Ejercicios Combinados de JavaScript

1. Gestor de Notas con Arrays y Objetos

Teoría

Los arrays y los objetos permiten almacenar y organizar información de forma estructurada. Podemos recorrer arrays con bucles o con métodos como `map`, `filter` o `reduce`, y guardar resultados en objetos para tener un acceso más semántico.

Enunciado

Crea un programa que gestione las notas de varios alumnos:

- Pide al usuario el número de alumnos y sus nombres.
- Por cada alumno, pide 3 notas y guárdalas en un array.
- Calcula la media de cada alumno mediante una función.
- Guarda los resultados en un objeto con la forma `{ nombre: media }`.
- Muestra el alumno con la media más alta y la media general del grupo.

💡 **Ayuda de código:**

```
function calcularMedia(notas) {
  const suma = notas.reduce((acc, n) => acc + n, 0);
  return (suma / notas.length).toFixed(2);
}
```

2. Simulador de Tienda Online

Teoría

Las clases en JavaScript permiten definir estructuras reutilizables. Con métodos y propiedades, se pueden representar objetos del mundo real, como productos o pedidos.

Enunciado

Crea una aplicación que simule una tienda:

- Implementa una clase `Producto` con propiedades `nombre`, `precio` y `stock`.
- Incluye un método `vender(cantidad)` que reste unidades del stock.
- Añade un método estático `buscarPorNombre(lista, texto)` que devuelva coincidencias.
- Permite al usuario añadir productos a un array y simular compras.

💡 **Ayuda de código:**

```
class Producto {
  constructor(nombre, precio, stock) {
    this.nombre = nombre;
```

```
    this.precio = precio;
    this.stock = stock;
  }
  vender(cantidad) {
    if (cantidad <= this.stock) this.stock -= cantidad;
  }
  static buscarPorNombre(lista, texto) {
    return lista.filter(p => p.nombre.toLowerCase().includes(texto.toLowerCase()));
  }
}
```

3. Biblioteca con Filtros Funcionales

Teoría

Las funciones de orden superior como `map`, `filter` y `reduce` permiten procesar colecciones de datos fácilmente. Junto a las clases, son ideales para crear gestores de información.

Enunciado

Crea una clase `Biblioteca` que gestione libros:

- Cada libro tendrá `titulo`, `autor` y `paginas`.
- Implementa los métodos:
  - `agregar(libro)`
  - `listar()`
  - `buscarPorAutor(nombre)`
  - `ordenarPorTitulo()`
  - `totalPaginas()`
- Permite al usuario elegir opciones de un menú interactivo.

💡 **Ayuda de código:**

```
class Libro {
  constructor(titulo, autor, paginas) {
    this.titulo = titulo;
    this.autor = autor;
    this.paginas = paginas;
  }
}
```

4. Cajero Evolucionado con Clases

Teoría

El uso de clases y bucles permite simular procesos reales como las operaciones de un cajero. Los métodos encapsulan el comportamiento asociado a cada cuenta.

Enunciado

Implementa una clase `Cuenta` con:

- Propiedades `titular` y `saldo`.
- Métodos `ingresar(cantidad)`, `retirar(cantidad)` y `mostrarSaldo()`.
- Crea varias cuentas en un array y permite operar sobre ellas con un menú (`do...while`).

💡 **Ayuda de código:**

```
class Cuenta {
  constructor(titular, saldo = 0) {
    this.titular = titular;
    this.saldo = saldo;
  }
  ingresar(x) { this.saldo += x; }
  retirar(x) { if (x <= this.saldo) this.saldo -= x; }
  mostrarSaldo() { console.log(`${this.titular}: ${this.saldo}€`); }
}
```

5. Análisis de Texto con Funciones Compuestas

Teoría

Las funciones compuestas (como `pipe` o `compose`) permiten encadenar operaciones. Es una técnica funcional que facilita procesar datos paso a paso.

Enunciado

Crea un programa que analice un texto:

- Solicita un texto al usuario.
- Define una función compuesta que limpie el texto, lo pase a minúsculas y cuente vocales y consonantes.
- Devuelve un objeto con los resultados `{ vocales, consonantes }`.
- Muestra el análisis por consola.

💡 **Ayuda de código:**

```
const contarLetras = (t) => {
  let v = 0, c = 0;
  for (const ch of t) {
    if (/^[a-záéíóúñ]$/i.test(ch)) {
      if (/[aeiouáéíóú]/i.test(ch)) v++; else c++;
    }
  }
  return { vocales: v, consonantes: c };
};
```

6. Simulador de Dados Orientado a Objetos

Teoría

La programación orientada a objetos permite modelar entidades que interactúan entre sí. En este caso, los dados y el juego son clases relacionadas.

Enunciado

Crea dos clases:

- `Dado` con un método `lanzar()` que devuelve un número aleatorio.
- `JuegoDeDados` que contenga varios dados y un método `jugar(veces)` que calcule la media de todas las tiradas.
- Muestra los resultados finales por consola.

💡 **Ayuda de código:**

```
class Dado {
  lanzar() { return Math.floor(Math.random() * 6) + 1; }
}
```

7. Gestor de Cursos

Teoría

El trabajo con clases y arrays facilita la gestión de entidades relacionadas, como cursos y alumnos, permitiendo cálculos agregados sobre sus datos.

Enunciado

Crea dos clases:

- `Curso` con propiedades `nombre`, `horas` y `alumnos` (array).
- `Alumno` con `nombre` y `nota`.
- Añade métodos para agregar alumnos, calcular la media del curso y mostrar el mejor alumno.

💡 **Ayuda de código:**

```
class Alumno {
  constructor(nombre, nota) {
    this.nombre = nombre;
    this.nota = nota;
  }
}
```

8. Catálogo Multimedia Interactivo

Teoría

La herencia y el polimorfismo permiten que distintas clases compartan métodos con comportamientos adaptados a cada tipo de objeto.

Enunciado

Crea una jerarquía de clases:

- `ElementoMultimedia` con `titulo`, `autor` y `año`.
- `Pelicula` y `Cancion` que sobrescriban el método `mostrar()`.
- Crea una función externa que filtre los elementos anteriores a 2010.
- Muestra todos los resultados en consola.

💡 **Ayuda de código:**

```
class ElementoMultimedia {
  constructor(titulo, autor, año) {
    this.titulo = titulo;
    this.autor = autor;
    this.año = año;
  }
  mostrar() { console.log(`${this.titulo} - ${this.autor}`); }
}
```

9. Sistema de Validación Modular

Teoría

Las funciones de validación modular permiten construir reglas combinables. Usar `and` y `or` mejora la legibilidad y reusabilidad del código.

Enunciado

Crea un conjunto de validadores:

- `minLen(n)`
- `hasDigit()`
- `hasUpper()`
- `and(...rules)` y `or(...rules)`
- Define `isStrong = and(minLen(8), hasDigit(), hasUpper())`
- Permite al usuario probar varias contraseñas hasta escribir "fin".

💡 **Ayuda de código:**

```
const minLen = (n) => (s) => s.length >= n;
const hasDigit = () => (s) => /\d/.test(s);
```

```
const hasUpper = () => (s) => /[A-Z]/.test(s);
```

10. Ranking de Videojuegos

Teoría

El uso de métodos como `filter`, `sort` y `reduce` permite crear estadísticas sobre colecciones de objetos. Ideal para rankings y listados.

Enunciado

Crea una clase `Videojuego` con `titulo`, `plataforma` y `puntuacion`.

- Guarda varios juegos en un array.
- Implementa funciones:
  - `ordenarPorPuntuacion(juegos)`
  - `filtrarPorPlataforma(juegos, plataforma)`
  - `mediaPuntuacion(juegos)`
- Muestra el ranking y la media total por consola.

💡 **Ayuda de código:**

```
function mediaPuntuacion(juegos) {
  return (juegos.reduce((acc, j) => acc + j.puntuacion, 0) / juegos.length).toFixed(2);
}
```