Objetivo

Aprender a combinar paradigmas efectivamente: cuándo usar cada uno y cómo integrarlos en proyectos reales.

¿Qué es multiparadigma?

Usar múltiples paradigmas en el mismo proyecto, eligiendo el más adecuado para cada problema específico.

Filosofía: "La herramienta correcta para cada trabajo"

Combinaciones comunes

OO + Funcional (híbrido moderno)

```
// Dominio con OO
class Producto {
  constructor(
    private id: string,
    private nombre: string,
    private precio: number
  ) {}

  // Métodos que devuelven nuevas instancias (inmutable)
  conDescuento(porcentaje: number): Producto {
    return new Producto(
      this.id,
      this.nombre,
      this.precio * (1 - porcentaje / 100)
    );
  }
```

```
  obtenerPrecio(): number { return this.precio; }
  obtenerNombre(): string { return this.nombre; }
}

// Lógica de negocio con FP
const aplicarDescuentos = (productos: Producto[]) =>
  productos
    .filter(p => p.obtenerPrecio() > 100)
    .map(p => p.conDescuento(10));

const calcularTotal = (productos: Producto[]) =>
  productos.reduce((sum, p) => sum + p.obtenerPrecio(), 0);

// Uso combinado
const productos = [
  new Producto("1", "Laptop", 1000),
  new Producto("2", "Mouse", 25)
];

const conDescuento = aplicarDescuentos(productos);
const total = calcularTotal(conDescuento);
```

Imperativo + Declarativo

```
// Imperativo para algoritmos específicos
function busquedaBinaria(arr: number[], target: number): number {
  let left = 0;
  let right = arr.length - 1;

  while (left <= right) {
    const mid = Math.floor((left + right) / 2);
    if (arr[mid] === target) return mid;
    if (arr[mid] < target) left = mid + 1;
    else right = mid - 1;
  }
  return -1;
}

// Declarativo para consultas de datos
const buscarProductos = (productos: Producto[], criterios: any) =>
  productos
    .filter(p => criterios.minPrecio ? p.obtenerPrecio() >= criterios.minPrecio :
true)
    .filter(p => criterios.nombre ? p.obtenerNombre().includes(criterios.nombre) :
true)
    .sort((a, b) => a.obtenerPrecio() - b.obtenerPrecio());
```

Reactivo + Funcional

```typescript
// Simulación de streams reactivos
type Observer<T> = (value: T) => void;

class Observable<T> {
  private observers: Observer<T>[] = [];

  subscribe(observer: Observer<T>): void {
    this.observers.push(observer);
  }

  emit(value: T): void {
    this.observers.forEach(obs => obs(value));
  }

  // Operadores funcionales
  map<U>(fn: (value: T) => U): Observable<U> {
    const mapped = new Observable<U>();
    this.subscribe(value => mapped.emit(fn(value)));
    return mapped;
  }

  filter(predicate: (value: T) => boolean): Observable<T> {
    const filtered = new Observable<T>();
    this.subscribe(value => {
      if (predicate(value)) filtered.emit(value);
    });
    return filtered;
  }
}

// Uso: pipeline reactivo-funcional
const clicks = new Observable<{ x: number; y: number }>();
const validClicks = clicks
  .filter(click => click.x > 0 && click.y > 0)
  .map(click => `Click en (${click.x}, ${click.y})`);

validClicks.subscribe(console.log);
```