PHP & Laravel Backend Engineering

Week 5 – Object-Oriented Programming (OOP) in PHP

Program: PHP & Laravel Backend Engineering
Week: 5
Level: Intermediate
Duration: 3 Days

📋 Learning Objectives

By the end of this week, you will be able to:

- 🎯 Understand what OOP is and why we use it
- 🏗️ Create classes and objects
- 🔒 Use properties and methods
- 🎨 Implement encapsulation with access modifiers
- 🧬 Apply inheritance to reuse code
- 🔄 Understand polymorphism
- 🛠️ Work with constructors and destructors
- 📦 Use static methods and properties

Day 1 – Classes, Objects & Properties

🎯 What is Object-Oriented Programming?

OOP is a programming style that uses **objects** to organize code.

Real-world analogy:
Think of a **Car**:

- **Properties** (characteristics): color, brand, model, speed
- **Methods** (actions): start(), stop(), accelerate(), brake()

In OOP, we create a **blueprint** (class) and then make **actual cars** (objects) from it.

```
Class (Blueprint) → Object (Actual Thing)
Car Class → My Red Toyota, Your Blue Honda
```

📝 Why Use OOP?

Benefit	Explanation	Example
Reusability	Write once, use many times	Create one User class, make many users
Organization	Code is well-structured	All user-related code in one place
Maintainability	Easy to update and fix	Change User class, all users updated
Real-world modeling	Code matches real life	Bank Account class for banking app

🏗️ Creating Your First Class

Step 1: Define a Class

A class is like a blueprint or template.

```php
<?php
// Define a class
class Car {
    // Properties (variables inside a class)
    public $brand;
    public $color;
    public $speed;

    // Methods (functions inside a class)
    public function start() {
        return "The car is starting...";
    }

    public function accelerate() {
        $this->speed += 10;
        return "Speed increased to {$this->speed} km/h";
    }

    public function brake() {
        $this->speed -= 10;
        if ($this->speed < 0) {
            $this->speed = 0;
        }
        return "Speed decreased to {$this->speed} km/h";
```

```
    }
}
?>
```

Explanation:

- `class Car` - Creates a new class named Car
- `public` - Makes property/method accessible from outside
- `$this->` - Refers to the current object
- Properties store data, methods perform actions

🎨 Creating Objects from Classes

Step 2: Create Objects (Instances)

Once you have a class, you can create multiple objects from it.

```
<?php
// Create objects from the Car class
$car1 = new Car();
$car1->brand = "Toyota";
$car1->color = "Red";
$car1->speed = 0;

$car2 = new Car();
$car2->brand = "Honda";
$car2->color = "Blue";
$car2->speed = 0;

// Use the methods
echo $car1->start();  // Output: The car is starting...
echo $car1->accelerate();  // Output: Speed increased to 10 km/h
echo $car1->accelerate();  // Output: Speed increased to 20 km/h

echo $car2->start();  // Output: The car is starting...
echo $car2->accelerate();  // Output: Speed increased to 10 km/h
?>
```

Explanation:

- `new Car()` - Creates a new object from the Car class
- `$car1->brand` - Access object property

- `$car1->start()` - Call object method
- Each object has its own properties (car1 and car2 are independent)

🔧 Constructor Method

A **constructor** is a special method that runs automatically when you create an object.

```
<?php
class Car {
    public $brand;
    public $color;
    public $speed;

    // Constructor - runs when object is created
    public function __construct($brand, $color) {
        $this->brand = $brand;
        $this->color = $color;
        $this->speed = 0;
        echo "A new {$this->color} {$this->brand} has been created!<br>";
    }

    public function getInfo() {
        return "{$this->color} {$this->brand} - Speed: {$this->speed} km/
    }
}

// Create objects using constructor
$car1 = new Car("Toyota", "Red");
// Output: A new Red Toyota has been created!

$car2 = new Car("Honda", "Blue");
// Output: A new Blue Honda has been created!

echo $car1->getInfo();  // Output: Red Toyota - Speed: 0 km/h
?>
```

Explanation:

- `__construct()` - Special method name for constructor
- Runs automatically when you use `new Car()`
- Helps initialize object properties
- Makes object creation easier

🎯 Practice Exercises

1. **Create a Student Class**

```
class Student {
    public $name;
    public $age;
    public $grade;

    public function __construct($name, $age, $grade) {
        $this->name = $name;
        $this->age = $age;
        $this->grade = $grade;
    }

    public function introduce() {
        return "Hi, I'm {$this->name}, {$this->age} years old, grade
    }
}
```

2. **Create a BankAccount Class**

```
class BankAccount {
    public $accountNumber;
    public $balance;

    public function __construct($accountNumber, $initialBalance) {
        $this->accountNumber = $accountNumber;
        $this->balance = $initialBalance;
    }

    public function deposit($amount) {
        $this->balance += $amount;
        return "Deposited: $amount. New balance: {$this->balance}";
    }

    public function withdraw($amount) {
        if ($amount > $this->balance) {
            return "Insufficient funds!";
        }
        $this->balance -= $amount;
        return "Withdrawn: $amount. New balance: {$this->balance}";
```

```
        }
    }
```

3. **Create a Book Class**
4. **Create a Product Class with price and discount methods**

Day 2 – Encapsulation & Access Modifiers

🔒 What is Encapsulation?

Encapsulation means hiding data and only allowing access through methods.

Real-world analogy:
Think of an ATM machine:

- You can't directly access the money inside (private)
- You use buttons and screen to interact (public methods)
- The internal mechanism is hidden (protected)

🎨 Access Modifiers

PHP has three access modifiers:

Modifier	Access Level	Example Use
`public`	Accessible everywhere	Methods users should use
`private`	Only inside the class	Sensitive data, internal methods
`protected`	Class and child classes	Shared with inheritance

📝 Using Private Properties

```
<?php
class BankAccount {
    // Private properties - can't access directly
    private $accountNumber;
    private $balance;
    private $pin;

    public function __construct($accountNumber, $initialBalance, $pin) {
```

```
        $this->accountNumber = $accountNumber;
        $this->balance = $initialBalance;
        $this->pin = $pin;
    }

    // Public method to deposit money
    public function deposit($amount) {
        if ($amount > 0) {
            $this->balance += $amount;
            return "Deposited: $$amount";
        }
        return "Invalid amount";
    }

    // Public method to withdraw money
    public function withdraw($amount, $enteredPin) {
        // Check PIN first
        if (!$this->verifyPin($enteredPin)) {
            return "Invalid PIN!";
        }

        if ($amount > $this->balance) {
            return "Insufficient funds!";
        }

        $this->balance -= $amount;
        return "Withdrawn: $$amount";
    }

    // Private method - only used inside class
    private function verifyPin($enteredPin) {
        return $this->pin === $enteredPin;
    }

    // Public method to check balance
    public function getBalance($enteredPin) {
        if ($this->verifyPin($enteredPin)) {
            return "Balance: $" . $this->balance;
        }
        return "Invalid PIN!";
    }
}

// Usage
```

```php
$account = new BankAccount("123456", 1000, "1234");

echo $account->deposit(500);  // Output: Deposited: $500
echo $account->getBalance("1234");  // Output: Balance: $1500
echo $account->withdraw(200, "1234");  // Output: Withdrawn: $200
echo $account->withdraw(200, "9999");  // Output: Invalid PIN!

// This will cause an error:
// echo $account->balance;  // ERROR! balance is private
?>
```

Explanation:

- `private $balance` - Can't access from outside
- Public methods provide controlled access
- `verifyPin()` is private - only used internally
- Protects sensitive data from direct manipulation

🎯 Getters and Setters

Getters and **Setters** are methods to access and modify private properties.

```php
<?php
class User {
    private $name;
    private $email;
    private $age;

    public function __construct($name, $email, $age) {
        $this->name = $name;
        $this->setEmail($email);  // Use setter for validation
        $this->setAge($age);
    }

    // Getter for name
    public function getName() {
        return $this->name;
    }

    // Setter for name
    public function setName($name) {
        if (strlen($name) >= 3) {
            $this->name = $name;
```

```
        } else {
            echo "Name must be at least 3 characters!";
        }
    }

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

    // Setter for email with validation
    public function setEmail($email) {
        if (filter_var($email, FILTER_VALIDATE_EMAIL)) {
            $this->email = $email;
        } else {
            echo "Invalid email format!";
        }
    }

    // Getter for age
    public function getAge() {
        return $this->age;
    }

    // Setter for age with validation
    public function setAge($age) {
        if ($age >= 18 && $age <= 100) {
            $this->age = $age;
        } else {
            echo "Age must be between 18 and 100!";
        }
    }
}

// Usage
$user = new User("John Doe", "john@example.com", 25);

echo $user->getName();  // Output: John Doe
echo $user->getEmail();  // Output: john@example.com

$user->setAge(30);  // Valid
$user->setAge(150);  // Output: Age must be between 18 and 100!
?>
```

Why use getters and setters?

- **Validation** - Check data before setting
- **Control** - Decide what can be changed
- **Flexibility** - Can change internal implementation later

🎯 Practice Exercises

1. **Create a Product Class with private properties**
    - Private: name, price, stock
    - Public: getters and setters with validation
    - Method: calculateDiscount()
2. **Create a Password Class**
    - Private: password (hashed)
    - Public: setPassword() - hashes before storing
    - Public: verify() - checks if password matches
3. **Create an Email Class with validation**

Day 3 – Inheritance & Polymorphism

🧬 What is Inheritance?

Inheritance allows a class to inherit properties and methods from another class.

Real-world analogy:
Think of animals:

- **Parent Class**: Animal (has eat(), sleep())
- **Child Classes**: Dog (has bark()), Cat (has meow())
- Dogs and cats inherit eat() and sleep() from Animal

```
Animal (Parent)
  ├── Dog (Child) - inherits from Animal
  ├── Cat (Child) - inherits from Animal
  └── Bird (Child) - inherits from Animal
```

📝 Creating Parent and Child Classes

```php
<?php
// Parent class (Base class)
class Animal {
    protected $name;
    protected $age;

    public function __construct($name, $age) {
        $this->name = $name;
        $this->age = $age;
    }

    public function eat() {
        return "{$this->name} is eating...";
    }

    public function sleep() {
        return "{$this->name} is sleeping...";
    }

    public function getInfo() {
        return "{$this->name} is {$this->age} years old";
    }
}

// Child class - inherits from Animal
class Dog extends Animal {
    private $breed;

    public function __construct($name, $age, $breed) {
        parent::__construct($name, $age);  // Call parent constructor
        $this->breed = $breed;
    }

    // Dog-specific method
    public function bark() {
        return "{$this->name} says: Woof! Woof!";
    }

    public function fetch() {
        return "{$this->name} is fetching the ball!";
    }
}

// Another child class
```

```php
class Cat extends Animal {
    private $color;

    public function __construct($name, $age, $color) {
        parent::__construct($name, $age);
        $this->color = $color;
    }

    // Cat-specific method
    public function meow() {
        return "{$this->name} says: Meow!";
    }

    public function scratch() {
        return "{$this->name} is scratching the furniture!";
    }
}

// Usage
$dog = new Dog("Buddy", 3, "Golden Retriever");
echo $dog->eat();     // Inherited from Animal
echo $dog->sleep();   // Inherited from Animal
echo $dog->bark();    // Dog-specific
echo $dog->fetch();   // Dog-specific

$cat = new Cat("Whiskers", 2, "Orange");
echo $cat->eat();     // Inherited from Animal
echo $cat->meow();    // Cat-specific
echo $cat->scratch(); // Cat-specific
?>
```

Explanation:

- `extends` - Creates inheritance relationship
- `parent::__construct()` - Calls parent class constructor
- `protected` - Accessible in parent and child classes
- Child classes get all parent methods automatically

🔄 Method Overriding

Child classes can **override** (replace) parent methods.

```php
<?php
class Animal {
    protected $name;

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

    public function makeSound() {
        return "{$this->name} makes a sound";
    }
}

class Dog extends Animal {
    // Override parent method
    public function makeSound() {
        return "{$this->name} says: Woof! Woof!";
    }
}

class Cat extends Animal {
    // Override parent method
    public function makeSound() {
        return "{$this->name} says: Meow!";
    }
}

class Cow extends Animal {
    // Override parent method
    public function makeSound() {
        return "{$this->name} says: Moo!";
    }
}

// Usage
$dog = new Dog("Buddy");
$cat = new Cat("Whiskers");
$cow = new Cow("Bessie");

echo $dog->makeSound();  // Output: Buddy says: Woof! Woof!
echo $cat->makeSound();  // Output: Whiskers says: Meow!
echo $cow->makeSound();  // Output: Bessie says: Moo!
?>
```

Explanation:

- Each child class has its own version of `makeSound()`
- Same method name, different behavior
- This is called **Polymorphism**

🎨 Polymorphism in Action

Polymorphism means "many forms" - same interface, different implementations.

```
<?php
// Parent class
class Shape {
    protected $name;

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

    public function calculateArea() {
        return "Area calculation not implemented";
    }
}

// Child classes with different area calculations
class Rectangle extends Shape {
    private $width;
    private $height;

    public function __construct($width, $height) {
        parent::__construct("Rectangle");
        $this->width = $width;
        $this->height = $height;
    }

    public function calculateArea() {
        return $this->width * $this->height;
    }
}

class Circle extends Shape {
    private $radius;
```

```
    public function __construct($radius) {
        parent::__construct("Circle");
        $this->radius = $radius;
    }

    public function calculateArea() {
        return 3.14159 * $this->radius * $this->radius;
    }
}

class Triangle extends Shape {
    private $base;
    private $height;

    public function __construct($base, $height) {
        parent::__construct("Triangle");
        $this->base = $base;
        $this->height = $height;
    }

    public function calculateArea() {
        return 0.5 * $this->base * $this->height;
    }
}

// Polymorphism in action
$shapes = [
    new Rectangle(5, 10),
    new Circle(7),
    new Triangle(6, 8)
];

foreach ($shapes as $shape) {
    echo "{$shape->name} area: " . $shape->calculateArea() . "<br>";
}
// Output:
// Rectangle area: 50
// Circle area: 153.93804
// Triangle area: 24
?>
```

Explanation:

- All shapes have `calculateArea()` method

- Each shape calculates area differently
- We can treat all shapes the same way (polymorphism)

📦 Static Methods and Properties

Static members belong to the class itself, not to objects.

```
<?php
class MathHelper {
    // Static property
    public static $pi = 3.14159;

    // Static method
    public static function add($a, $b) {
        return $a + $b;
    }

    public static function multiply($a, $b) {
        return $a * $b;
    }

    public static function circleArea($radius) {
        return self::$pi * $radius * $radius;
    }
}

// Use static methods without creating object
echo MathHelper::add(5, 3);  // Output: 8
echo MathHelper::multiply(4, 7);  // Output: 28
echo MathHelper::circleArea(5);  // Output: 78.53975

// Access static property
echo MathHelper::$pi;  // Output: 3.14159
?>
```

Explanation:

- `static` - Belongs to class, not object
- `::` - Used to access static members
- `self::` - Refers to current class
- No need to create object

🎯 Practical Example: User Management System

```php
<?php
// Parent class
class Person {
    protected $name;
    protected $email;
    protected $age;

    public function __construct($name, $email, $age) {
        $this->name = $name;
        $this->email = $email;
        $this->age = $age;
    }

    public function getInfo() {
        return "{$this->name} ({$this->email})";
    }
}

// Child class - Student
class Student extends Person {
    private $studentId;
    private $grade;
    private $courses = [];

    public function __construct($name, $email, $age, $studentId, $grade)
        parent::__construct($name, $email, $age);
        $this->studentId = $studentId;
        $this->grade = $grade;
    }

    public function enrollCourse($course) {
        $this->courses[] = $course;
        return "{$this->name} enrolled in {$course}";
    }

    public function getCourses() {
        return implode(", ", $this->courses);
    }

    public function getInfo() {
        return "Student: {$this->name} (ID: {$this->studentId}, Grade: {$
```

```
    }
}

// Child class - Teacher
class Teacher extends Person {
    private $employeeId;
    private $subject;
    private $salary;

    public function __construct($name, $email, $age, $employeeId, $subjec
        parent::__construct($name, $email, $age);
        $this->employeeId = $employeeId;
        $this->subject = $subject;
        $this->salary = $salary;
    }

    public function teach($course) {
        return "{$this->name} is teaching {$course}";
    }

    public function getInfo() {
        return "Teacher: {$this->name} (Subject: {$this->subject})";
    }
}

// Usage
$student = new Student("Ali Ahmed", "ali@example.com", 20, "S12345", "A")
echo $student->enrollCourse("PHP Programming");
echo $student->enrollCourse("Database Design");
echo $student->getInfo();
echo "Courses: " . $student->getCourses();

$teacher = new Teacher("Sara Khan", "sara@example.com", 35, "T001", "Comp
echo $teacher->teach("PHP Programming");
echo $teacher->getInfo();
?>
```

🎯 Practice Exercises

1. **Create a Vehicle Hierarchy**
    - Parent: Vehicle (brand, model, year)
    - Children: Car, Motorcycle, Truck

- Each with specific properties and methods

2. **Create an Employee System**
   - Parent: Employee (name, salary)
   - Children: Manager, Developer, Designer
   - Each with different bonus calculations

3. **Create a Payment System**
   - Parent: Payment
   - Children: CreditCard, PayPal, BankTransfer
   - Each with different processing methods

📚 Resources

- 📖 PHP OOP Documentation
- 🎓 PHP OOP Tutorial
- 💡 OOP Principles
- 🔍 PHP Design Patterns

💡 Tips for Success

- 🔄 **Practice daily** - Create different classes every day
- 🎨 **Think in objects** - Model real-world things as classes
- 📝 **Use meaningful names** - Class names should be nouns (User, Product)
- 🔒 **Encapsulate properly** - Make properties private, use getters/setters
- 🧬 **Don't over-inherit** - Only use inheritance when it makes sense
- 📖 **Read code** - Study how frameworks use OOP
- 🐞 **Debug with var_dump()** - Inspect objects to understand them

✅ Week 5 Summary

What you learned:

- 🎯 OOP concepts and benefits
- 🏗️ Creating classes and objects
- 🔧 Using constructors
- 🔒 Encapsulation with access modifiers
- 📝 Getters and setters
- 🧬 Inheritance and code reuse
- 🔄 Method overriding and polymorphism

- 📦 Static methods and properties

Next week: Building a complete PHP project using OOP principles!