Guía Práctica – Programación Funcional con JavaScript (Node.js)

Arrays, Funciones de Orden Superior, Transformaciones, Reducciones y Composición

Propósito: Desarrollar competencias en programación funcional usando JavaScript en Node.js, desde nivel inicial hasta avanzado. La guía prioriza pensamiento declarativo, funciones puras, inmutabilidad y composición de operaciones sobre colecciones.  
  
Cómo ejecutar: cree un archivo .js (por ejemplo main.js) con el dataset, y ejecute con:  
 node main.js  
o con recarga automática durante la práctica:  
 npx nodemon main.js

# Dataset base (copiar al inicio del archivo)

// =====================  
// Dataset base  
// =====================  
const lista = [4, 1, 2, 4, 5, 8, 7, 6, 9, 10];  
  
const personas = [  
 { nombre: "Ana", edad: 18, pais: "Argentina", sexo: "F" },  
 { nombre: "Luis", edad: 30, pais: "Perú", sexo: "M" },  
 { nombre: "Fernanda", edad: 25, pais: "Argentina", sexo: "F" },  
 { nombre: "Laura", edad: 15, pais: "Chile", sexo: "F" },  
 { nombre: "Veronica", edad: 22, pais: "Chile", sexo: "F" },  
 { nombre: "Miguel", edad: 22, pais: "Chile", sexo: "M" },  
];  
  
const productos = [  
 { id: 1, nombre: "Teclado", categoria: "Periféricos", precio: 30, stock: 20, rating: 4.2 },  
 { id: 2, nombre: "Mouse", categoria: "Periféricos", precio: 20, stock: 50, rating: 3.9 },  
 { id: 3, nombre: "Monitor", categoria: "Pantallas", precio: 150, stock: 8, rating: 4.6 },  
 { id: 4, nombre: "Laptop", categoria: "Computo", precio: 1500,stock: 5, rating: 4.8 },  
 { id: 5, nombre: "Cable HDMI",categoria: "Accesorios", precio: 10, stock: 100,rating: 3.5 },  
];  
  
const ventas = [  
 { idVenta: 101, productoId: 1, unidades: 3, precioUnitario: 30, fecha: "2025-08-01" },  
 { idVenta: 102, productoId: 2, unidades: 5, precioUnitario: 20, fecha: "2025-08-02" },  
 { idVenta: 103, productoId: 3, unidades: 2, precioUnitario: 150, fecha: "2025-08-03" },  
 { idVenta: 104, productoId: 5, unidades: 10,precioUnitario: 10, fecha: "2025-08-04" },  
 { idVenta: 105, productoId: 2, unidades: 1, precioUnitario: 20, fecha: "2025-08-05" },  
];

# A. Conceptos base y sintaxis (nivel inicial)

E1. Escribe tres funciones flecha equivalentes que sumen 2 a un número: (a) con return implícito, (b) con llaves y return explícito, (c) como función tradicional.

// (a) Flecha con return implícito  
const masDosA = n => n + 2;  
// (b) Flecha con llaves y return explícito  
const masDosB = n => { return n + 2; };  
// (c) Función tradicional  
function masDosC(n){ return n + 2; }

E2. Explica con un ejemplo por qué `v => { v > 5 }` falla dentro de un `filter`. Corrige el ejemplo.

// Falla porque con llaves {} NO hay return implícito:  
const mal = lista.filter(v => { v > 5 }); // siempre undefined -> filtra nada  
// Corrección:  
const bien1 = lista.filter(v => v > 5); // implícito  
const bien2 = lista.filter(v => { return v > 5; }); // explícito  
// Pedagogía: "si hay llaves, pon return".

E3. Crea una función pura `esPar(n)` y úsala con `filter` para obtener los pares de `lista`.

// Función pura: mismo input -> mismo output; sin efectos colaterales  
const esPar = n => n % 2 === 0;  
const pares = lista.filter(esPar);

E4. Usa `map` para transformar `lista` en sus dobles. No mutar `lista`.

// map NO muta el original, crea uno nuevo  
const dobles = lista.map(x => x \* 2);

# B. Operaciones esenciales sobre arrays

E5. Con `some`, verifica si `lista` contiene al menos un múltiplo de 3.

// some se detiene en el primero que cumple  
const hayMultiploDe3 = lista.some(x => x % 3 === 0);

E6. Con `every`, verifica si todos los elementos de `lista` son mayores que 0.

// every exige que todos cumplan  
const todosPositivos = lista.every(x => x > 0);

E7. Obtén con `find` la primera persona de Chile.

// find devuelve el primero o undefined  
const primeroChile = personas.find(p => p.pais === "Chile");

E8. Usa `filter` para obtener personas con edad entre 20 y 25 (inclusive).

// 20 <= edad <= 25  
const entre20y25 = personas.filter(p => p.edad >= 20 && p.edad <= 25);

E9. Con `map`, genera un arreglo de strings: `"<nombre> vive en <pais>"`.

// Strings descriptivos  
const frases = personas.map(p => `${p.nombre} vive en ${p.pais}`);

E10. Ordena alfabéticamente (A→Z) los nombres de `personas` con `map` + `sort`. Usar `localeCompare`.

// Orden alfabético por nombre  
const nombresOrden = personas.map(p => p.nombre).sort((a,b) => a.localeCompare(b));

# C. Transformaciones y ordenamiento

E11. Ordena `lista` de forma ascendente SIN mutar el array original.

// No mutar: clonar primero y ordenar el clon  
const asc = [...lista].sort((a,b) => a - b);

### 1) [...lista] — clona el array

* Crea una **copia superficial** de lista.
* Así **no** modificás el original (porque sort **sí** muta el array sobre el que actúa).

### 2) .sort((a, b) => a - b) Comparador numérico ascendente

### “a-b ⇒ asc, b-a ⇒ desc”

El comparador recibe **pares de elementos** y debe devolver:

* **< 0** → a va **antes** que b
* **> 0** → a va **después** de b
* **0** → se mantienen en el mismo orden relativo (en motores modernos, el sort es estable)

Como a - b:

* Si a < b ⇒ a - b es **negativo** ⇒ a antes que b ⇒ **ascendente** ✅
* Si a > b ⇒ a - b es **positivo** ⇒ a después de b
* Si a === b ⇒ 0 ⇒ mantiene el orden relativo

**Con criterio secundario (edad asc, y a igualdad, nombre A→Z):**

const cmp = (p, q) => (p.edad - q.edad) || p.nombre.localeCompare(q.nombre);

const ordenadas = [...personas].sort(cmp);

E12. Ordena `productos` por `precio` descendente y obtén sólo los nombres.

// Ordenar por precio desc y mapear nombres  
const nombresPorPrecioDesc = [...productos]  
 .sort((a,b) => b.precio - a.precio)  
 .map(p => p.nombre);

E13. Filtra `productos` con `stock > 10`, ordénalos por `rating` desc y devuelve `{nombre, rating}`.

// Filtrar por stock>10, ordenar por rating desc y mapear  
const destacados = productos  
 .filter(p => p.stock > 10)  
 .sort((a,b) => b.rating - a.rating)  
 .map(p => ({ nombre: p.nombre, rating: p.rating }));

E14. A partir de `personas`, devuelve los nombres de Chile ordenados por edad ascendente.

// Chile por edad ascendente -> nombres  
const nombresChilePorEdad = personas  
 .filter(p => p.pais === "Chile")  
 .sort((a,b) => a.edad - b.edad)  
 .map(p => p.nombre);

# D. Patrones con reduce (agregaciones)

E15. Suma total de `lista` usando `reduce`.

// Suma con reduce  
const suma = lista.reduce((acc, x) => acc + x, 0);

E16. Máximo y mínimo de `lista` usando `reduce` (en una sola pasada devuelve `{max, min}`).

// Max y Min en una sola pasada  
const extremos = lista.reduce((acc, x) => ({  
 max: acc.max === undefined ? x : Math.max(acc.max, x),  
 min: acc.min === undefined ? x : Math.min(acc.min, x)  
}), {});  
// Resultado: {max: ..., min: ...}

E17. Suma total del valor de inventario (`precio \* stock`) de `productos`.

// Valor de inventario: sumatoria de precio\*stock  
const valorInventario = productos  
 .map(p => p.precio \* p.stock)  
 .reduce((acc, v) => acc + v, 0);

E18. Promedio de edad por país a partir de `personas`. Devuelve objeto `{ pais: promedio }`.

// Promedio de edades por país  
const acum = personas.reduce((acc, p) => {  
 if (!acc[p.pais]) acc[p.pais] = { suma: 0, count: 0 };  
 acc[p.pais].suma += p.edad;  
 acc[p.pais].count += 1;  
 return acc;  
}, {});  
const promedioPorPais = Object.fromEntries(  
 Object.entries(acum).map(([pais, {suma, count}]) => [pais, suma / count])  
);

E19. Construye `{ pais: { F: n, M: n } }` a partir de `personas`.

// Conteo por sexo en cada país  
const sexoPorPais = personas.reduce((acc, p) => {  
 if (!acc[p.pais]) acc[p.pais] = { F: 0, M: 0 };  
 acc[p.pais][p.sexo] += 1;  
 return acc;  
}, {});

E20. Implementa `partition(arr, pred)` con `reduce` que devuelva `[verdaderos, falsos]`. Úsalo para separar adultos (>=18) de menores.

// partition con reduce  
const partition = (arr, pred) => arr.reduce((acc, item) => {  
 (pred(item) ? acc[0] : acc[1]).push(item);  
 return acc;  
}, [[], []]);  
// Uso:  
const [adultos, menores] = partition(personas, p => p.edad >= 18);

# E. Unicidad, mapeos y utilidades HOF

E21. Obtén los valores únicos de `lista` con `reduce` y compáralo con la solución usando `Set`.

// Únicos con reduce  
const unicosReduce = lista.reduce((acc, x) => {  
 if (!acc.includes(x)) acc.push(x);  
 return acc;  
}, []);  
// Únicos con Set (más simple y rápido)  
const unicosSet = [...new Set(lista)];

E22. Implementa `uniqueBy(arr, selector)` que elimine duplicados por clave computada. Prueba con `personas` por `pais`.

// uniqueBy por selector  
const uniqueBy = (arr, selector) => {  
 const vistos = new Set();  
 return arr.reduce((acc, item) => {  
 const key = selector(item);  
 if (!vistos.has(key)) { vistos.add(key); acc.push(item); }  
 return acc;  
 }, []);  
};  
const personasPorPaisUnico = uniqueBy(personas, p => p.pais);

E23. Implementa `groupBy(arr, selector)` y agrupa `personas` por `pais`.

// groupBy por selector  
const groupBy = (arr, selector) => arr.reduce((acc, item) => {  
 const key = selector(item);  
 (acc[key] ||= []).push(item);  
 return acc;  
}, {});  
const personasPorPais = groupBy(personas, p => p.pais);

E24. Implementa `countBy(arr, selector)` para contar elementos por clave (por ejemplo, cantidad de personas por país).

// countBy por selector  
const countBy = (arr, selector) => arr.reduce((acc, item) => {  
 const key = selector(item);  
 acc[key] = (acc[key] || 0) + 1;  
 return acc;  
}, {});  
const conteoPorPais = countBy(personas, p => p.pais);

# F. Composición y pipelines declarativos

E25. Pipeline: de `productos`, quedarse con `precio >= 30`, mapear a `{nombre, total: precio\*stock}` y ordenar desc por `total`.

// Pipeline de negocio sobre productos  
const rankingPorValor = productos  
 .filter(p => p.precio >= 30)  
 .map(p => ({ nombre: p.nombre, total: p.precio \* p.stock }))  
 .sort((a,b) => b.total - a.total);

E26. A partir de `ventas`, calcular el ingreso total (suma de `unidades \* precioUnitario`).

// Ingreso total de ventas  
const ingresoTotal = ventas  
 .map(v => v.unidades \* v.precioUnitario)  
 .reduce((acc, x) => acc + x, 0);

E27. En `ventas`, devolver un objeto `{ productoId: ingreso }` (ingreso por producto).

// Ingreso por productoId  
const ingresoPorProducto = ventas.reduce((acc, v) => {  
 acc[v.productoId] = (acc[v.productoId] || 0) + v.unidades \* v.precioUnitario;  
 return acc;  
}, {});

E28. Cruzar `ventas` y `productos` para obtener `{ nombreProducto, ingreso }` ordenado desc por ingreso.

// Cruce ventas x productos con nombre  
const ingresoPorNombre = Object.entries(ventas.reduce((acc, v) => {  
 acc[v.productoId] = (acc[v.productoId] || 0) + v.unidades \* v.precioUnitario;  
 return acc;  
}, {})).map(([productoId, ingreso]) => {  
 const prod = productos.find(p => p.id === Number(productoId));  
 return { nombreProducto: prod ? prod.nombre : `#${productoId}`, ingreso };  
}).sort((a,b) => b.ingreso - a.ingreso);

# G. Funciones que retornan funciones, currying y predicados

E29. Crea un generador de predicados: `mayorQue(n)` que retorne una función que pruebe `x > n`. Úsalo con `filter` en `lista`.

// Generador de predicados  
const mayorQue = n => x => x > n;  
const mayoresA5 = lista.filter(mayorQue(5));

E30. Crea `conCampo(campo, pred)` que retorne un predicado que aplique `pred` sobre `obj[campo]`. Úsalo para filtrar `personas` con `edad > 20`.

// Predicado por campo  
const conCampo = (campo, pred) => obj => pred(obj[campo]);  
const mayores20 = personas.filter(conCampo("edad", n => n > 20));

E31. Crea `mapearCampo(campo, f)` que devuelva una función que transforme un array de objetos aplicando `f` al `campo`. Prueba con `precio` de `productos`.

// Mapeador por campo  
const mapearCampo = (campo, f) => arr => arr.map(o => ({ ...o, [campo]: f(o[campo]) }));  
const subir10 = x => x + 10;  
const productosConPrecioSubido = mapearCampo("precio", subir10)(productos);

E32. Implementa `compose(f,g)` y `pipe(...fns)` y úsalos para construir una transformación que: (1) filtre `lista` por pares, (2) duplique, (3) sume todo con `reduce`.

// compose y pipe  
const compose = (f, g) => x => f(g(x));  
const pipe = (...fns) => x => fns.reduce((acc, fn) => fn(acc), x);  
  
// Construcción de pipeline: pares -> duplicar -> sumar  
const soloPares = arr => arr.filter(x => x % 2 === 0);  
const duplicar = arr => arr.map(x => x \* 2);  
const sumar = arr => arr.reduce((acc, x) => acc + x, 0);  
  
const total = pipe(soloPares, duplicar, sumar)(lista);

# H. Inmutabilidad y efectos

E33. Demuestra (con código) la diferencia entre mutar y no mutar al agregar un elemento: `push` vs `concat`/spread.

// Mutar vs no mutar  
const arr = [1,2];  
arr.push(3); // MUTACIÓN: arr ahora es [1,2,3]  
  
const arr2 = [1,2];  
const arr3 = arr2.concat(3); // NO muta: arr2 queda [1,2]; arr3 es [1,2,3]  
const arr4 = [...arr2, 3]; // NO muta: sintaxis spread

E34. Escribe una función pura `agregarSiNoExiste(arr, x)` que devuelva un NUEVO array con `x` si no estaba.

// Agregar si no existe (inmutable)  
const agregarSiNoExiste = (arr, x) => arr.includes(x) ? arr : [...arr, x];

E35. Escribe `actualizarPrecio(productos, id, nuevoPrecio)` que devuelva una NUEVA colección con el precio cambiado solo para ese id (sin mutar los demás objetos).

// Actualizar precio sin mutar  
const actualizarPrecio = (productos, id, nuevoPrecio) =>  
 productos.map(p => p.id === id ? { **...p**, precio: nuevoPrecio } : p);

¿Que hace ese …p?

Ese ...p es el **operador de propagación (spread)** aplicado a **objetos**. Sirve para **copiar (de forma superficial)** todas las propiedades enumerables de p dentro de un **objeto nuevo**.

En tu función:

const actualizarPrecio = (productos, id, nuevoPrecio) =>

productos.map(p => p.id === id ? { ...p, precio: nuevoPrecio } : p);

console.log( actualizarPrecio(productos, 2, 9999) );

* { ...p, precio: nuevoPrecio } crea **un objeto nuevo** que:
  1. Primero copia todas las propiedades de p
  2. Luego **agrega/sobrescribe** la propiedad precio con nuevoPrecio

Así lográs **inmutabilidad**: no tocás el objeto original p; generás uno nuevo con el cambio.

### Ejemplo simple

const p = { id: 2, nombre: "Mouse", precio: 20 };

const actualizado = { ...p, precio: 25 };

// actualizado: { id: 2, nombre: "Mouse", precio: 25 }

// p queda intacto: { id: 2, nombre: "Mouse", precio: 20 }

### Orden importa

El último en la lista gana:

{ ...p, precio: 25 } // toma 'precio' = 25

{ precio: 25, ...p } // 'precio' queda como el de p (sobrescribe 25)

### Copia ****superficial**** (shallow copy)

...p **no** hace copia profunda. Si p tiene objetos anidados, sus **referencias** se copian:

const p = { id: 1, info: { garantia: 12 } };

const copia = { ...p };

copia.info.garantia = 24;

console.log(p.info.garantia); // 24 (cambió también)