



¿Qué es la Programación Orientada a Objetos?

Un paradigma que organiza el código en **objetos** que combinan datos (atributos) y comportamiento (métodos). Modela el mundo real mediante entidades que interactúan entre sí.

Los 4 pilares fundamentales

1) Encapsulación

Ocultar los detalles internos y exponer solo lo necesario.

```
class CuentaBancaria {  
    private saldo: number = 0; // Privado  
    private readonly numeroCuenta: string; // Solo lectura  
  
    constructor(numeroCuenta: string) {  
        this.numeroCuenta = numeroCuenta;  
    }  
  
    // Métodos públicos (interfaz)  
    depositar(cantidad: number): boolean {  
        if (cantidad <= 0) {  
            console.log("Cantidad debe ser positiva");  
            return false;  
        }  
        this.saldo += cantidad;  
        return true;  
    }  
  
    retirar(cantidad: number): boolean {  
        if (!this.puedeRetirar(cantidad)) {
```

```

        console.log("Fondos insuficientes");
        return false;
    }
    this.saldo -= cantidad;
    return true;
}

consultarSaldo(): number {
    return this.saldo;
}

// Método privado (implementación interna)
private puedeRetirar(cantidad: number): boolean {
    return cantidad > 0 && this.saldo >= cantidad;
}
}

const cuenta = new CuentaBancaria("123456");
cuenta.depositar(1000);
cuenta.retirar(200);
console.log(cuenta.consultarSaldo()); // 800
// cuenta.saldo = 5000; // ✗ Error: propiedad privada

```

2) Herencia 🛡️

Crear nuevas clases basadas en clases existentes, reutilizando código.

```

// Clase base
abstract class Vehiculo {
    protected marca: string;
    protected modelo: string;
    protected velocidad: number = 0;

    constructor(marca: string, modelo: string) {
        this.marca = marca;
        this.modelo = modelo;
    }

    acelerar(incremento: number): void {
        this.velocidad += incremento;
        console.log(` ${this.marca} ${this.modelo} acelera a ${this.velocidad} km/h`);
    }

    // Método abstracto: debe implementarse en clases hijas
    abstract obtenerTipo(): string;

    obtenerInfo(): string {
        return `${this.marca} ${this.modelo} (${this.obtenerTipo()})`;
    }
}

// Clases derivadas

```

```

class Coche extends Vehiculo {
    private numeroPuertas: number;

    constructor(marca: string, modelo: string, puertas: number) {
        super(marca, modelo); // Llamar al constructor padre
        this.numeroPuertas = puertas;
    }

    obtenerTipo(): string {
        return `Coche de ${this.numeroPuertas} puertas`;
    }

    // Método específico de Coche
    abrirMaletero(): void {
        console.log("Maletero abierto");
    }
}

class Motocicleta extends Vehiculo {
    private cilindrada: number;

    constructor(marca: string, modelo: string, cilindrada: number) {
        super(marca, modelo);
        this.cilindrada = cilindrada;
    }

    obtenerTipo(): string {
        return `Motocicleta ${this.cilindrada}cc`;
    }

    hacerCaballito(): void {
        console.log("¡Haciendo caballito!");
    }
}

// Uso
const coche = new Coche("Toyota", "Corolla", 4);
const moto = new Motocicleta("Honda", "CBR", 600);

coche.acelerar(50);
moto.acelerar(80);
console.log(coche.obtenerInfo());
console.log(moto.obtenerInfo());

```

3) Polimorfismo 🎭

Objetos de diferentes tipos pueden responder al mismo mensaje de forma distinta.

```

// Interface común
interface Reproducible {
    reproducir(): void;
    pausar(): void;
}

```

```
        obtenerDuracion(): number;
    }

class Audio implements Reproducible {
    constructor(private archivo: string, private duracion: number) {}

    reproducir(): void {
        console.log(`🎵 Reproduciendo audio: ${this.archivo}`);
    }

    pausar(): void {
        console.log("⏸️ Audio pausado");
    }

    obtenerDuracion(): number {
        return this.duracion;
    }
}

class Video implements Reproducible {
    constructor(private archivo: string, private duracion: number) {}

    reproducir(): void {
        console.log(`🎥 Reproduciendo video: ${this.archivo}`);
    }

    pausar(): void {
        console.log("⏸️ Video pausado");
    }

    obtenerDuracion(): number {
        return this.duracion;
    }

    // Método específico de video
    cambiarResolucion(resolucion: string): void {
        console.log(`Resolución cambiada a ${resolucion}`);
    }
}

// Reproductor que funciona con cualquier tipo
class ReproductorMultimedia {
    private playlist: Reproducible[] = [];

    agregar(medio: Reproducible): void {
        this.playlist.push(medio);
    }

    reproducirTodo(): void {
        this.playlist.forEach(medio => {
            medio.reproducir(); // Polimorfismo en acción
            console.log(`Duración: ${medio.obtenerDuracion()}s`);
        });
    }
}
```

```
// Uso
const reproductor = new ReproductorMultimedia();
reproductor.agregar(new Audio("cancion.mp3", 180));
reproductor.agregar(new Video("pelicula.mp4", 7200));
reproductor.reproducirTodo();
```

4) Composición ☰

Preferir "tiene un" sobre "es un". Construir objetos complejos combinando objetos simples.

```
// Componentes
class Motor {
    constructor(private potencia: number, private tipo: string) {}

    arrancar(): void {
        console.log(`Motor ${this.tipo} de ${this.potencia}HP arrancado`);
    }

    obtenerPotencia(): number {
        return this.potencia;
    }
}

class GPS {
    calcularRuta(destino: string): string[] {
        console.log(`Calculando ruta hacia ${destino}`);
        return ["Calle A", "Avenida B", destino];
    }
}

class SistemaAudio {
    private volumen: number = 50;

    reproducirMusica(cancion: string): void {
        console.log(`🎵 Reproduciendo: ${cancion} (Vol: ${this.volumen})`);
    }

    ajustarVolumen(nivel: number): void {
        this.volumen = Math.max(0, Math.min(100, nivel));
    }
}

// Composición: Coche "tiene" Motor, GPS, SistemaAudio
class CocheModerno {
    private motor: Motor;
    private gps: GPS;
    private audio: SistemaAudio;

    constructor(
        marca: string,
        modelo: string,
```

```

        potenciaMotor: number,
        tipoMotor: string
    ) {
    this.motor = new Motor(potenciaMotor, tipoMotor);
    this.gps = new GPS();
    this.audio = new SistemaAudio();
}

encender(): void {
    console.log("🚗 Encendiendo coche...");
    this.motor.arrancar();
}

navegarA(destino: string): void {
    const ruta = this.gps.calcularRuta(destino);
    console.log(`Ruta: ${ruta.join(" → ")}`);
}

ponerMusica(cancion: string): void {
    this.audio.reproducirMusica(cancion);
}

// Delegación: exponer funcionalidad de componentes
ajustarVolumen(nivel: number): void {
    this.audio.ajustarVolumen(nivel);
}
}

// Uso
const coche = new CocheModerno("Tesla", "Model 3", 300, "Eléctrico");
coche.encender();
coche.navegarA("Centro Comercial");
coche.ponerMusica("Bohemian Rhapsody");
coche.ajustarVolumen(75);

```

Ventajas de OOP

- **Modelado natural:** Refleja cómo pensamos sobre el mundo real
- **Reutilización:** Herencia y composición permiten reutilizar código
- **Mantenibilidad:** Cambios localizados en clases específicas
- **Escalabilidad:** Fácil agregar nuevas funcionalidades
- **Encapsulación:** Oculta complejidad y protege datos

Desventajas y cuidados

- **Sobrecarga:** Puede ser excesivo para problemas simples
- **Jerarquías complejas:** Herencia profunda dificulta mantenimiento
- **Acoplamiento:** Objetos muy interdependientes
- **Rendimiento:** Overhead de llamadas a métodos virtuales