Tema 4: Clases y Herencia en PHP 8.4

Programación Orientada a Objetos

PHP 8.4 | Grado Superior | Desarrollo Web

Índice

Introducción a la Programación Orientada a Objetos

¿Qué es la POO?

La **Programación Orientada a Objetos (POO)** es un paradigma de programación que organiza el código en **objetos** que contienen datos (propiedades) y comportamientos (métodos).

A diferencia de la programación procedural, donde el código se organiza en funciones, la POO modela el mundo real mediante objetos que interactúan entre sí.

Paradigmas de Programación

Procedural: Funciones y procedimientos
Orientado a Objetos: Clases y objetos
Funcional: Funciones puras

Conceptos Básicos

- **Clase:** Plantilla o molde para crear objetos
- **Objeto:** Instancia de una clase
- **Instancia:** Objeto específico creado a partir de una clase

Ventajas de la POO

✓ **Reutilización:** El código se puede reutilizar mediante herencia y composición

✓ **Encapsulación:** Los datos y métodos se agrupan en objetos, ocultando la implementación interna

✓ **Modularidad:** El código se organiza en módulos independientes y manejables

✓ **Mantenibilidad:** Es más fácil mantener y actualizar el código

✓ **Escalabilidad:** Facilita el crecimiento de aplicaciones complejas

Ejemplo del Mundo Real

Imagina una **tienda de frutas**:

• **Clase:** "Producto" (plantilla)
• **Objetos:** "Manzana", "Naranja", "Plátano" (instancias específicas)
• **Propiedades:** nombre, precio, stock
• **Métodos:** vender(), reabastecer(), calcularDescuento()

Clases y Objetos

Definición de Clase

Una **clase** es una plantilla que define las propiedades (atributos) y métodos (funciones) que tendrán los objetos creados a partir de ella.

Sintaxis Básica

```php
// Definir una clase
class Producto {
    // Propiedades (atributos)
    public $nombre;
    public $precio;
    public $stock;

    // Métodos (funciones)
    public function mostrarInfo() {
        echo "Producto: " . $this->nombre;
        echo " - Precio: $" . $this->precio;
    }
}
```

Crear Objetos (Instancias)

Un **objeto** es una instancia de una clase. Se crea usando la palabra clave **new**.

```php
// Crear un objeto
$producto1 = new Producto();

// Asignar valores a las propiedades
$producto1->nombre = "Manzana";
$producto1->precio = 1.50;
$producto1->stock = 100;

// Llamar a un método
$producto1->mostrarInfo();
// Salida: Producto: Manzana - Precio: $1.50
```

Múltiples Objetos

```php
// Crear varios objetos de la misma clase
$producto2 = new Producto();
$producto2->nombre = "Pera";
$producto2->precio = 1.20;

$producto3 = new Producto();
$producto3->nombre = "Naranja";
$producto3->precio = 1.80;
```

Propiedades y Métodos

Clase con Propiedades y Métodos

```php
class CuentaBancaria {
    // Propiedades
    public $titular;
    public $saldo;
    public $numeroCuenta;

    // Método para depositar dinero
    public function depositar($cantidad) {
        $this->saldo += $cantidad;
        echo "Depósito de $" . $cantidad . " realizado\n";
    }

    // Método para retirar dinero
    public function retirar($cantidad) {
        if ($this->saldo >= $cantidad) {
            $this->saldo -= $cantidad;
            echo "Retiro de $" . $cantidad . " realizado\n";
        } else {
            echo "Saldo insuficiente\n";
        }
    }

    // Método para consultar saldo
    public function consultarSaldo() {
        return $this->saldo;
    }
}
```

Uso de la Clase

```php
// Crear objeto
$cuenta = new CuentaBancaria();

// Asignar propiedades
$cuenta->titular = "Juan Pérez";
$cuenta->saldo = 1000;
$cuenta->numeroCuenta = "ES123456";

// Usar métodos
$cuenta->depositar(500);
// Salida: Depósito de $500 realizado
$cuenta->retirar(200);
// Salida: Retiro de $200 realizado
$saldoActual = $cuenta->consultarSaldo();
echo "Saldo actual: $" . $saldoActual;
// Salida: Saldo actual: $1300
```

Acceso con $this

La variable **$this** se refiere al objeto actual y permite acceder a sus propiedades y métodos desde dentro de la clase.

Constructores y Destructores

Constructor Tradicional

El **constructor** es un método especial que se ejecuta automáticamente al crear un objeto.

```php
class Producto {
    public $nombre;
    public $precio;
    public $stock;

    // Constructor
    public function __construct(
        $nombre,
        $precio,
        $stock
    ) {
        $this->nombre = $nombre;
        $this->precio = $precio;
        $this->stock = $stock;
    }
}

// Crear objeto con constructor
$producto = new Producto("Manzana", 1.50, 100);
```

Promoted Properties (PHP 8.0+)

PHP 8.0 introduce **promoted properties**, una sintaxis más concisa para declarar propiedades en el constructor.

```php
class Producto {
    // Constructor con promoted properties
    public function __construct(
        public string $nombre,
        public float $precio,
        public int $stock
    ) {
        // Las propiedades se asignan automáticamente
    }

    public function mostrarInfo() {
        echo "{$this->nombre}: ${$this->precio}";
    }
}

// Uso idéntico
$producto = new Producto("Pera", 1.20, 50);
$producto->mostrarInfo();

// Salida: Pera: $1.20
```

Herencia en PHP

Clase Padre (Base)

La **herencia** permite crear clases que heredan propiedades y métodos de otras clases.

```php
// Clase padre
class Vehiculo {
    public function __construct(
        public string $marca,
        public string $modelo,
        public int $año
    ) {}

    public function arrancar() {
        echo "El vehículo está arrancando...\n";
    }

    public function detener() {
        echo "El vehículo se ha detenido.\n";
    }

    public function mostrarInfo() {
        echo "{$this->marca} {$this->modelo} ({$this->año})\n";
    }
}
```

Clase Hija (Derivada)

La clase hija **hereda** todo de la clase padre y puede añadir nuevas propiedades y métodos.

```php
// Clase hija que hereda de Vehiculo
class Coche extends Vehiculo {
    public function __construct(
        string $marca,
        string $modelo,
        int $año,
        public int $numPuertas
    ) {
        // Llamar al constructor padre
        parent::__construct($marca, $modelo, $año);
    }

    // Método específico de Coche
    public function abrirMaletero() {
        echo "Maletero abierto.\n";
    }
}

// Uso
$miCoche = new Coche("Toyota", "Corolla", 2024, 4);
$miCoche->mostrarInfo();  // Heredado
$miCoche->arrancar();     // Heredado
$miCoche->abrirMaletero(); // Propio
```

Property Hooks

✨ Novedad en PHP 8.4

¿Qué son los Property Hooks?

Los property hooks permiten definir lógica personalizada al leer (get) o escribir (set) una propiedad, sin necesidad de métodos getter/setter explícitos.

Sintaxis Básica

```php
class Producto {
    // Propiedad con hook 'get'
    public string $nombre {
        get => strtoupper($this->nombre);
    }

    // Propiedad con hook 'set' para validación
    public float $precio {
        set {
            if ($value < 0) {
                throw new Exception("El precio no puede ser negativo");
            }
            $this->precio = $value;
        }
    }

    public function __construct(
        string $nombre,
        float $precio
    ) {
        $this->nombre = $nombre;
        $this->precio = $precio;
    }
}
```

Uso de Property Hooks

```php
// Crear producto
$producto = new Producto("manzana", 1.50);

// El hook 'get' convierte a mayúsculas automáticamente
echo $producto->nombre;
// Salida: MANZANA
// El hook 'set' valida el precio
$producto->precio = 2.00;  // OK
// Esto lanzará una excepción
$producto->precio = -5;

// Exception: El precio no puede ser negativo
```

Ventajas

✅ Código más limpio y conciso
✅ Validación automática al asignar valores
✅ Transformación de datos al leer propiedades
✅ No necesitas métodos getter/setter manuales

Visibilidad: Public, Private, Protected

public

Acceso desde cualquier lugar

Las propiedades y métodos **public** son accesibles desde cualquier parte del código.

```php
class Producto {
    public $nombre;
    public $precio;

    public function mostrar() {
        echo $this->nombre;
    }
}

$p = new Producto();
$p->nombre = "Manzana";
$p->mostrar(); // ✓ OK
```

✓ Dentro de la clase
✓ Desde objetos externos
✓ Desde clases heredadas

private

Solo dentro de la clase

Las propiedades y métodos **private** solo son accesibles desde dentro de la misma clase.

```php
class Producto {
    private $costo;

    private function calcular() {
        return $this->costo * 1.2;
    }

    public function getPrecio() {
        return $this->calcular();
    }
}

$p = new Producto();
$p->costo; // ✗ Error
```

✓ Dentro de la clase
✗ Desde objetos externos
✗ Desde clases heredadas

protected

Clase y clases heredadas

Las propiedades y métodos **protected** son accesibles desde la clase y sus clases heredadas.

```php
class Producto {
    protected $margen;

    protected function calcular() {
        return $this->margen;
    }
}

class Libro extends Producto {
    public function mostrar() {
        echo $this->calcular(); // ✓ OK
    }
}
```

✓ Dentro de la clase
✗ Desde objetos externos
✓ Desde clases heredadas

Herencia II: Override y Parent

Sobrescribir Métodos (Override)

Las clases hijas pueden **sobrescribir** (override) los métodos heredados para cambiar su comportamiento.

```php
// Clase padre
class Producto {
    public function __construct(
        public string $nombre,
        public float $precio
    ) {}

    public function calcularPrecioFinal() {
        return $this->precio;
    }

    public function mostrarInfo() {
        return "Producto: {$this->nombre}";
    }
}

// Clase hija que sobrescribe métodos
class ProductoDescuento extends Producto {
    public function __construct(
        string $nombre,
        float $precio,
        public float $descuento
    ) {
        parent::__construct($nombre, $precio);
    }

    // Override del método padre
    public function calcularPrecioFinal() {
        return $this->precio * (1 - $this->$descuento);
    }
}
```

Uso de parent::

La palabra clave **parent::** permite acceder a métodos de la clase padre desde la clase hija.

```php
class ProductoConIVA extends Producto {
    public function __construct(
        string $nombre,
        float $precio,
        public float $iva = 0.21
    ) {
        parent::__construct($nombre, $precio);
    }

    public function calcularPrecioFinal() {
        // Llamar al método padre y añadir IVA
        $precioBase = parent::calcularPrecioFinal();
        return $precioBase * (1 + $this->iva);
    }

    public function mostrarInfo() {
        // Reutilizar método padre y añadir info
        $info = parent::mostrarInfo();
        return $info . " (IVA: {$this->iva})";
    }
}

// Uso
$producto = new ProductoConIVA("Manzana", 1.50);
echo $producto->calcularPrecioFinal();

// Salida: 1.815 (1.50 + 21% IVA)
```

Visibilidad Asimétrica

✨ Novedad en PHP 8.4

¿Qué es la Visibilidad Asimétrica?

La **visibilidad asimétrica** permite definir diferentes niveles de acceso para **lectura** y **escritura** de una propiedad.

Por ejemplo: una propiedad puede ser **pública para lectura** pero **privada para escritura**.

Sintaxis

```php
class Producto {
    // Sintaxis: public(set) private// Lectura: public | Escritura: private
    public(set) private float $precio;

    public function __construct(
        string $id,
        float $precio
    ) {
        // ✓ OK (dentro de la clase)
        $this->id = $id;
        $this->precio = $precio;
    }

    public function aplicarDescuento(float $desc) {
        // ✓ OK (modificar dentro de la clase)
        $this->precio *= (1 - $desc);
    }
}
```

Uso de Visibilidad Asimétrica

```php
// Crear producto
$producto = new Producto("P001", 100.00);

// ✓ Lectura permitida (public)
echo $producto->id;
// Salida: P001
echo $producto->precio;
// Salida: 100.00
// ✗ Escritura NO permitida (private)
$producto->precio = 50.00;
// Error: Cannot modify private(set) property
// ✓ Modificar mediante método público
$producto->aplicarDescuento(0.10);
echo $producto->precio;

// Salida: 90.00
```

Ventajas

✅ Propiedades de solo lectura desde fuera
✅ Control total de modificaciones
✅ Código más seguro y predecible

Interfaces

¿Qué son las Interfaces?

Una **interface** es un contrato que define qué métodos debe implementar una clase, pero no cómo se implementan.

Características

- Todos los métodos son **públicos**
- No pueden tener propiedades (solo constantes)
- Una clase puede implementar **múltiples interfaces**
- Se usa la palabra clave **implements**

Definir una Interface

```php
// Definir interface
interface Pagable {
    public function procesarPago(float $monto): bool;
    public function obtenerRecibo(): string;
}

interface Enviable {
    public function calcularEnvio(): float;
    public function obtenerDireccion(): string;
}
```

Implementar Interfaces

```php
// Implementar una interface
class Pedido implements Pagable {
    private float $total;

    public function __construct(float $total) {
        $this->total = $total;
    }

    public function procesarPago(float $monto): bool {
        if ($monto >= $this->total) {
            echo "Pago procesado\n";
            return true;
        }
        return false;
    }

    public function obtenerRecibo(): string {
        return "Recibo: ${$this->total}";
    }
}

// Implementar múltiples interfaces
class PedidoOnline implements Pagable, Enviable {
    public function procesarPago(float $monto): bool { /*... */ }
    public function obtenerRecibo(): string { /*... */ }
    public function calcularEnvio(): float { /*... */ }
    public function obtenerDireccion(): string { /*... */ }
}
```

Clases Abstractas

¿Qué son las Clases Abstractas?

Una **clase abstracta** es una clase que no puede ser instanciada directamente. Sirve como plantilla base para otras clases.

Se declara con **abstract** y puede contener métodos normales y métodos abstractos.

Definir Clase Abstracta

```php
abstract class Producto {
    protected string $nombre;
    protected float $precio;

    // Método normal (con implementación)
    public function __construct(
        string $nombre,
        float $precio
    ) {
        $this->nombre = $nombre;
        $this->precio = $precio;
    }

    // Método abstracto (sin implementación)
    abstract public function calcularPrecioFinal(): float;

    abstract public function obtenerTipo(): string;
}

// ✗ ERROR: No se puede instanciar
$p = new Producto();
```

Implementar Clase Abstracta

```php
// Clase hija que implementa métodos abstractos
class Libro extends Producto {
    public function __construct(
        string $nombre,
        float $precio,
        private float $iva = 0.04
    ) {
        parent::__construct($nombre, $precio);
    }

    // Implementar método abstracto
    public function calcularPrecioFinal(): float {
        return $this->precio * (1 + $this->iva);
    }

    // Implementar método abstracto
    public function obtenerTipo(): string {
        return "Libro";
    }
}

// ✓ OK: Instanciar clase hija
$libro = new Libro("PHP 8.4", 29.99);
echo $libro->calcularPrecioFinal();

// Salida: 31.19
```

Traits

¿Qué son los Traits?

Un **trait** es un mecanismo de reutilización de código que permite compartir métodos entre clases sin usar herencia.

Los traits resuelven el problema de la **herencia múltiple** en PHP.

Características

- Se declaran con **trait**
- Se usan en clases con **use**
- Una clase puede usar **múltiples traits**
- No se pueden instanciar directamente

Definir un Trait

```php
// Definir un traittraitTimestamp {
    publicstring$fechaCreacion;

    public functionmarcarCreacion() {
        $this->fechaCreacion = date('Y-m-d H:i:s');
    }

    public functionobtenerFecha() {
        return$this->fechaCreacion;
    }
}
```

Usar Traits en Clases

```php
// Usar un trait en una claseclassProducto {
    useTimestamp;

    public function__construct(
        publicstring$nombre
    ) {}
}

$p = newProducto("Manzana");
$p->marcarCreacion();
echo$p->obtenerFecha();
// Salida: 2024-11-19 15:30:45// Usar múltiples traitstraitAuditable {
    public functionregistrarCambio() {
        echo"Cambio registrado\n";
    }
}

classUsuario {
    useTimestamp, Auditable;

    public function__construct(
        publicstring$nombre
    ) {}
}

$usuario = newUsuario("Juan");
$usuario->marcarCreacion();

$usuario->registrarCambio();
```

Ejemplo Completo I: Sistema de Productos

Clase Base: Producto

```php
class Producto {
    protected string $nombre;
    protected float $precio;
    protected int $stock;

    public function __construct(
        string $nombre,
        float $precio,
        int $stock
    ) {
        $this->nombre = $nombre;
        $this->precio = $precio;
        $this->stock = $stock;
    }

    public function vender(int $cantidad): bool {
        if ($this->stock >= $cantidad) {
            $this->stock -= $cantidad;
            return true;
        }
        return false;
    }

    public function obtenerInfo(): string {
        return "{$this->nombre} - €{$this->precio}";
    }
}
```

Clase Hija: ProductoFisico

```php
class ProductoFisico extends Producto {
    private float $peso;
    private float $costoEnvio = 5.0;

    public function __construct(
        string $nombre,
        float $precio,
        int $stock,
        float $peso
    ) {
        parent::__construct($nombre, $precio, $stock);
        $this->peso = $peso;
    }

    public function calcularEnvio(): float {
        return $this->costoEnvio + ($this->peso * 0.5);
    }

    public function obtenerInfo(): string {
        $info = parent::obtenerInfo();
        return $info . " (Peso: {$this->peso}kg)";
    }
}

// Uso
$producto = new ProductoFisico("Libro", 29.99, 50, 0.5);
echo $producto->obtenerInfo();

echo "Envío: €" . $producto->calcularEnvio();
```

Ejemplo Completo II: Sistema de Usuarios

Interface + Trait + Clase Abstracta

```php
// Interface
interface Autenticable {
    public function login(string $password): bool;
    public function logout(): void;
}

// Trait
trait Timestamps {
    private DateTime $createdAt;

    public function setTimestamps() {
        $this->createdAt = new DateTime();
    }

    public function getCreatedAt() {
        return $this->createdAt;
    }
}

// Clase abstracta
abstract class Usuario implements Autenticable {
    use Timestamps;

    protected string $nombre;
    protected string $email;
    private string $passwordHash;

    public function __construct(
        string $nombre,
        string $email,
        string $password
    ) {
        $this->nombre = $nombre;
        $this->email = $email;
        $this->passwordHash = password_hash(
            $password,
            PASSWORD_DEFAULT
        );
        $this->setTimestamps();
    }

    public function login(string $password): bool {
```

Clase Hija: Admin

```php
class Admin extends Usuario {
    public function obtenerPermisos(): array {
        return ['crear', 'leer', 'actualizar', 'eliminar'];
    }
}

class Cliente extends Usuario {
    public function obtenerPermisos(): array {
        return ['leer'];
    }
}

// Uso
$admin = new Admin("Juan", "juan@mail.com", "pass123");

if ($admin->login("pass123")) {
    echo "Login exitoso\n";
    print_r($admin->obtenerPermisos());
    $admin->logout();
}
```

Buenas Prácticas en POO

Principios SOLID

S - Single Responsibility
Una clase debe tener una sola responsabilidad

O - Open/Closed
Abierto para extensión, cerrado para modificación

L - Liskov Substitution
Las clases hijas deben poder sustituir a las padre

I - Interface Segregation
Interfaces específicas mejor que generales

D - Dependency Inversion
Depende de abstracciones, no de implementaciones

Encapsulación

```php
// ✗ MAL
class Usuario {
    public $saldo = 0;
}
$u = new Usuario();
$u->saldo = 9999; // Modificación directa

// ✓ BIEN
class Usuario {
    private float $saldo = 0;

    public function depositar(float $monto) {
```

Nombres Descriptivos

```php
// ✗ MAL
class A {
    public function b() {}
}

// ✓ BIEN
class ProductoService {
    public function calcularPrecioFinal() {}
}
```

Cohesión y Acoplamiento

- **Alta cohesión:** Métodos relacionados en la misma clase
- **Bajo acoplamiento:** Clases independientes entre sí

Composición sobre Herencia

Prefiere composición (usar objetos dentro de objetos) en lugar de herencia profunda.

```php
// Composición
class Coche {
    private Motor $motor;

    public function __construct() {
        $this->motor = new Motor();
    }
}
```

Ejercicios Prácticos

Ejercicio 1: Clase Básica

Crea una clase **Vehiculo** con propiedades marca, modelo y año. Incluye un constructor y un método para mostrar la información completa del vehículo.

Ejercicio 2: Herencia Simple

Extiende la clase Vehiculo para crear una clase **Coche** que añada la propiedad numeroPuertas. Sobrescribe el método de información para incluir este dato.

Ejercicio 3: Visibilidad

Crea una clase **CuentaBancaria** con:

- Propiedad privada: saldo
- Métodos públicos: depositar(), retirar(), consultarSaldo()
- Validaciones: no permitir saldo negativo

Ejercicio 4: Property Hooks (PHP 8.4)

Crea una clase **Empleado** con property hooks para:

- Validar que el salario sea mayor a 0
- Convertir el nombre a mayúsculas al leerlo
- Calcular el salario anual automáticamente

Ejercicio 5: Interface

Define una interface **Calculable** con métodos calcularArea() y calcularPerimetro(). Implementa esta interface en clases Rectangulo y Circulo.

Ejercicio 6: Clase Abstracta

Crea una clase abstracta **Figura** con:

- Propiedad protegida: color
- Método abstracto: calcularArea()
- Método normal: obtenerColor()

Crea clases Triangulo y Cuadrado que hereden de Figura.

Ejercicio 7: Trait

Crea un trait **Timestamp** que añada propiedades fechaCreacion y fechaModificacion, con métodos para actualizarlas. Úsalo en una clase Articulo.

Ejercicio 8: Sistema Completo

Diseña un sistema de biblioteca con:

- Clase abstracta: Material (titulo, autor)
- Clases hijas: Libro, Revista, DVD
- Interface: Prestable (prestar(), devolver())
- Trait: Valorable (añadir/obtener valoración)
- Implementa todo el sistema completo