Ejercicios de Funciones de Orden Superior en JavaScript (con comentarios pedagógicos)

Totalmente válido lo que sentís. A simple vista, algo como f(x)(y) = x + y parece rebuscado. La **clave** es que las funciones que retornan funciones (y, en general, las funciones de orden superior) te permiten **configurar una vez** y **reutilizar muchas veces**, además de **inyectar comportamiento** y **componer** lógica sin duplicar código. Acá van casos reales y cortitos donde sí tienen mucho sentido:

### 1) Configurar una vez (tarifas, impuestos, descuentos)

// Fábrica de funciones: dejo fija la tasa una sola vez

const aplicarIVA = (tasa) => (precio) => Math.round(precio \* (1 + tasa));

const iva21 = aplicarIVA(0.21);

const iva105 = aplicarIVA(0.105);

iva21(100); // 121

iva105(100); // 111

Evita repetir “cómo” se calcula y te deja reusar la versión ya configurada.

### 2) Validaciones reutilizables (formularios, reglas de negocio)

const minLength = (n) => (s) => typeof s === 'string' && s.length >= n;

const maxValue = (n) => (x) => typeof x === 'number' && x <= n;

const nombreOk = minLength(3);

const precioOk = maxValue(1000);

nombreOk("Lea"); // true

precioOk(1200); // fa**lse**

Creamos validadores “a la carta” y los usamos en muchos campos sin repetir lógica.

### 9) Predicados a medida y reutilización en colecciones

const between = (min, max) => (n) => n >= min && n <= max;

[5, 12, 18, 25].filter(between(10, 20)); // [12, 18]

Reusás la misma regla en varios filtros sin duplicar.

######################################

Estos ejercicios progresan desde lo básico a lo avanzado. En cada ejercicio encontrarás el objetivo, una breve consigna y una solución comentada paso a paso. Se asume que trabajarás con Node.js o cualquier entorno que ejecute JavaScript.

## 1) Asignar una función a una variable (lo más básico)

Objetivos:

* - Comprender que las funciones son valores y se pueden asignar a variables.
* - Ejecutar la función almacenada.

Consigna:

Declara una función que salude y asígnala a una variable. Luego invócala desde esa variable.

Solución comentada (JavaScript):

// Paso 1: Declaramos una función tradicional que retorna un saludo  
function saludar(nombre) {  
 return "Hola " + nombre;  
}  
  
// Paso 2: Asignamos la función a una variable (las funciones son valores)  
const miSaludo = saludar;  
  
// Paso 3: Invocamos a través de la variable  
console.log(miSaludo("Lea")); // Esperado: "Hola Lea"

## 2) Pasar una función por parámetro

Objetivos:

* - Entender que una función puede recibir otra función como argumento.
* - Aplicar la función recibida a distintos valores.

Consigna:

Crea una función 'aplicar' que reciba una función y un valor, y devuelva el resultado de ejecutar la función con ese valor.

Solución comentada (JavaScript):

// Paso 1: Definimos una función de orden superior que recibe otra función  
function aplicar(fn, valor) {  
 // Paso 2: Ejecutamos la función recibida con el valor dado  
 return fn(valor);  
}  
  
// Paso 3: Creamos funciones simples de ejemplo  
const doble = (x) => x \* 2;  
const cuadrado = (x) => x \* x;  
  
// Paso 4: Probamos  
console.log(aplicar(doble, 5)); // 10  
console.log(aplicar(cuadrado, 5)); // 25

## 3) Función que retorna otra función (closures básicos)

Objetivos:

* - Practicar la creación de funciones que devuelven funciones.
* - Comprender el cierre (closure) capturando variables del entorno.

Consigna:

Implementa 'sumarA(a)' que retorne una función 'f(b)' tal que f(b) = a + b.

Solución comentada (JavaScript):

// Paso 1: Definimos una función que retorna otra función  
function sumarA(a) {  
 // Paso 2: 'a' queda 'cerrada' (closure) para la función interna  
 return function(b) {  
 return a + b; // Usa 'a' desde el entorno externo  
 };  
}  
  
// Paso 3: Creamos funciones especializadas  
const sumar5 = sumarA(5);  
const sumar10 = sumarA(10);  
  
// Paso 4: Probamos  
console.log(sumar5(3)); // 8  
console.log(sumar10(3)); // 13

## 4) Closure con estado (contador)

Objetivos:

* - Usar closures para mantener estado privado.
* - Crear una función fábrica (factory) de contadores.

Consigna:

Construye 'crearContador()' que devuelva una función que, al llamarla, incremente y retorne un contador interno.

Solución comentada (JavaScript):

// Paso 1: 'crearContador' genera una función con estado privado  
function crearContador() {  
 let count = 0; // Estado interno, no accesible desde fuera  
 // Paso 2: Retornamos una función que manipula ese estado  
 return function() {  
 count = count + 1;  
 return count;  
 };  
}  
  
// Paso 3: Probamos dos contadores independientes  
const c1 = crearContador();  
const c2 = crearContador();  
  
console.log(c1()); // 1  
console.log(c1()); // 2  
console.log(c2()); // 1  
console.log(c1()); // 3

## 5) Currying básico

Objetivos:

* - Aplicar el concepto de currying: transformar f(a, b) en f(a)(b).
* - Reutilizar funciones parcialmente aplicadas.

Consigna:

Escribe 'sumar(a)(b)' y úsala para crear 'sumar2' y 'sumar100'.

Solución comentada (JavaScript):

// Paso 1: Definimos una función curried  
const sumar = (a) => (b) => a + b;  
  
// Paso 2: Creamos versiones especializadas  
const sumar2 = sumar(2);  
const sumar100 = sumar(100);  
  
// Paso 3: Probamos  
console.log(sumar2(5)); // 7  
console.log(sumar100(5)); // 105

## 6) Partial application (aplicación parcial) sin currying explícito

Objetivos:

* - Crear funciones parcialmente aplicadas capturando parámetros.
* - Practicar reutilización con funciones de orden superior.

Consigna:

Implementa 'multiplicarPor(n)' que retorne una función que multiplique por n.

Solución comentada (JavaScript):

// Paso 1: Fábrica de multiplicadores  
function multiplicarPor(n) {  
 // Paso 2: Retornamos una función que recuerda 'n'  
 return function(x) {  
 return x \* n;  
 };  
}  
  
// Paso 3: Usos  
const por3 = multiplicarPor(3);  
const por10 = multiplicarPor(10);  
  
console.log(por3(7)); // 21  
console.log(por10(7)); // 70

## 7) Implementar 'compose' y 'pipe' (composición de funciones)

Objetivos:

* - Componer funciones de derecha a izquierda (compose) y de izquierda a derecha (pipe).
* - Encadenar transformaciones de forma declarativa.

Consigna:

Crea 'compose' y 'pipe' y utilízalos para procesar cadenas y números.

Solución comentada (JavaScript):

// Paso 1: Definimos 'compose' (f(g(x)))  
const compose = (...fns) => (valor) =>  
 fns.reduceRight((acc, fn) => fn(acc), valor);  
  
// Paso 2: Definimos 'pipe' (pipe(f, g)(x) = g(f(x)))  
const pipe = (...fns) => (valor) =>  
 fns.reduce((acc, fn) => fn(acc), valor);  
  
// Paso 3: Funciones simples  
const trim = (s) => s.trim();  
const toUpper = (s) => s.toUpperCase();  
const exclam = (s) => s + "!";  
  
// Paso 4: Probamos composición  
const gritar = compose(exclam, toUpper, trim);  
console.log(gritar(" hola ")); // "HOLA!"  
  
// Paso 5: Probamos pipe  
const gritar2 = pipe(trim, toUpper, exclam);  
console.log(gritar2(" mundo ")); // "MUNDO!"

## 8) Re-crear 'map' (HOF sobre arrays)

Objetivos:

* - Entender internamente cómo funciona 'map'.
* - Trabajar con callbacks que reciben (elemento, índice, array).

Consigna:

Implementa 'miMap(array, fn)' que devuelva un nuevo array con la transformación de 'fn'.

Solución comentada (JavaScript):

// Paso 1: Implementamos nuestra propia versión de 'map'  
function miMap(arr, fn) {  
 const resultado = [];  
 for (let i = 0; i < arr.length; i++) {  
 // Paso 2: Llamamos al callback con (elemento, índice, array)  
 resultado.push(fn(arr[i], i, arr));  
 }  
 return resultado;  
}  
  
// Paso 3: Probamos  
const nums = [1, 2, 3];  
const dobles = miMap(nums, (x) => x \* 2);  
console.log(dobles); // [2, 4, 6]

## 9) Re-crear 'filter' y 'reduce' (HOF sobre arrays)

Objetivos:

* - Implementar versiones propias de 'filter' y 'reduce'.
* - Ver cómo el callback decide si incluir y cómo acumular.

Consigna:

Escribe 'miFilter(arr, pred)' y 'miReduce(arr, reducer, inicial)'.

Solución comentada (JavaScript):

// Paso 1: 'filter' propio  
function miFilter(arr, predicado) {  
 const out = [];  
 for (let i = 0; i < arr.length; i++) {  
 if (predicado(arr[i], i, arr)) {  
 out.push(arr[i]);  
 }  
 }  
 return out;  
}  
  
// Paso 2: 'reduce' propio  
function miReduce(arr, reducer, inicial) {  
 let acc = inicial;  
 for (let i = 0; i < arr.length; i++) {  
 acc = reducer(acc, arr[i], i, arr);  
 }  
 return acc;  
}  
  
// Paso 3: Probamos  
const datos = [5, 12, 3, 20, 7];  
const mayores10 = miFilter(datos, (x) => x > 10); // [12, 20]  
const suma = miReduce(datos, (acc, x) => acc + x, 0); // 47  
  
console.log(mayores10);  
console.log(suma);

## 10) Combinadores de predicados (and/or/not)

Objetivos:

* - Construir funciones de orden superior que compongan predicados.
* - Usar combinadores para filtros complejos.

Consigna:

Implementa 'and(...preds)', 'or(...preds)' y 'not(pred)'.

Solución comentada (JavaScript):

// Paso 1: Combinador lógico AND  
const and = (...preds) => (x) => preds.every((p) => p(x));  
  
// Paso 2: Combinador lógico OR  
const or = (...preds) => (x) => preds.some((p) => p(x));  
  
// Paso 3: Negación  
const not = (pred) => (x) => !pred(x);  
  
// Predicados simples  
const esPar = (n) => n % 2 === 0;  
const mayorQue10 = (n) => n > 10;  
  
// Paso 4: Combinaciones  
const parYMayorQue10 = and(esPar, mayorQue10);  
const imparONoMayorQue10 = or(not(esPar), not(mayorQue10));  
  
console.log(parYMayorQue10(12)); // true  
console.log(imparONoMayorQue10(12)); // false

## 11) Decorador: medir tiempo de ejecución (timeIt)

Objetivos:

* - Crear un decorador que envuelva funciones para medir su rendimiento.
* - Comparar tiempos entre implementaciones.

Consigna:

Implementa 'timeIt(fn)' que devuelva una nueva función que mida el tiempo de ejecución.

Solución comentada (JavaScript):

// Paso 1: 'timeIt' recibe una función y devuelve otra  
function timeIt(fn, etiqueta = "tiempo") {  
 return function(...args) {  
 const inicio = Date.now();  
 const resultado = fn(...args);  
 const fin = Date.now();  
 console.log(`[${etiqueta}] ${fin - inicio} ms`);  
 return resultado;  
 };  
}  
  
// Paso 2: Función de ejemplo 'lenta'  
function sumaLenta(n) {  
 let acc = 0;  
 for (let i = 0; i < n \* 1e5; i++) acc += i;  
 return acc;  
}  
  
// Paso 3: Decoramos y probamos  
const sumaLentaMedida = timeIt(sumaLenta, "sumaLenta");  
sumaLentaMedida(200); // Imprime el tiempo y devuelve el resultado

## 12) Memoización (decorador para funciones puras)

Objetivos:

* - Evitar recomputar resultados de funciones puras costosas.
* - Usar un caché interno cerrado por closure.

Consigna:

Crea 'memoize(fn)' y úsalo con 'fibonacci(n)'.

Solución comentada (JavaScript):

// Paso 1: Implementamos 'memoize'  
function memoize(fn) {  
 const cache = new Map();  
 return function(...args) {  
 const key = JSON.stringify(args);  
 if (cache.has(key)) {  
 return cache.get(key); // Devuelve desde caché  
 }  
 const resultado = fn(...args);  
 cache.set(key, resultado);  
 return resultado;  
 };  
}  
  
// Paso 2: Función costosa (fibonacci recursivo)  
function fib(n) {  
 if (n <= 1) return n;  
 return fib(n - 1) + fib(n - 2);  
}  
  
// Paso 3: Versión memoizada  
const fibMemo = memoize(fib);  
  
console.log(fibMemo(35)); // Mucho más rápido en subsiguientes llamadas  
console.log(fibMemo(35)); // Devuelve desde caché

## 13) HOFs para arrays de objetos: sortBy y groupBy

Objetivos:

* - Crear utilidades reutilizables para ordenar y agrupar.
* - Practicar callbacks que seleccionan claves.

Consigna:

Implementa 'sortBy(arr, selector)' y 'groupBy(arr, keyFn)'.

Solución comentada (JavaScript):

// Paso 1: Ordenar por una clave derivada  
function sortBy(arr, selector) {  
 return [...arr].sort((a, b) => {  
 const ka = selector(a);  
 const kb = selector(b);  
 if (ka < kb) return -1;  
 if (ka > kb) return 1;  
 return 0;  
 });  
}  
  
// Paso 2: Agrupar por clave  
function groupBy(arr, keyFn) {  
 return arr.reduce((acc, item) => {  
 const key = keyFn(item);  
 if (!acc[key]) acc[key] = [];  
 acc[key].push(item);  
 return acc;  
 }, {});  
}  
  
// Paso 3: Prueba  
const personas = [  
 { nombre: "Ana", edad: 30, ciudad: "CABA" },  
 { nombre: "Luis", edad: 25, ciudad: "CABA" },  
 { nombre: "Mara", edad: 30, ciudad: "La Plata" },  
];  
  
console.log(sortBy(personas, (p) => p.edad));  
console.log(groupBy(personas, (p) => p.ciudad));

## 14) Fábricas de validadores (function factories)

Objetivos:

* - Generar funciones personalizadas a partir de parámetros.
* - Componer validadores para entradas de usuario.

Consigna:

Crea 'minLength(n)', 'maxValue(n)' y combínalos con 'and'/'or'.

Solución comentada (JavaScript):

// Paso 1: Fábricas de validadores  
const minLength = (n) => (s) => typeof s === "string" && s.length >= n;  
const maxValue = (n) => (x) => typeof x === "number" && x <= n;  
  
// Paso 2: Reutilizamos combinadores de predicados  
const and = (...preds) => (x) => preds.every((p) => p(x));  
const or = (...preds) => (x) => preds.some((p) => p(x));  
  
// Paso 3: Validaciones  
const esNombreValido = minLength(3);  
const esPrecioAceptable = maxValue(1000);  
  
console.log(esNombreValido("Lea")); // true  
console.log(esPrecioAceptable(1200)); // false  
console.log(and(esNombreValido)("Al")); // false

## 15) asyncMap: versión asíncrona de map con Promise.all

Objetivos:

* - Procesar colecciones de forma asíncrona con promesas.
* - Ejecutar tareas en paralelo y esperar sus resultados.

Consigna:

Implementa 'asyncMap(arr, asyncFn)' que resuelva todas las promesas.

Solución comentada (JavaScript):

// Paso 1: 'asyncMap' aplica una función asíncrona a cada elemento  
async function asyncMap(arr, asyncFn) {  
 // Paso 2: Obtenemos un array de promesas  
 const promesas = arr.map(asyncFn);  
 // Paso 3: Esperamos a que todas se resuelvan  
 return Promise.all(promesas);  
}  
  
// Simulación de tarea asíncrona (p. ej. consulta a API)  
const tareaAsync = (x) => new Promise((res) => {  
 setTimeout(() => res(x \* 2), 200);  
});  
  
// Prueba  
(async () => {  
 const resultados = await asyncMap([1, 2, 3], tareaAsync);  
 console.log(resultados); // [2, 4, 6]  
})();

## 16) Retry: decorador para reintentar funciones asíncronas

Objetivos:

* - Aumentar la robustez reintentando operaciones que pueden fallar.
* - Controlar cantidad de reintentos y retrasos (backoff simple).

Consigna:

Crea 'withRetry(asyncFn, intentos, esperaMs)' que reintente si falla.

Solución comentada (JavaScript):

// Paso 1: Implementamos un decorador asíncrono con reintentos  
function withRetry(asyncFn, intentos = 3, esperaMs = 200) {  
 return async function(...args) {  
 let ultimoError;  
 for (let i = 0; i < intentos; i++) {  
 try {  
 return await asyncFn(...args);  
 } catch (e) {  
 ultimoError = e;  
 await new Promise((r) => setTimeout(r, esperaMs));  
 }  
 }  
 throw ultimoError;  
 };  
}  
  
// Función que a veces falla  
let contador = 0;  
async function tareaInestable() {  
 contador++;  
 if (contador % 2 === 0) {  
 return "Ok en intento " + contador;  
 }  
 throw new Error("Fallo en intento " + contador);  
}  
  
// Probamos con reintentos  
(async () => {  
 const segura = withRetry(tareaInestable, 5, 100);  
 try {  
 const res = await segura();  
 console.log(res);  
 } catch (e) {  
 console.error("No se logró completar:", e.message);  
 }  
})();

## 17) Manejo seguro: safe(fn, fallback)

Objetivos:

* - Envolver funciones para capturar errores sin romper el flujo.
* - Devolver un valor por defecto cuando ocurra una excepción.

Consigna:

Crea 'safe(fn, fallback)' que retorne fallback si fn arroja error.

Solución comentada (JavaScript):

// Paso 1: Decorador 'safe' para capturar errores  
function safe(fn, fallback = null) {  
 return function(...args) {  
 try {  
 return fn(...args);  
 } catch (e) {  
 console.warn("Error capturado:", e.message);  
 return fallback;  
 }  
 };  
}  
  
// Función que puede fallar  
function dividir(a, b) {  
 if (b === 0) throw new Error("División por cero");  
 return a / b;  
}  
  
// Uso seguro  
const dividirSeguro = safe(dividir, Infinity);  
console.log(dividirSeguro(10, 2)); // 5  
console.log(dividirSeguro(10, 0)); // Infinity

## 18) Pipeline de transformación de datos (end-to-end)

Objetivos:

* - Integrar varias funciones de orden superior en un flujo de datos.
* - Practicar composición, map/filter/reduce y utilidades creadas.

Consigna:

Dado un conjunto de productos, filtra activos, mapea a precios con IVA y calcula el total.

Solución comentada (JavaScript):

// Paso 1: Datos de ejemplo  
const productos = [  
 { nombre: "A", precio: 100, activo: true },  
 { nombre: "B", precio: 50, activo: false },  
 { nombre: "C", precio: 200, activo: true },  
];  
  
// Paso 2: Utilidades  
const IVA = 0.21;  
const soloActivos = (p) => p.activo;  
const precioConIVA = (p) => p.precio \* (1 + IVA);  
const sumar = (acc, x) => acc + x;  
  
// Paso 3: Pipeline declarativo  
const totalConIVA = productos  
 .filter(soloActivos) // Filtra  
 .map(precioConIVA) // Transforma  
 .reduce(sumar, 0); // Agrega  
  
console.log(totalConIVA); // 100\*1.21 + 200\*1.21 = 363