Introducción a la Programación Orientada a Objetos en PHP

Por: DANIEL JESÚS CASTILLO BOTERO

Objetivos de Aprendizaje

- Comprender los conceptos fundamentales de la Programación Orientada a Objetos (POO)
- Aprender a implementar clases y objetos en PHP
- Entender los pilares de la POO: encapsulamiento, herencia, polimorfismo y abstracción
- Dominar las relaciones entre objetos: agregación y composición
- Aplicar conceptos de POO en ejemplos prácticos usando PHP

1. Conceptos Fundamentales de la Programación Orientada a Objetos

¿Qué es la Programación Orientada a Objetos?

La Programación Orientada a Objetos (POO) es un paradigma de programación que organiza el diseño de software en torno a objetos en lugar de funciones y lógica. Un objeto es una entidad que contiene datos (propiedades o atributos) y código (métodos o funciones).

Podemos entender un objeto comparándolo con algo del mundo real. Por ejemplo, un coche tiene:

- **Propiedades**: color, marca, modelo, velocidad actual, nivel de combustible
- **Métodos**: arrancar, acelerar, frenar, girar, detenerse

En lugar de pensar en un programa como una secuencia de instrucciones, lo vemos como una colección de objetos que interactúan entre sí para resolver problemas.

Comparación con la Programación Procedural

Para entender mejor el cambio de paradigma, veamos una comparación:

Programación Procedural:

```php
// Enfoque procedural para gestionar información de estudiantes
function crearEstudiante($nombre, $edad, $calificaciones) {
    return [
        'nombre' => $nombre,
        'edad' => $edad,
        'calificaciones' => $calificaciones
    ];
}

function calcularPromedio($estudiante) {
    return array_sum($estudiante['calificaciones']) /
           count($estudiante['calificaciones']);
```

```php
}

function mostrarInformacion($estudiante) {
    echo "Nombre: " . $estudiante['nombre'] . "\n";
    echo "Edad: " . $estudiante['edad'] . " años\n";
    echo "Promedio: " . calcularPromedio($estudiante) . "\n";
}

// Uso
$estudiante1 = crearEstudiante("Ana García", 20, [85, 90, 78]);
mostrarInformacion($estudiante1);
```

Programación Orientada a Objetos:

```php
// Enfoque orientado a objetos para gestionar información de estudiantes
class Estudiante {
    // Propiedades
    public $nombre;
    public $edad;
    public $calificaciones;

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

    // Métodos
    public function calcularPromedio() {
        return array_sum($this->calificaciones) / count($this->calificaciones);
    }

    public function mostrarInformacion() {
        echo "Nombre: " . $this->nombre . "\n";
        echo "Edad: " . $this->edad . " años\n";
        echo "Promedio: " . $this->calcularPromedio() . "\n";
    }
}

// Uso
$estudiante1 = new Estudiante("Ana García", 20, [85, 90, 78]);
$estudiante1->mostrarInformacion();
```

Ventajas de la Programación Orientada a Objetos

1. **Modularidad**: El código se divide en unidades independientes (objetos) que pueden desarrollarse, probarse y mantenerse por separado.

2. **Reutilización**: Las clases pueden reutilizarse en diferentes partes del programa o incluso en diferentes proyectos.
3. **Mantenibilidad**: Es más fácil detectar y corregir errores cuando el código está organizado en objetos con responsabilidades claras.
4. **Extensibilidad**: Es posible extender la funcionalidad existente sin modificar el código original (a través de la herencia).
5. **Encapsulamiento**: Los detalles de implementación se ocultan, exponiendo solo lo necesario para interactuar con el objeto.
6. **Modelado del mundo real**: La POO permite modelar conceptos del mundo real de forma más intuitiva, lo que facilita la comprensión del código.

2. Clases y Objetos en PHP

Clases: Los Planos para Crear Objetos

Una clase es como un plano o plantilla que define las propiedades y comportamientos que tendrán los objetos creados a partir de ella. En PHP, se definen usando la palabra clave `class`.

```php
class Automovil {
    // Propiedades (variables de la clase)
    public $marca;
    public $modelo;
    public $color;
    public $velocidad = 0;

    // Métodos (funciones de la clase)
    public function arrancar() {
        echo "El automóvil ha arrancado\n";
    }

    public function acelerar($incremento) {
        $this->velocidad += $incremento;
        echo "Acelerando. Velocidad actual: {$this->velocidad} km/h\n";
    }

    public function frenar($decremento) {
        if ($this->velocidad - $decremento < 0) {
            $this->velocidad = 0;
        } else {
            $this->velocidad -= $decremento;
        }
        echo "Frenando. Velocidad actual: {$this->velocidad} km/h\n";
    }
}
```

Anatomía de una Clase:

- **Declaración**: Comienza con la palabra clave `class` seguida del nombre de la clase (por convención, en CamelCase iniciando con mayúscula).

- **Propiedades**: Variables que almacenan datos relacionados con la clase.
- **Métodos**: Funciones que definen el comportamiento de los objetos.
- **$this**: Referencia especial al objeto actual. Se usa para acceder a propiedades y métodos dentro de la clase.

Creación de Objetos (Instanciación)

Un objeto es una instancia de una clase. Para crear un objeto, usamos la palabra clave `new` seguida del nombre de la clase.

```
// Crear un objeto de la clase Automovil
$miAuto = new Automovil();

// Asignar valores a las propiedades
$miAuto->marca = "Toyota";
$miAuto->modelo = "Corolla";
$miAuto->color = "Azul";

// Llamar a los métodos
$miAuto->arrancar();
$miAuto->acelerar(20);
$miAuto->acelerar(30);
$miAuto->frenar(15);

// Crear otro objeto de la misma clase
$otroAuto = new Automovil();
$otroAuto->marca = "Honda";
$otroAuto->modelo = "Civic";
$otroAuto->color = "Rojo";

// Cada objeto tiene su propio estado
$otroAuto->arrancar();
$otroAuto->acelerar(25);
```

Resultado:

```
El automóvil ha arrancado
Acelerando. Velocidad actual: 20 km/h
Acelerando. Velocidad actual: 50 km/h
Frenando. Velocidad actual: 35 km/h
El automóvil ha arrancado
Acelerando. Velocidad actual: 25 km/h
```

Observe cómo cada objeto mantiene su propio estado (valores de propiedades) independientemente de otros objetos creados a partir de la misma clase.

El Constructor: Inicializando Objetos

El método `__construct()` es un método especial que se ejecuta automáticamente cuando se crea un objeto. Se utiliza para inicializar propiedades y realizar configuraciones iniciales.

```
class Automovil {
    public $marca;
    public $modelo;
    public $color;
    public $velocidad = 0;

    // Constructor
    public function __construct($marca, $modelo, $color) {
        $this->marca = $marca;
        $this->modelo = $modelo;
        $this->color = $color;
        echo "Se ha creado un {$this->marca} {$this->modelo} de color {$this-
>color}\n";
    }

    // Otros métodos...
    public function arrancar() {
        echo "El {$this->marca} {$this->modelo} ha arrancado\n";
    }

    public function mostrarInformacion() {
        echo "Automóvil: {$this->marca} {$this->modelo}\n";
        echo "Color: {$this->color}\n";
        echo "Velocidad actual: {$this->velocidad} km/h\n";
    }
}

// Ahora podemos crear objetos pasando parámetros al constructor
$miAuto = new Automovil("Toyota", "Corolla", "Azul");
$miAuto->mostrarInformacion();
$miAuto->arrancar();

$otroAuto = new Automovil("Honda", "Civic", "Rojo");
$otroAuto->arrancar();
```

Resultado:

```
Se ha creado un Toyota Corolla de color Azul
Automóvil: Toyota Corolla
Color: Azul
Velocidad actual: 0 km/h
El Toyota Corolla ha arrancado
Se ha creado un Honda Civic de color Rojo
El Honda Civic ha arrancado
```

3. Encapsulamiento: Protegiendo los Datos

El encapsulamiento es uno de los pilares fundamentales de la POO. Consiste en ocultar los detalles internos de una clase y solo exponer lo necesario. Esto se logra mediante los modificadores de acceso.

Modificadores de Acceso

PHP ofrece tres modificadores de acceso:

1. **public**: Las propiedades y métodos son accesibles desde cualquier lugar, dentro y fuera de la clase.
2. **private**: Las propiedades y métodos solo son accesibles dentro de la clase que los define.
3. **protected**: Las propiedades y métodos son accesibles dentro de la clase que los define y sus clases descendientes (herencia).

```php
class CuentaBancaria {
    // Propiedades privadas
    private $numeroCuenta;
    private $saldo;
    private $propietario;

    // Constructor
    public function __construct($propietario, $saldoInicial = 0) {
        $this->propietario = $propietario;
        $this->saldo = $saldoInicial;
        $this->numeroCuenta = $this->generarNumeroCuenta();
    }

    // Método privado
    private function generarNumeroCuenta() {
        // En un caso real, esto sería un algoritmo más complejo
        return "CTA-" . rand(10000, 99999);
    }

    // Métodos públicos (interfaz para interactuar con la clase)
    public function depositar($monto) {
        if ($monto <= 0) {
            echo "Error: El monto debe ser positivo\n";
            return false;
        }
        $this->saldo += $monto;
        echo "Depósito de ${monto} realizado. Nuevo saldo: {$this->saldo}\n";
        return true;
    }

    public function retirar($monto) {
        if ($monto <= 0) {
            echo "Error: El monto debe ser positivo\n";
            return false;
        }
        if ($monto > $this->saldo) {
            echo "Error: Fondos insuficientes\n";
```

```
            return false;
        }
        $this->saldo -= $monto;
        echo "Retiro de ${monto} realizado. Nuevo saldo: {$this->saldo}\n";
        return true;
    }

    public function consultarSaldo() {
        return $this->saldo;
    }

    public function obtenerDetalles() {
        return [
            'propietario' => $this->propietario,
            'numeroCuenta' => $this->numeroCuenta,
            'saldo' => $this->saldo
        ];
    }
}

// Uso de la clase
$cuenta = new CuentaBancaria("María López", 1000);
$detalles = $cuenta->obtenerDetalles();
echo "Cuenta de {$detalles['propietario']}\n";
echo "Número: {$detalles['numeroCuenta']}\n";
echo "Saldo inicial: {$detalles['saldo']}\n";

$cuenta->depositar(500);
$cuenta->retirar(200);
echo "Saldo final: " . $cuenta->consultarSaldo() . "\n";

// Esto generaría un error pues $saldo es privado
// echo $cuenta->saldo;

// Esto generaría un error pues generarNumeroCuenta() es privado
// $cuenta->generarNumeroCuenta();
```

Getters y Setters

Para proporcionar un acceso controlado a propiedades privadas, se utilizan métodos especiales llamados getters y setters:

- **Getters**: Métodos que permiten obtener el valor de una propiedad privada.
- **Setters**: Métodos que permiten modificar el valor de una propiedad privada, a menudo con validaciones.

```
class Persona {
    private $nombre;
    private $edad;
    private $email;
```

```php
    public function __construct($nombre, $edad, $email) {
        $this->setNombre($nombre);
        $this->setEdad($edad);
        $this->setEmail($email);
    }

    // Getters
    public function getNombre() {
        return $this->nombre;
    }

    public function getEdad() {
        return $this->edad;
    }

    public function getEmail() {
        return $this->email;
    }

    // Setters
    public function setNombre($nombre) {
        if (empty($nombre)) {
            throw new Exception("El nombre no puede estar vacío");
        }
        $this->nombre = $nombre;
    }

    public function setEdad($edad) {
        $edad = intval($edad);
        if ($edad < 0 || $edad > 120) {
            throw new Exception("La edad debe estar entre 0 y 120");
        }
        $this->edad = $edad;
    }

    public function setEmail($email) {
        if (!filter_var($email, FILTER_VALIDATE_EMAIL)) {
            throw new Exception("Email inválido");
        }
        $this->email = $email;
    }
}

// Uso
try {
    $persona = new Persona("Juan Pérez", 30, "juan@ejemplo.com");
    echo "Nombre: " . $persona->getNombre() . "\n";
    echo "Edad: " . $persona->getEdad() . "\n";
    echo "Email: " . $persona->getEmail() . "\n";

    // Cambiar el email
    $persona->setEmail("nuevo.email@ejemplo.com");
```

```
    echo "Nuevo email: " . $persona->getEmail() . "\n";

    // Esto generaría una excepción
    // $persona->setEdad(150);
} catch (Exception $e) {
    echo "Error: " . $e->getMessage();
}
```

Importancia del Encapsulamiento

1. **Control de acceso**: Los datos se acceden y modifican de manera controlada.
2. **Validación**: Se pueden aplicar reglas de validación en los setters.
3. **Abstracción**: Los usuarios de la clase no necesitan conocer los detalles internos.
4. **Flexibilidad**: La implementación interna puede cambiar sin afectar a los usuarios de la clase.
5. **Mantenibilidad**: El código es más fácil de mantener y depurar.

4. Herencia: Extendiendo la Funcionalidad

La herencia es un mecanismo que permite a una clase adquirir propiedades y métodos de otra clase. Esto facilita la reutilización de código y el establecimiento de jerarquías de clases.

Creando Clases Derivadas

En PHP, la herencia se implementa utilizando la palabra clave `extends`.

```
// Clase base o padre
class Vehiculo {
    protected $marca;
    protected $modelo;
    protected $año;
    protected $encendido = false;

    public function __construct($marca, $modelo, $año) {
        $this->marca = $marca;
        $this->modelo = $modelo;
        $this->año = $año;
    }

    public function encender() {
        $this->encendido = true;
        echo "El vehículo ha sido encendido\n";
    }

    public function apagar() {
        $this->encendido = false;
        echo "El vehículo ha sido apagado\n";
    }

    public function obtenerInformacion() {
```

```
        return "{$this->marca} {$this->modelo} ({$this->año})";
    }
}

// Clase derivada o hija
class Automovil extends Vehiculo {
    private $puertas;
    private $tipo;

    public function __construct($marca, $modelo, $año, $puertas, $tipo) {
        // Llamar al constructor de la clase padre
        parent::__construct($marca, $modelo, $año);
        $this->puertas = $puertas;
        $this->tipo = $tipo;
    }

    public function obtenerInformacion() {
        // Sobrescribe el método de la clase padre
        $infoBase = parent::obtenerInformacion();
        return "$infoBase - Automóvil {$this->tipo} de {$this->puertas}
puertas";
    }

    public function abrirMaletero() {
        echo "Maletero abierto\n";
    }
}

// Otra clase derivada
class Motocicleta extends Vehiculo {
    private $cilindrada;

    public function __construct($marca, $modelo, $año, $cilindrada) {
        parent::__construct($marca, $modelo, $año);
        $this->cilindrada = $cilindrada;
    }

    public function obtenerInformacion() {
        $infoBase = parent::obtenerInformacion();
        return "$infoBase - Motocicleta de {$this->cilindrada}cc";
    }

    public function hacerCaballito() {
        if ($this->encendido) {
            echo "¡Haciendo un caballito!\n";
        } else {
            echo "No se puede hacer un caballito con la moto apagada\n";
        }
    }
}

// Uso
$auto = new Automovil("Toyota", "Corolla", 2023, 4, "Sedán");
```

```php
echo $auto->obtenerInformacion() . "\n";
$auto->encender();
$auto->abrirMaletero();
$auto->apagar();

echo "\n";

$moto = new Motocicleta("Honda", "CBR", 2023, 600);
echo $moto->obtenerInformacion() . "\n";
$moto->hacerCaballito(); // No se puede porque está apagada
$moto->encender();
$moto->hacerCaballito();
```

Uso de parent

En el ejemplo anterior, hemos utilizado `parent::` para:

1. Llamar al constructor de la clase padre desde el constructor de la clase hija.
2. Llamar al método de la clase padre que estamos sobrescribiendo para extender su funcionalidad.

5. Polimorfismo: Diferentes Formas, Misma Interfaz

El polimorfismo permite que objetos de diferentes clases sean tratados como objetos de una clase común. En PHP, esto se logra principalmente a través de la herencia y las interfaces.

Polimorfismo a través de Herencia

```php
// Clase base
class Forma {
    protected $nombre;

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

    public function dibujar() {
        echo "Dibujando una forma genérica\n";
    }

    public function obtenerNombre() {
        return $this->nombre;
    }
}

// Clases derivadas
class Circulo extends Forma {
    private $radio;

    public function __construct($radio) {
        parent::__construct("Círculo");
```

```
        $this->radio = $radio;
    }

    public function dibujar() {
        echo "Dibujando un círculo de radio {$this->radio}\n";
    }

    public function calcularArea() {
        return pi() * pow($this->radio, 2);
    }
}

class Rectangulo extends Forma {
    private $ancho;
    private $alto;

    public function __construct($ancho, $alto) {
        parent::__construct("Rectángulo");
        $this->ancho = $ancho;
        $this->alto = $alto;
    }

    public function dibujar() {
        echo "Dibujando un rectángulo de {$this->ancho}x{$this->alto}\n";
    }

    public function calcularArea() {
        return $this->ancho * $this->alto;
    }
}

// Función que acepta cualquier objeto de la clase Forma
function dibujarForma(Forma $forma) {
    echo "Vamos a dibujar un " . $forma->obtenerNombre() . "\n";
    $forma->dibujar();
}

// Uso
$circulo = new Circulo(5);
$rectangulo = new Rectangulo(4, 6);

dibujarForma($circulo);
dibujarForma($rectangulo);

// También podemos usar polimorfismo con arrays
$formas = [$circulo, $rectangulo];
foreach ($formas as $forma) {
    echo $forma->obtenerNombre() . ": ";
    echo "Área = " . $forma->calcularArea() . "\n";
}
```

Interfaces: Definiendo Contratos

Las interfaces definen un contrato que las clases deben cumplir. Especifican qué métodos debe implementar una clase, pero no cómo implementarlos.

```php
// Definir una interfaz
interface Dibujable {
    public function dibujar();
    public function obtenerColor();
}

interface Calculable {
    public function calcularArea();
    public function calcularPerimetro();
}

// Implementar múltiples interfaces
class Triangulo implements Dibujable, Calculable {
    private $base;
    private $altura;
    private $color;

    public function __construct($base, $altura, $color) {
        $this->base = $base;
        $this->altura = $altura;
        $this->color = $color;
    }

    // Implementación de Dibujable
    public function dibujar() {
        echo "Dibujando un triángulo {$this->color} de base {$this->base} y
altura {$this->altura}\n";
    }

    public function obtenerColor() {
        return $this->color;
    }

    // Implementación de Calculable
    public function calcularArea() {
        return ($this->base * $this->altura) / 2;
    }

    public function calcularPerimetro() {
        // Para simplificar, asumamos que es un triángulo rectángulo
        $hipotenusa = sqrt(pow($this->base, 2) + pow($this->altura, 2));
        return $this->base + $this->altura + $hipotenusa;
    }
}

// Función que espera un objeto Dibujable
```

```php
function dibujarObjeto(Dibujable $objeto) {
    echo "Dibujando objeto de color " . $objeto->obtenerColor() . ":\n";
    $objeto->dibujar();
}

// Función que espera un objeto Calculable
function mostrarCalculos(Calculable $objeto) {
    echo "Área: " . $objeto->calcularArea() . "\n";
    echo "Perímetro: " . $objeto->calcularPerimetro() . "\n";
}

// Uso
$triangulo = new Triangulo(5, 8, "Rojo");
dibujarObjeto($triangulo);
mostrarCalculos($triangulo);
```

6. Abstracción: Clases e Interfaces Abstractas

La abstracción consiste en identificar las características y comportamientos esenciales de un objeto y ocultar la complejidad de la implementación.

Clases Abstractas

Una clase abstracta es una clase que no se puede instanciar directamente y que puede contener métodos abstractos (métodos sin implementación que deben ser implementados por las clases hijas).

```php
// Clase abstracta
abstract class Animal {
    protected $nombre;

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

    // Método concreto (con implementación)
    public function presentarse() {
        echo "Soy {$this->nombre}, un " . $this->obtenerEspecie() . "\n";
    }

    // Métodos abstractos (sin implementación)
    abstract public function hacerSonido();
    abstract public function obtenerEspecie();
}

class Perro extends Animal {
    private $raza;

    public function __construct($nombre, $raza) {
        parent::__construct($nombre);
        $this->raza = $raza;
```

```
    }

    public function hacerSonido() {
        echo "¡Guau guau!\n";
    }

    public function obtenerEspecie() {
        return "perro de raza {$this->raza}";
    }

    public function moverCola() {
        echo "{$this->nombre} está moviendo la cola\n";
    }
}

class Gato extends Animal {
    public function hacerSonido() {
        echo "¡Miau miau!\n";
    }

    public function obtenerEspecie() {
        return "gato";
    }

    public function ronronear() {
        echo "{$this->nombre} está ronroneando\n";
    }
}

// Uso
$perro = new Perro("Rex", "Pastor Alemán");
$gato = new Gato("Whiskers");

$perro->presentarse();
$perro->hacerSonido();
$perro->moverCola();

echo "\n";

$gato->presentarse();
$gato->hacerSonido();
$gato->ronronear();

// Esto daría un error: No se pueden instanciar clases abstractas
// $animal = new Animal("Genérico");
```

7. Relaciones entre Objetos: Agregación y Composición

Las relaciones entre objetos son fundamentales en POO para modelar sistemas complejos. Dos tipos importantes de relaciones son la agregación y la composición, que representan relaciones "tiene-un" (has-a) entre clases.

Agregación: "Tiene-un" con Independencia

La agregación representa una relación "tiene-un" donde los objetos pueden existir independientemente. Es una relación más débil donde el objeto contenido puede existir sin el contenedor.

Características de la agregación:

- Los objetos componentes pueden existir independientemente del objeto contenedor
- Los componentes pueden ser compartidos por múltiples objetos contenedores
- Si el contenedor se destruye, los componentes pueden continuar existiendo

```php
class Estudiante {
    private $nombre;
    private $id;

    public function __construct($nombre, $id) {
        $this->nombre = $nombre;
        $this->id = $id;
    }

    public function getNombre() {
        return $this->nombre;
    }

    public function getId() {
        return $this->id;
    }

    public function __toString() {
        return "{$this->nombre} (ID: {$this->id})";
    }
}

class Profesor {
    private $nombre;
    private $especialidad;

    public function __construct($nombre, $especialidad) {
        $this->nombre = $nombre;
        $this->especialidad = $especialidad;
    }

    public function getNombre() {
        return $this->nombre;
    }

    public function getEspecialidad() {
        return $this->especialidad;
    }

    public function __toString() {
```

```php
        return "Prof. {$this->nombre} ({$this->especialidad})";
    }
}

// Agregación: El aula "tiene" estudiantes y un profesor
class Aula {
    private $numeroAula;
    private $estudiantes = [];
    private $profesor;

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

    // Agregar estudiante existente al aula
    public function agregarEstudiante(Estudiante $estudiante) {
        $this->estudiantes[] = $estudiante;
        echo "Estudiante {$estudiante->getNombre()} agregado al aula {$this-
>numeroAula}\n";
    }

    // Asignar profesor existente al aula
    public function asignarProfesor(Profesor $profesor) {
        $this->profesor = $profesor;
        echo "Profesor {$profesor->getNombre()} asignado al aula {$this-
>numeroAula}\n";
    }

    public function removerEstudiante($idEstudiante) {
        foreach ($this->estudiantes as $index => $estudiante) {
            if ($estudiante->getId() == $idEstudiante) {
                $nombreEstudiante = $estudiante->getNombre();
                unset($this->estudiantes[$index]);
                echo "Estudiante {$nombreEstudiante} removido del aula {$this-
>numeroAula}\n";
                return;
            }
        }
    }

    public function mostrarInformacion() {
        echo "\n=== Aula {$this->numeroAula} ===\n";
        echo "Profesor: " . ($this->profesor ? $this->profesor : "Sin asignar")
. "\n";
        echo "Estudiantes (" . count($this->estudiantes) . "):\n";
        foreach ($this->estudiantes as $estudiante) {
            echo "  - {$estudiante}\n";
        }
        echo "\n";
    }
}

// Ejemplo de uso de agregación
```

```php
$estudiante1 = new Estudiante("Ana García", 101);
$estudiante2 = new Estudiante("Carlos López", 102);
$estudiante3 = new Estudiante("María Rodríguez", 103);

$profesor1 = new Profesor("Dr. Juan Martínez", "Matemáticas");

// Crear aulas
$aula1 = new Aula("A-101");
$aula2 = new Aula("A-102");

// Los estudiantes y profesores existen independientemente de las aulas
$aula1->asignarProfesor($profesor1);
$aula1->agregarEstudiante($estudiante1);
$aula1->agregarEstudiante($estudiante2);

// Un estudiante puede estar en múltiples aulas (compartido)
$aula2->asignarProfesor($profesor1); // Mismo profesor en ambas aulas
$aula2->agregarEstudiante($estudiante2); // Mismo estudiante en ambas aulas
$aula2->agregarEstudiante($estudiante3);

$aula1->mostrarInformacion();
$aula2->mostrarInformacion();

// Los objetos siguen existiendo aunque se destruya el aula
unset($aula1);
echo "El aula A-101 fue destruida, pero los estudiantes y profesor siguen
existiendo:\n";
echo "- {$estudiante1}\n";
echo "- {$estudiante2}\n";
echo "- {$profesor1}\n";
```

Resultado:

```
Profesor Dr. Juan Martínez asignado al aula A-101
Estudiante Ana García agregado al aula A-101
Estudiante Carlos López agregado al aula A-101
Profesor Dr. Juan Martínez asignado al aula A-102
Estudiante Carlos López agregado al aula A-102
Estudiante María Rodríguez agregado al aula A-102

=== Aula A-101 ===
Profesor: Prof. Dr. Juan Martínez (Matemáticas)
Estudiantes (2):
  - Ana García (ID: 101)
  - Carlos López (ID: 102)

=== Aula A-102 ===
Profesor: Prof. Dr. Juan Martínez (Matemáticas)
Estudiantes (2):
  - Carlos López (ID: 102)
  - María Rodríguez (ID: 103)
```

```
El aula A-101 fue destruida, pero los estudiantes y profesor siguen existiendo:
- Ana García (ID: 101)
- Carlos López (ID: 102)
- Prof. Dr. Juan Martínez (Matemáticas)
```

Composición: "Parte-de" con Dependencia

La composición representa una relación más fuerte donde los objetos componentes no pueden existir sin el objeto contenedor. Es una relación "parte-de" donde la vida de los componentes está ligada a la del contenedor.

Características de la composición:

- Los objetos componentes no pueden existir independientemente del objeto contenedor
- Los componentes son creados y destruidos junto con el contenedor
- La relación es más fuerte y representa una dependencia existencial

```php
// Clases que forman parte integral de un automóvil
class Motor {
    private $cilindros;
    private $caballosFuerza;
    private $encendido = false;

    public function __construct($cilindros, $caballosFuerza) {
        $this->cilindros = $cilindros;
        $this->caballosFuerza = $caballosFuerza;
        echo "Motor de {$cilindros} cilindros y {$caballosFuerza} HP creado\n";
    }

    public function encender() {
        $this->encendido = true;
        echo "Motor encendido\n";
    }

    public function apagar() {
        $this->encendido = false;
        echo "Motor apagado\n";
    }

    public function getEspecificaciones() {
        return "{$this->cilindros} cilindros, {$this->caballosFuerza} HP";
    }

    public function estaEncendido() {
        return $this->encendido;
    }

    // Destructor para mostrar cuando se destruye
    public function __destruct() {
```

```
        echo "Motor destruido\n";
    }
}

class Transmision {
    private $tipo;
    private $marchas;
    private $marchaActual = 0;

    public function __construct($tipo, $marchas) {
        $this->tipo = $tipo;
        $this->marchas = $marchas;
        echo "Transmisión {$tipo} de {$marchas} marchas creada\n";
    }

    public function cambiarMarcha($marcha) {
        if ($marcha >= 0 && $marcha <= $this->marchas) {
            $this->marchaActual = $marcha;
            if ($marcha == 0) {
                echo "Transmisión en neutral\n";
            } else {
                echo "Transmisión en marcha {$marcha}\n";
            }
        }
    }

    public function getTipo() {
        return $this->tipo;
    }

    public function __destruct() {
        echo "Transmisión destruida\n";
    }
}

class SistemaElectrico {
    private $voltaje;
    private $bateria;

    public function __construct($voltaje = 12) {
        $this->voltaje = $voltaje;
        $this->bateria = 100; // Porcentaje de batería
        echo "Sistema eléctrico de {$voltaje}V creado\n";
    }

    public function encenderLuces() {
        if ($this->bateria > 10) {
            echo "Luces encendidas\n";
            $this->bateria -= 5;
        } else {
            echo "Batería insuficiente para encender luces\n";
        }
    }
```

```
    public function getNivelBateria() {
        return $this->bateria;
    }

    public function __destruct() {
        echo "Sistema eléctrico destruido\n";
    }
}

// Composición: El automóvil "contiene" motor, transmisión y sistema eléctrico
class AutomovilCompleto {
    private $marca;
    private $modelo;
    private $motor;      // Composición
    private $transmision; // Composición
    private $sistemaElectrico; // Composición

    public function __construct($marca, $modelo, $cilindrosMotor, $hpMotor,
$tipoTransmision, $marchas) {
        $this->marca = $marca;
        $this->modelo = $modelo;

        echo "Creando {$marca} {$modelo}...\n";

        // Los componentes se crean junto con el automóvil (composición)
        $this->motor = new Motor($cilindrosMotor, $hpMotor);
        $this->transmision = new Transmision($tipoTransmision, $marchas);
        $this->sistemaElectrico = new SistemaElectrico();

        echo "{$marca} {$modelo} creado exitosamente\n\n";
    }

    public function encender() {
        echo "Encendiendo {$this->marca} {$this->modelo}...\n";
        $this->motor->encender();
        $this->sistemaElectrico->encenderLuces();
        $this->transmision->cambiarMarcha(0);
        echo "Vehículo listo para conducir\n\n";
    }

    public function acelerar() {
        if ($this->motor->estaEncendido()) {
            $this->transmision->cambiarMarcha(1);
            echo "Acelerando...\n";
        } else {
            echo "No se puede acelerar, el motor está apagado\n";
        }
    }

    public function mostrarEspecificaciones() {
        echo "=== Especificaciones {$this->marca} {$this->modelo} ===\n";
        echo "Motor: " . $this->motor->getEspecificaciones() . "\n";
```

```
        echo "Transmisión: " . $this->transmision->getTipo() . "\n";
        echo "Batería: " . $this->sistemaElectrico->getNivelBateria() . "%\n";
        echo "\n";
    }

    // Cuando se destruye el automóvil, todos sus componentes también se
destruyen
    public function __destruct() {
        echo "Destruyendo {$this->marca} {$this->modelo}...\n";
        // PHP automáticamente destruye los objetos componentes
        // porque no hay referencias externas a ellos
    }
}

// Ejemplo de uso de composición
echo "=== Ejemplo de Composición ===\n";
$miAuto = new AutomovilCompleto("Toyota", "Camry", 4, 200, "Automática", 6);
$miAuto->mostrarEspecificaciones();
$miAuto->encender();
$miAuto->acelerar();

// Cuando el automóvil se destruye, todos sus componentes internos también se
destruyen
echo "Destruyendo el automóvil...\n";
unset($miAuto);
echo "El automóvil y todos sus componentes han sido destruidos\n";
```

Resultado:

```
=== Ejemplo de Composición ===
Creando Toyota Camry...
Motor de 4 cilindros y 200 HP creado
Transmisión Automática de 6 marchas creada
Sistema eléctrico de 12V creado
Toyota Camry creado exitosamente

=== Especificaciones Toyota Camry ===
Motor: 4 cilindros, 200 HP
Transmisión: Automática
Batería: 100%

Encendiendo Toyota Camry...
Motor encendido
Luces encendidas
Transmisión en neutral
Vehículo listo para conducir

Transmisión en marcha 1
Acelerando...
Destruyendo el automóvil...
Destruyendo Toyota Camry...
```

```
Motor destruido
Transmisión destruida
Sistema eléctrico destruido
El automóvil y todos sus componentes han sido destruidos
```

Ejemplo Mixto: Agregación y Composición Juntas

En la práctica, es común que una clase use tanto agregación como composición. Veamos un ejemplo de una empresa:

```php
class Empleado {
    private $nombre;
    private $id;
    private $puesto;

    public function __construct($nombre, $id, $puesto) {
        $this->nombre = $nombre;
        $this->id = $id;
        $this->puesto = $puesto;
    }

    public function getNombre() {
        return $this->nombre;
    }

    public function getId() {
        return $this->id;
    }

    public function getPuesto() {
        return $this->puesto;
    }

    public function trabajar() {
        echo "{$this->nombre} está trabajando como {$this->puesto}\n";
    }

    public function __toString() {
        return "{$this->nombre} ({$this->puesto})";
    }
}

// Clase que existe solo como parte de la empresa (composición)
class Departamento {
    private $nombre;
    private $presupuesto;
    private $empleados = [];

    public function __construct($nombre, $presupuesto) {
        $this->nombre = $nombre;
```

```
        $this->presupuesto = $presupuesto;
        echo "Departamento de {$nombre} creado con presupuesto de
${presupuesto}\n";
    }

    public function agregarEmpleado(Empleado $empleado) {
        $this->empleados[] = $empleado;
        echo "Empleado {$empleado->getNombre()} agregado al departamento de
{$this->nombre}\n";
    }

    public function getNombre() {
        return $this->nombre;
    }

    public function getEmpleados() {
        return $this->empleados;
    }

    public function mostrarInformacion() {
        echo "\nDepartamento: {$this->nombre}\n";
        echo "Presupuesto: ${$this->presupuesto}\n";
        echo "Empleados (" . count($this->empleados) . "):\n";
        foreach ($this->empleados as $empleado) {
            echo "  - {$empleado}\n";
        }
    }

    public function __destruct() {
        echo "Departamento de {$this->nombre} eliminado\n";
    }
}

class Empresa {
    private $nombre;
    private $direccion;
    private $empleados = [];     // Agregación: empleados pueden existir sin la
empresa
    private $departamentos = []; // Composición: departamentos son parte
integral de la empresa

    public function __construct($nombre, $direccion) {
        $this->nombre = $nombre;
        $this->direccion = $direccion;
        echo "Empresa {$nombre} creada\n";
    }

    // Agregar empleado existente (agregación)
    public function contratarEmpleado(Empleado $empleado) {
        $this->empleados[] = $empleado;
        echo "Empleado {$empleado->getNombre()} contratado por {$this-
>nombre}\n";
    }
```

```
    // Crear departamento (composición)
    public function crearDepartamento($nombreDpto, $presupuesto) {
        $departamento = new Departamento($nombreDpto, $presupuesto);
        $this->departamentos[] = $departamento;
        echo "Departamento de {$nombreDpto} creado en {$this->nombre}\n";
        return $departamento;
    }

    public function asignarEmpleadoADepartamento($idEmpleado,
$nombreDepartamento) {
        $empleado = $this->buscarEmpleadoPorId($idEmpleado);
        $departamento = $this-
>buscarDepartamentoPorNombre($nombreDepartamento);

        if ($empleado && $departamento) {
            $departamento->agregarEmpleado($empleado);
        }
    }

    private function buscarEmpleadoPorId($id) {
        foreach ($this->empleados as $empleado) {
            if ($empleado->getId() == $id) {
                return $empleado;
            }
        }
        return null;
    }

    private function buscarDepartamentoPorNombre($nombre) {
        foreach ($this->departamentos as $departamento) {
            if ($departamento->getNombre() == $nombre) {
                return $departamento;
            }
        }
        return null;
    }

    public function mostrarEstructuraOrganizacional() {
        echo "\n=== ESTRUCTURA ORGANIZACIONAL DE {$this->nombre} ===\n";
        echo "Dirección: {$this->direccion}\n";
        echo "Total de empleados: " . count($this->empleados) . "\n";
        echo "Total de departamentos: " . count($this->departamentos) . "\n\n";

        foreach ($this->departamentos as $departamento) {
            $departamento->mostrarInformacion();
            echo "\n";
        }
    }

    public function __destruct() {
        echo "Empresa {$this->nombre} cerrada\n";
        // Los departamentos se destruyen automáticamente (composición)
```

```
        // Los empleados pueden seguir existiendo (agregación)
    }
}

// Ejemplo de uso mixto
echo "=== Ejemplo Mixto: Agregación y Composición ===\n";

// Crear empleados independientes (existirán sin la empresa)
$empleado1 = new Empleado("Ana Martínez", 101, "Desarrolladora");
$empleado2 = new Empleado("Carlos Ruiz", 102, "Diseñador");
$empleado3 = new Empleado("Luis García", 103, "Contador");
$empleado4 = new Empleado("María López", 104, "Gerente");

// Crear empresa
$empresa = new Empresa("TechCorp", "Av. Principal 123");

// Contratar empleados (agregación)
$empresa->contratarEmpleado($empleado1);
$empresa->contratarEmpleado($empleado2);
$empresa->contratarEmpleado($empleado3);
$empresa->contratarEmpleado($empleado4);

// Crear departamentos (composición)
$empresa->crearDepartamento("Desarrollo", 50000);
$empresa->crearDepartamento("Diseño", 30000);
$empresa->crearDepartamento("Contabilidad", 25000);

// Asignar empleados a departamentos
$empresa->asignarEmpleadoADepartamento(101, "Desarrollo");
$empresa->asignarEmpleadoADepartamento(102, "Diseño");
$empresa->asignarEmpleadoADepartamento(103, "Contabilidad");
$empresa->asignarEmpleadoADepartamento(104, "Desarrollo");

$empresa->mostrarEstructuraOrganizacional();

echo "Cerrando la empresa...\n";
unset($empresa);

echo "\nLos empleados siguen existiendo después del cierre de la empresa:\n";
foreach ([$empleado1, $empleado2, $empleado3, $empleado4] as $empleado) {
    echo "- {$empleado} sigue disponible para trabajar\n";
}
```

Diferencias Clave entre Agregación y Composición

Aspecto	Agregación	Composición
Relación	"Tiene-un" (más débil)	"Parte-de" (más fuerte)
Independencia	Los componentes pueden existir sin el contenedor	Los componentes no pueden existir sin el contenedor

Aspecto	Agregación	Composición
Ciclo de vida	Los componentes tienen ciclo de vida independiente	Los componentes se crean y destruyen con el contenedor
Compartición	Los componentes pueden ser compartidos	Los componentes son exclusivos del contenedor
Ejemplo	Aula con estudiantes	Automóvil con motor
Implementación	Se pasan objetos existentes como parámetros	Se crean objetos internamente en el constructor

Cuándo Usar Cada Una

Usar Agregación cuando:

- Los objetos pueden existir independientemente
- Los objetos pueden ser compartidos entre múltiples contenedores
- La relación es temporal o puede cambiar
- Los objetos tienen responsabilidades y ciclos de vida independientes

Usar Composición cuando:

- Los objetos componentes no tienen sentido sin el contenedor
- Los componentes son parte integral e inseparable del todo
- Los componentes se crean específicamente para el contenedor
- La destrucción del contenedor debe implicar la destrucción de los componentes

Ventajas de las Relaciones Objeto

1. **Modelado realista**: Permiten representar relaciones del mundo real de manera natural
2. **Reutilización**: Especialmente en agregación, los objetos pueden ser reutilizados
3. **Mantenibilidad**: Separación clara de responsabilidades
4. **Flexibilidad**: Diferentes tipos de relaciones para diferentes necesidades
5. **Encapsulamiento**: Cada objeto mantiene su propia responsabilidad

Resumen de Conceptos

En esta guía hemos cubierto los conceptos fundamentales de la POO en PHP:

1. **Clases y Objetos**: Plantillas y sus instancias
2. **Encapsulamiento**: Protección y control de acceso a datos
3. **Herencia**: Extensión de funcionalidad entre clases
4. **Polimorfismo**: Diferentes implementaciones para una misma interfaz
5. **Abstracción**: Definición de contratos y comportamientos esenciales
6. **Agregación**: Relaciones "tiene-un" con independencia
7. **Composición**: Relaciones "parte-de" con dependencia

Tabla Comparativa de Conceptos

Concepto	Propósito	Implementación	Beneficio Principal
Encapsulamiento	Ocultar implementación	`private`, `protected`	Seguridad y control
Herencia	Reutilizar código	`extends`	Extensibilidad
Polimorfismo	Misma interfaz, múltiples formas	Interfaces, herencia	Flexibilidad
Abstracción	Definir contratos	`abstract`, `interface`	Consistencia
Agregación	Relación débil "tiene-un"	Pasar objetos existentes	Reutilización
Composición	Relación fuerte "parte-de"	Crear objetos internos	Integridad

La comprensión y aplicación correcta de estos conceptos es fundamental para desarrollar aplicaciones robustas, mantenibles y escalables usando PHP orientado a objetos.