manzana', 'banana', 'manzana', 'pera', 'banana', 'manzana'];

// Reducir: Contar la ocurrencia de cada fruta
const conteo = frutas.reduce((contador, fruta) => {
  contador[fruta] = (contador[fruta] || 0) + 1;
  return contador;
}, {}); // El {} es el valor inicial, un objeto vacío

console.log(conteo); // Salida: { manzana: 3, banana: 2, pera: 1 }
```

⚙️ Manipulación y Creación de Objetos

1. Object.create()

El método estático Object.create() crea un **nuevo objeto**, usando un objeto existente como **prototipo** del recién creado. Esto es clave para la **herencia prototípica** en JavaScript.

Ejemplo con Object.create()

JavaScript

```
// 1. Objeto Prototipo (el "molde")
const animal = {
  moverse: function() {
    console.log(`${this.nombre} se está moviendo.`);
  },
  especie: 'mamífero'
};

// 2. Creación de un nuevo objeto usando 'animal' como prototipo
const perro = Object.create(animal);

// 3. Asignación de propiedades propias
perro.nombre = "Fido";
perro.edad = 3;

// 4. Acceso a métodos y propiedades heredadas
perro.moverse(); // Salida: Fido se está moviendo.
console.log(perro.especie); // Salida: mamífero (Heredado del prototipo)
```

2. Acceso y Modificación de Propiedades

Acción	Sintaxis	Ejemplo
Acceso	Notación de punto: objeto.propiedad	persona.nombre
	Notación de corchetes: objeto['propiedad']	persona['edad'] (Útil si la propiedad es variable o tiene espacios)
Modificación	objeto.propiedad = nuevoValor	persona.edad = 31
Adición	objeto.nuevaPropiedad = valor	persona.ciudad = 'Madrid'
Eliminación	delete objeto.propiedad	delete persona.ciudad

Ejemplo de Manipulación

JavaScript

```
let coche = {
```

```
  marca: 'Toyota',
  modelo: 'Corolla',
  color: 'rojo'
};

// Modificación
coche.color = 'azul';

// Adición
coche.anio = 2020;

// Acceso con corchetes
const clave = 'modelo';
console.log(`El valor de ${clave} es: ${coche[clave]}`); // Salida: El valor de modelo es: Corolla

// Eliminación
delete coche.marca;

console.log(coche); // Salida: { modelo: 'Corolla', color: 'azul', anio: 2020 }
```

3. Métodos Object.keys(), Object.values(), Object.entries()

Estos métodos estáticos proporcionan formas sencillas de obtener las claves, valores o pares clave-valor de un objeto.

Método	Descripción	Resultado

Object.keys(obj)	Devuelve un array de las **claves** (nombres de propiedad).	['a', 'b', 'c']
Object.values(obj)	Devuelve un array de los **valores** de las propiedades.	[1, 2, 3]
Object.entries(obj)	Devuelve un array de arrays [clave, valor].	[['a', 1], ['b', 2], ['c', 3]]

Ejemplo con los métodos

JavaScript

```
const config = {
  tema: 'oscuro',
  notificaciones: true,
  idioma: 'es'
};

// Obtener todas las claves
const claves = Object.keys(config);
```

```
console.log(claves); // Salida: ['tema', 'notificaciones', 'idioma']

// Obtener todos los valores
const valores = Object.values(config);
console.log(valores); // Salida: ['oscuro', true, 'es']

// Obtener pares clave-valor
const pares = Object.entries(config);
console.log(pares);
/*
Salida: [
  ['tema', 'oscuro'],
  ['notificaciones', true],
  ['idioma', 'es']
]
*/
```

⚙️ Métodos y Propiedades Estáticas

Los **métodos estáticos** son funciones que pertenecen a la **clase o constructor** en sí, no a una instancia específica de esa clase. Se invocan directamente en el nombre de la clase, no en un objeto creado a partir de ella.

Concepto Clave: Un método estático no puede acceder directamente a las propiedades no estáticas de una instancia (porque no tiene acceso a un this de instancia).

Ejemplo de Método Estático

JavaScript

```
class Calculadora {
  // Propiedad estática: pertenece a la clase
  static PI = 3.14159;

  // Método estático: se llama en la clase Calculadora.
  static sumar(a, b) {
    return a + b;
  }

  // Método de instancia: se llama en una instancia de Calculadora.
  restar(a, b) {
    return a - b;
  }
}

// Uso de la propiedad estática
console.log(Calculadora.PI); // Salida: 3.14159

// Uso del método estático: se llama directamente en la clase
const resultadoSuma = Calculadora.sumar(10, 5);
console.log(resultadoSuma); // Salida: 15

// Intentar llamar un método estático en una instancia (no funcionará)
// const calc = new Calculadora();
```

```
// calc.sumar(1, 1); // Error: calc.sumar is not a function

// Uso del método de instancia: requiere crear un objeto primero
const calc = new Calculadora();
const resultadoResta = calc.restar(10, 5);
console.log(resultadoResta); // Salida: 5
```

💾 Local Storage (Almacenamiento Local)

localStorage es un mecanismo de almacenamiento web que permite a las aplicaciones web almacenar datos de forma persistente en el navegador del usuario. Los datos no tienen fecha de caducidad (a diferencia de las sessionStorage que se borran al cerrar la pestaña/navegador).

Importante: Solo puede almacenar **cadenas de texto**. Por eso, es fundamental usar JSON.stringify() y JSON.parse() (serialización/deserialización) para guardar y recuperar objetos o arrays.

Método / Propiedad	**Descripción**
localStorage.setItem(clave, valor)	Almacena un par clave/valor. El valor **debe** ser una cadena.
localStorage.getItem(clave)	Recupera el valor asociado a la clave. Devuelve null si la clave no existe.
localStorage.removeItem(clave)	Elimina el par clave/valor del

	almacenamiento.
localStorage.clear()	Elimina **todo** lo almacenado para el dominio actual.
localStorage.length	Propiedad que devuelve el número de entradas almacenadas.

Ejemplo con localStorage

JavaScript

```
const configuracion = {
  darkMode: true,
  ultimaVisita: new Date().toLocaleDateString()
};

// 1. Guardar un objeto (REQUIERE SERIALIZACIÓN)
// Convierte el objeto a una cadena JSON
const configJSON = JSON.stringify(configuracion);
localStorage.setItem('userConfig', configJSON);
console.log('Configuración guardada.');
```

```
// 2. Recuperar el objeto (REQUIERE DESERIALIZACIÓN)
const configGuardadaJSON = localStorage.getItem('userConfig');

if (configGuardadaJSON) {
  // Convierte la cadena JSON de vuelta a un objeto JavaScript
  const configRecuperada = JSON.parse(configGuardadaJSON);

  console.log('Configuración recuperada:');
  console.log(`Modo oscuro: ${configRecuperada.darkMode}`);
  console.log(`Última visita: ${configRecuperada.ultimaVisita}`);

  // 3. Eliminar la clave después de usarla (opcional)
  // localStorage.removeItem('userConfig');
} else {
  console.log('No se encontró configuración.');
}
```

---

1. Manipulación Avanzada de Propiedades: Object.defineProperty()

Descripción 📝

El método estático **Object.defineProperty()** permite agregar una nueva propiedad directamente a un objeto, o modificar una propiedad existente, y obtener un **control preciso** sobre los atributos internos de esa propiedad.

A diferencia de la asignación simple (objeto.propiedad = valor), este método permite definir los siguientes atributos (llamados *descriptores de propiedades*):

Descriptor	Tipo	Descripción
value	*Data*	El valor asociado con la propiedad.
writable	*Data*	Si es false, el valor de la propiedad no puede ser cambiado (read-only).
enumerable	*Config*	Si es false, la propiedad no aparecerá en bucles como for...in ni en Object.keys().

configurable	*Config*	Si es false, el tipo de descriptor no puede ser cambiado (de data a accessor o viceversa), y no puede ser eliminada (delete).
get / set	*Accessor*	Permite definir una función para obtener (get) o establecer (set) el valor de la propiedad, creando una propiedad **calculada**.

Ejemplo 💡

Vamos a crear un objeto config y definir una propiedad API_KEY que sea **de solo lectura** (writable: false) y que **no sea visible** en iteraciones (enumerable: false).

JavaScript

```
const app = {};

// Definir la propiedad 'nombre' de forma normal
```

```
app.nombre = 'MiApp';

// Definir una propiedad 'API_KEY' con Object.defineProperty()
Object.defineProperty(app, 'API_KEY', {
    value: 'abcd-1234-efgh',
    writable: false,       // No se puede reasignar el valor
    enumerable: false,     // No aparece al iterar el objeto
    configurable: false    // No se puede eliminar ni cambiar sus descriptores
});

// --- Intentos de modificación y acceso ---

console.log(app.nombre);     // Salida: MiApp
console.log(app.API_KEY);    // Salida: abcd-1234-efgh (Se puede leer)

// 1. Intento de modificar (falla silenciosamente o lanza error en modo estricto)
app.API_KEY = 'NUEVO_VALOR';
console.log(app.API_KEY);    // Salida: abcd-1234-efgh (El valor no cambió)

// 2. Intento de enumerar las propiedades
console.log(Object.keys(app)); // Salida: ['nombre'] (API_KEY está oculta)

// 3. Intento de eliminar (falla porque configurable es false)
delete app.API_KEY;
console.log(app.API_KEY);    // Salida: abcd-1234-efgh (No se eliminó)
```

---

2. Más Ejemplos con Array.prototype.reduce()

Descripción 📝

El método **reduce()** aplica una función de *callback* (la función reductora) contra un **acumulador** y cada elemento en el array (de izquierda a derecha) para reducirlo a un **único valor**. Es extremadamente versátil y puede usarse para operaciones más allá de la simple suma.

Ejemplo 1: Agrupar Elementos (Clustering) 📦

Se utiliza reduce() para transformar un array plano de objetos en un objeto donde las propiedades son las categorías, y los valores son arrays de objetos que pertenecen a esa categoría.

JavaScript

```
const productos = [
 { nombre: 'Manzana', categoria: 'Fruta' },
 { nombre: 'Lechuga', categoria: 'Verdura' },
 { nombre: 'Plátano', categoria: 'Fruta' },
 { nombre: 'Zanahoria', categoria: 'Verdura' }
];
```

```
// Objetivo: Agrupar por la propiedad 'categoria'
const agrupados = productos.reduce((acumulador, productoActual) => {
  const categoria = productoActual.categoria;

  // Si la categoría no existe en el acumulador, la inicializa como un array vacío
  if (!acumulador[categoria]) {
    acumulador[categoria] = [];
  }

  // Agrega el producto al array de su categoría
  acumulador[categoria].push(productoActual.nombre);

  return acumulador;
}, {}); // Valor inicial: un objeto vacío {}

console.log(agrupados);
/*
Salida: {
  Fruta: [ 'Manzana', 'Plátano' ],
  Verdura: [ 'Lechuga', 'Zanahoria' ]
}
*/
```

Ejemplo 2: Aplanar un Array (Flattening) 🗺️

Se utiliza reduce() para convertir un array que contiene otros arrays (array multidimensional) en un array de una sola dimensión.

JavaScript

```javascript
const colecciones = [
  [1, 2],
  [3, 4, 5],
  [6]
];

// Objetivo: Convertir a [1, 2, 3, 4, 5, 6]
const aplanado = colecciones.reduce((acumulador, arrayActual) => {
  // Concatena el array actual al acumulador
  return acumulador.concat(arrayActual);
}, []); // Valor inicial: un array vacío []

console.log(aplanado); // Salida: [1, 2, 3, 4, 5, 6]

// Nota: Para arrays simples, también se puede usar Array.prototype.flat()
```