Unit 7: Advanced PHP Concepts

Presented By:
Bipin Maharjan

Learning Objectives

- Understand the importance and benefits of using object-oriented programming (OOP) in PHP.
- Creating and using classes and objects to organize and encapsulate code.
- Understand the purpose and usage of static properties and methods, including declaring and accessing static members without the need for class instantiation.
- Learn about properties and methods in PHP classes and how they contribute to code organization and reusability.
- Explore advanced concepts such as inheritance, encapsulation, and polymorphism.
- Learn about PHP magic methods and their role in implementing specific functionality, enhancing code flexibility and extensibility.

Table of Contents

Object-oriented programming (OOP) in PHP

What is OOP?

- Object-Oriented Programming (OOP) is a programming paradigm that organizes code into objects, which are instances of classes.
- Each object combines data (properties) and behavior (methods) into a single logical unit.
- A relationship between one object and other object is created by the programmer.
- PHP supports OOP since version 5, allowing developers to write modular, maintainable, and reusable code.
- In PHP, OOP makes code more modular, reusable, and maintainable, especially for large projects such as e-commerce systems, CMS (like WordPress), or web apps built using frameworks (like Laravel or CodeIgniter).

Evolution of PHP Toward PHP

- **PHP 4:** Introduced basic class support (but limited and less strict).
- **PHP 5:** Added full OOP support with access modifiers, constructors, destructors, inheritance, interfaces, and abstract classes.
- **PHP 7+:** Improved OOP performance, added type hints, and introduced features like traits.
- **PHP 8+:** Enhanced OOP with features like union types, constructor property promotion, and named arguments.

OOP in PHP is now a core part of modern development — nearly all professional frameworks and libraries use it.

Key Concepts of OOP

OOP in PHP revolves around four main pillars:

Concept	Description	Example
Class	A blueprint that defines properties (variables) and methods (functions).	`class Car {}`
Object	An instance of a class, representing a real-world entity.	`$car1 = new Car();`
Encapsulation	Bundling data and methods together while restricting direct access.	Use of private/protected properties
Inheritance	A class can inherit methods and properties from another class.	`class Dog extends Animal {}`
Polymorphism	Different classes can define methods with the same name but different behaviors.	`draw()` method for Circle and Rectangle

Benefits of OOP

- **Reusability:** Classes can be reused across multiple projects.
- **Maintainability:** Easier to update and maintain large codebases.
- **Encapsulation:** Keeps related data and functions together.
- **Abstraction:** Hides complex details and exposes simple interfaces.
- **Inheritance and Polymorphism:** Enable flexibility and code extension.
- **Scalability:** Easier to extend functionality by adding new classes or inheriting existing ones.
- **Real-world Modeling:** OOP allows programmers to represent real entities (e.g., Users, Products, Orders) as code objects.
- **Framework Compatibility:** PHP frameworks (Laravel, Symfony, CodeIgniter) are built using OOP principles.

Example of OOP: (Procedural vs OOP)

Procedural Approach

```
$name = "John";

function greet($name) {

    return "Hello, $name!";

}

echo greet($name);
```

OOP Approach

```
class Greeter {
    public $name;

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

    public function greet() {
        return "Hello, {$this->name}!";
    }
}
$person = new Greeter("John");
echo $person->greet();
```

OOP Terminology

Term	Description	Example
Class	Blueprint for creating objects.	`class Car {}`
Object	Instance of a class.	`$car = new Car();`
Property	Variable inside a class.	`$this->color`
Method	Function inside a class.	`public function start()`
Constructor	Special method to initialize an object.	`__construct()`
$this	Refers to the current object.	`$this->brand`

Advantages and Disadvantages of OOP

Advantages	**Disadvantages**
Code reusability	Slightly more complex to learn
Easier maintenance	Requires careful class design
Scalable and modular	Overhead for small scripts
Better security via encapsulation	Debugging can be tricky in large hierarchies

Creating and Using Classes and Objects

What is a Class?

- A class in PHP is a blueprint or template for creating objects.
- It defines:
  - **Properties (variables)** that describe the object’s data.
  - **Methods (functions)** that define the object’s behavior.

Think of a class as a blueprint of a mobile phone.

The design, size, and features are defined (class definition).

When you create an object, you are manufacturing an actual phone using that blueprint.

Each phone (object) can have different data (e.g., color, model name), but all share the same structure.

Class Example

```
class Car {
    // Properties
    public $brand;
    public $color;

    // Methods
    public function startEngine() {
        echo "Engine started!";
    }
}
```

What is an Object?

- An object is an instance of a class
- It represents a real-world entity created from the class blueprint.
- You can create multiple objects from the same class, and each can have different data.

Creating an Object

Use the ***new*** keyword:

```
$car1 = new Car();  // Creates an object of class Car
```

Now $car1 is an object with access to all the properties and methods defined in Car.

Accessing Properties and Methods

Use the object operator (->) to access or modify class members.

```
$car1->brand = "Toyota";
$car1->color = "Red";

echo $car1->brand;           // Output: Toyota
$car1->startEngine();        // Output: Engine started!
```

Class and Object Example

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

    public function startEngine() {
        echo "The {$this->color} {$this->brand}'s engine is
starting...<br>";
    }

    public function stopEngine() {
        echo "The {$this->brand} engine has stopped.";
    }
}

// Create an object (instance) of class Car
$car1 = new Car();

// Set property values
$car1->brand = "Toyota";
$car1->color = "Blue";

// Access properties and methods
$car1->startEngine();
$car1->stopEngine();
?>
```

Output:

```
The Blue Toyota's engine is starting...

The Toyota engine has stopped.
```

Using the $this keyword

- $this refers to the current object inside a class.
- It is used to access the object’s own properties or methods.

```
class Person {
    public $name;
    public function introduce() {
        echo "Hello, my name is {$this->name}.";
    }
}
$p1 = new Person();
$p1->name = "Alice";
$p1->introduce();   // Output: Hello, my name is Alice.
```

Using Constructor (__construct)

- A constructor is a special method automatically called when an object is created.
- It is used to initialize property values.

Using Constructor Example

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

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

    public function display() {
        echo "Student Name: {$this->name}, Age:
{$this->age}";
    }
}

$student1 = new Student("John", 21);
$student1->display();
```

Output:

```
Student Name: John, Age: 21
```

Using Destructor (__destruct)

- A destructor (__destruct()) is automatically called when an object is destroyed, usually at the end of the script.
- It’s useful for closing database connections or cleaning up resources.

Using Destructor Example

```
class FileHandler {
    public function __construct() {
        echo "File opened.<br>";
    }

    public function __destruct() {
        echo "File closed.";
    }
}

$file = new FileHandler();
```

Output:

```
File opened.

File closed.
```

Properties and Methods

What is Properties and Methods?

- Properties are variables defined inside a class (they hold state/data of an object).
- Methods are functions defined inside a class (they define behavior/operations on that data).
- Inside a class you use $this->property to access instance properties and methods.

Syntax

```
class Car {
    public $brand;        // property
    protected $color;
    private $mileage;

    public function start() {          // method
        echo "Starting {$this->brand}";
    }

    private function updateMileage($km) {
        $this->mileage += $km;
    }
}
```

Properties and Methods Example

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

    public function __construct($owner, $balance = 0) {
        $this->owner = $owner;
        $this->balance = $balance;
    }

    public function deposit($amount) {
        $this->balance += $amount;
        echo "{$this->owner} deposited $amount. New balance:
{$this->balance}<br>";
    }

    public function withdraw($amount) {
        if ($amount <= $this->balance) {
            $this->balance -= $amount;
            echo "{$this->owner} withdrew $amount. Remaining balance:
{$this->balance}<br>";
        } else {
            echo "Insufficient balance!<br>";
        }
    }
}
```

```
$account1 = new BankAccount("Alex", 1000);
$account1->deposit(500);
$account1->withdraw(200);
$account1->withdraw(2000);
```

Output:

```
Alex deposited 500. New balance: 1500

Alex withdrew 200. Remaining balance: 1300

Insufficient balance!
```

Access Modifiers

When declaring properties or methods, you can define who can access them:

Modifier	Description	Accessible From
`public`	Accessible everywhere	Class, object, child classes
`protected`	Accessible within class and its subclasses	Class + inherited classes
`private`	Accessible only inside the same class	Only within class

Access Modifiers Example

```
class Demo {
    public $publicVar = "Public";
    protected $protectedVar = "Protected";
    private $privateVar = "Private";

    public function showAll() {
        echo $this->publicVar."<br>";
        echo $this->protectedVar."<br>";
        echo $this->privateVar."<br>";
    }
}

$obj = new Demo();
echo $obj->publicVar."<br>"; // Works
// echo $obj->protectedVar;  // Error
// echo $obj->privateVar;    // Error
$obj->showAll(); // Works fine
```

Access Modifier Example (Inheritance Access)

```
class ParentClass {
    private $a = 1;
    protected $b = 2;
    public $c = 3;

    public function show() {
        echo $this->a, $this->b, $this->c;
    }
}

class ChildClass extends ParentClass {
    public function demo() {
        // $this->a // ERROR (private)
        echo $this->b; // OK (protected)
    }
}
```

Typed Properties (PHP 7.4+)

You can declare property types to catch bugs early.

```
class User {
    public string $name;
    public ?int $age = null;          // nullable int (int or null)
    public array $roles = [];
}
```

nullable / union types (PHP 8.0+)

```
class Price {
    public int|float $amount;        // accepts int OR float
}
```

Default Values and initializers

- Properties may have default values.
- Default values must be constant expressions (no runtime expressions) prior to PHP 8.1 (some relaxations exist later).

```
public int $count = 0;

public array $items = ['a', 'b'];
```

Constructor - Including Access Modifier

```
class Student {
    private string $name;
    private int $age;

    public function __construct(string $name, int $age = 0) {
        $this->name = $name;
        $this->age = $age;
    }

    public function getName(): string {
        return $this->name;
    }
}
```

Constructor property promotion (PHP 8.0+) [Optional]

Shorter syntax that declares and assigns from constructor parameters:

```
class Product {
    public function __construct(
        private string $name,
        private float $price = 0.0
    ) {}
}
```

Read-only properties (PHP 8.1+) [Optional]

- readonly properties can be written once (usually in the constructor) and then become immutable.
- Immutability is a great topic to tie to safer design and thread-safety ideas (where applicable).

```
class Point {
    public readonly float $x;
    public readonly float $y;

    public function __construct(float $x, float $y) {
        $this->x = $x;
        $this->y = $y;
    }
}
```

Methods — signatures, typing, and return types

- Methods can have parameter types and return types:

```
public function setPrice(float $p): void {
    $this->price = $p;
}


public function getPrice(): float {
    return $this->price;
}
```

- : void means no return.
- Use nullable return types : ?string where appropriate.
- Use union return types : int|float in PHP 8.0+ if needed.

By Reference Parameters

```
public function increaseQuantity(int &$q) {
    $q++;
}
```

- Prefer returning new values

Method modifiers

- **final** on a method prevents child classes from overriding it.
- **abstract** methods declare a signature without implementation (requires the class be abstract).
- **static** methods/properties belong to the class (not instance) — we'll cover static in full in the static subtopic, but mention:
  - Use **self::** or **static::** inside class definitions for static access.

Example of abstract + final

```
abstract class Shape {
    abstract public function area(): float;
}

final class Circle extends Shape {
    public function area(): float { /*...*/ }
}
```

Static Properties and Methods

What is Static Properties and Methods?

- Normally, to access a property or method in PHP, you must create an object of the class.
- However, sometimes we need certain shared data or behaviors that should belong to the class itself, not to any individual object.
- For such cases, PHP provides static properties and static methods.
- A static property is a variable shared among all instances of a class.
- A static method belongs to the class and can be called without creating an object.
- They are declared using the static keyword and accessed using the scope resolution operator (::).

Declaring Static Members

```
class Example {
    public static $count = 0;

    public static function sayHello() {
        echo "Hello from static method!";
    }
}

// Accessing static property
echo Example::$count;

// Calling static method
Example::sayHello();
```

Output:

```
0

Hello from static method!
```

Accessing Static Members

Context	Access Syntax	Example
Inside class	`self::$property` or `self::method()`	`self::$count++`
Outside class	`ClassName::$property` or `ClassName::method()`	`Student::getTotalStudents()`
In child class (with inheritance)	`parent::method()`	Used to access parent class static members

Static Method: Example

```
class MathHelper {
    public static function add($a, $b) {
        return $a + $b;
    }

    public static function square($n) {
        return $n * $n;
    }
}

echo "Sum: " . MathHelper::add(5, 10) . "<br>";
echo "Square: " . MathHelper::square(4);
```

Output:

```
Sum: 15

Square: 16
```

Using self::, parent::, and static::

Keyword	Description	Example
`self::`	Refers to the current class	`self::display()`
`parent::`	Refers to the parent class	`parent::display()`
`static::`	Late static binding (refers to the class that was actually called)	`static::display()`

Inheritance, Encapsulation, and Polymorphism

Inheritance

- Inheritance allows a class (called the child class or subclass) to inherit properties and methods from another class (called the parent class or base class).
- It helps you reuse code instead of rewriting it, and extend functionality easily.

Syntax:

```
class ParentClass {
   // properties and methods
}
class ChildClass extends ParentClass {
   // additional or overridden properties/methods
}
```

Inheritance Example

```
class Animal {
    public function eat() {
        echo "Eating food<br>";
    }
}

class Dog extends Animal {
    public function bark() {
        echo "Woof! Woof!<br>";
    }
}

$dog = new Dog();
$dog->eat();   // Inherited from Animal
$dog->bark();  // Defined in Dog
```

Output:

```
Eating food

Woof! Woof!
```

Note:

The Dog class inherits the eat() method from Animal, showing code reuse.

Method Overriding

A child class can override a parent class method by redefining it.

Example:

```
class Animal {
    public function sound() {
        echo "Animal makes a sound<br>";
    }
}

class Cat extends Animal {
    public function sound() {
        echo "Cat says Meow<br>";
    }
}
```

```
$cat = new Cat();
$cat->sound();
```

Output:

```
Cat says Meow
```

Note:

The Cat class overrides the parent sound() method.

Using parent:: keyword

You can still access the parent method inside the overridden one:

Example:

```
class Animal {
    public function sound() {
        echo "Animal makes a sound<br>";
    }
}

class Dog extends Animal {
    public function sound() {
        parent::sound();
        echo "Dog says Woof<br>";
    }
}
```

```
$dog = new Dog();

$dog->sound();
```

Output:

```
Animal makes a sound

Dog says Woof
```

Encapsulation

Encapsulation means bundling data (properties) and operations on that data (methods) inside one class, while restricting direct access to some class components.

It’s implemented using access modifiers in PHP:

- public – accessible from anywhere
- protected – accessible within the class and subclasses
- private – accessible only within the same class

Encapsulation Example

```
class BankAccount {
    private $balance = 0;

    public function deposit($amount) {
        $this->balance += $amount;
    }

    public function getBalance() {
        return $this->balance;
    }
}

$account = new BankAccount();
$account->deposit(500);
// $account->balance = 10000; ❌ Not allowed
— private property
echo "Current balance: " .
$account->getBalance();
```

Output:

```
Current balance: 500
```

Note:

Encapsulation hides the sensitive property $balance from direct access and provides controlled access through methods.

Getter and Setter Methods

Encapsulation often uses getters and setters for safe data access:

Example:

```
class Student {
    private $name;

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

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

$student = new Student();
$student->setName("John Doe");
echo $student->getName();
```

Output:

```
John Doe
```

Polymorphism

Polymorphism means “many forms.”

It allows different classes to use methods with the same name but possibly different implementations.

In PHP, polymorphism is achieved via:

- Method overriding (inheritance)
- Interfaces and abstract classes

Method Overriding: Example

```
class Shape {
    public function draw() {
        echo "Drawing a shape<br>";
    }
}

class Circle extends Shape {
    public function draw() {
        echo "Drawing a circle<br>";
    }
}

class Rectangle extends Shape {
    public function draw() {
        echo "Drawing a rectangle<br>";
    }
}
```

```
$shapes = [new Circle(), new Rectangle()];

foreach ($shapes as $shape) {
    $shape->draw();
}
```

Output:

```
Drawing a circle

Drawing a rectangle
```

Note:

Both classes share the same method name (draw()), but behave differently.

Interfaces for Polymorphism: Example

```
interface Payment {
    public function pay($amount);
}

class CreditCardPayment implements Payment
{
    public function pay($amount) {
        echo "Paid $amount using Credit
Card<br>";
    }
}

class PayPalPayment implements Payment {
    public function pay($amount) {
        echo "Paid $amount using PayPal<br>";
    }
}
```

```
$payments = [new CreditCardPayment(), new
PayPalPayment()];

foreach ($payments as $p) {
    $p->pay(100);
}
```

Output:

```
Paid 100 using Credit Card

Paid 100 using PayPal
```

Note:

Both classes implement the same interface and method name (pay()), but perform different actions.

Abstraction

- Abstraction means hiding complex implementation details and showing only essential features of an object.
- It allows developers to focus on what an object does rather than how it does it.
- In PHP, abstraction is implemented using abstract classes and interfaces.

Abstract Classes

- Cannot be instantiated directly (you can’t create an object from it).
- Can contain abstract methods (methods without implementation) and regular methods (with implementation).
- Must be inherited by another class that provides the actual method implementation.

Syntax:

```
abstract class ClassName {

    abstract protected function methodName();

}
```

Abstract Classes: Example

```
abstract class Shape {
    protected $color;

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

    abstract public function area(); // Abstract method

    public function describe() {
        echo "This is a {$this->color} shape.<br>";
    }
}

class Circle extends Shape {
    private $radius;

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

    public function area() {
        return 3.1416 * $this->radius * $this->radius;
    }
}

$circle = new Circle("red", 5);
$circle->describe();
echo "Area: " . $circle->area();
```

Output:

```
This is a red shape.

Area: 78.54
```

Note:

Shape is abstract — you can’t create a new Shape().

The method area() is abstract — every child class must define it.

The describe() method is concrete — it’s shared by all subclasses.

Interfaces (Pure Abstraction)

An interface is a contract that defines what methods a class must implement, but not how.

In contrast to abstract classes:

- Interfaces cannot contain any implemented methods.
- A class can implement multiple interfaces, unlike single inheritance for classes.

Interface: Example

```
interface Payment {
    public function pay($amount);
}

interface Refundable {
    public function refund($amount);
}

class CreditCardPayment implements Payment,
Refundable {
    public function pay($amount) {
        echo "Paid $amount using Credit Card<br>";
    }

    public function refund($amount) {
        echo "Refunded $amount to Credit Card<br>";
    }
}

$payment = new CreditCardPayment();
$payment->pay(100);
$payment->refund(50);
```

Output:

```
Paid 100 using Credit Card

Refunded 50 to Credit Card
```

Note:

The interfaces Payment and Refundable define what should happen (the method names),

while the class CreditCardPayment defines how it happens (the implementation).

Magic Methods

What is Magic Methods?

Magic methods in PHP are special built-in methods that start with double underscores (__).

They are automatically called by PHP when certain events occur in an object's lifecycle — for example:

- When an object is created or destroyed
- When inaccessible properties are used
- When an object is treated as a string or invoked as a function

They allow developers to add custom behavior to standard PHP operations, giving objects a lot more flexibility and control.

Characteristics of Magic Methods

- Always start with two underscores (e.g., __construct, __get, __set).
- Are automatically invoked — you don’t call them directly.
- Used to control object behavior dynamically.
- Must be declared public (most of the time).

__construct() – The Constructor

- Automatically called when an object is created.
- Used to initialize properties or perform setup tasks.

Example:

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

    public function __construct($name, $age) {
        $this->name = $name;
        $this->age = $age;
        echo "Object created for {$this->name}<br>";
    }
}

$student1 = new Student("John", 20);
```

Output:

```
Object created for John
```

Note:

The constructor runs automatically when the object is instantiated.

__destruct() – The Destructor

- Called automatically when an object is destroyed, or the script ends.
- Useful for closing connections or freeing resources.

Example:

```
class FileHandler {
    public function __construct() {
        echo "Opening file...<br>";
    }

    public function __destruct() {
        echo "Closing file...<br>";
    }
}

$handler = new FileHandler();
echo "Doing some operations...<br>";
```

Output:

```
Opening file...

Doing some operations...

Closing file...
```

Note:

Destructor ensures cleanup happens automatically.

__get() – Accessing Inaccessible Properties

- Called when trying to read a private or non-existent property.

Example:

```
class Person {
    private $data = ["name" => "Alice", "age" => 25];

    public function __get($property) {
        if (array_key_exists($property, $this->data)) {
            return $this->data[$property];
        } else {
            return "Property '{$property}' not found.";
        }
    }
}

$p = new Person();
echo $p->name . "<br>"; // Accessing private property
indirectly
echo $p->gender . "<br>"; // Non-existent property
```

Output:

```
Alice

Property 'gender' not found.
```

Note:

__get() gives you control over what happens when someone tries to access hidden data.

__set() – Setting Inaccessible Properties

- Called when trying to assign a value to a private or non-existent property.

Example:

```
class Student {
    private $info = [];

    public function __set($key, $value) {
        echo "Setting '$key' to '$value'<br>";
        $this->info[$key] = $value;
    }

    public function __get($key) {
        return $this->info[$key] ?? null;
    }
}

$s = new Student();
$s->name = "Emma";
$s->age = 21;

echo $s->name;
```

Output:

```
Setting 'name' to 'Emma'

Setting 'age' to '21'

Emma
```

Note:

You can manage private or dynamic data safely.

__call() – Handling Undefined Methods

- Triggered when an undefined or inaccessible method is called.

Example:

```
class Calculator {
    public function __call($name, $arguments) {
        echo "Method '$name' not found.
Arguments: " . implode(", ", $arguments) .
"<br>";
    }
}

$obj = new Calculator();
$obj->add(5, 10);
```

Output:

```
Method 'add' not found. Arguments: 5, 10
```

Note:

Useful for implementing method overloading or dynamic method handling.

__callStatic() – Handling Undefined Static Methods

- Triggered when an undefined static method is called.

Example:

```
class Math {
    public static function __callStatic($name,
$arguments) {
        echo "Static method '$name' called with
arguments: " . implode(", ", $arguments);
    }
}

Math::multiply(5, 3);
```

Output:

```
Static method 'multiply' called with arguments: 5,
3
```

__toString() – When Object is Treated as a String

- Called when you try to echo an object.

Example:

```
class User {
    public $name;

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

    public function __toString() {
        return "User name: " . $this->name;
    }
}

$user = new User("John");
echo $user;
```

Output:

```
User name: John
```

Note:

Useful for providing meaningful text when printing objects.

__isset() and __unset()

- Handle when isset() or unset() is used on inaccessible properties.

Example:

```
class Example {
    private $data = ["name" => "Sam"];

    public function __isset($key) {
        return isset($this->data[$key]);
    }

    public function __unset($key) {
        unset($this->data[$key]);
        echo "Unset property '$key'<br>";
    }
}

$obj = new Example();
var_dump(isset($obj->name)); // true
unset($obj->name);           // calls __unset()
```

__clone() – Object Cloning

- Called when an object is cloned using the clone keyword.

Example:

```
class Student {
    public $name;

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

    public function __clone() {
        $this->name = "Copy of " . $this->name;
    }
}

$s1 = new Student("Alice");
$s2 = clone $s1;

echo $s2->name;
```

Output:

```
Copy of Alice
```

Note:

Helps manage how an object behaves when duplicated.

__invoke() – When Object is Called Like a Function

- Called when you treat an object as a function.

Example:

```
class Greeting {
    public function __invoke($name) {
        echo "Hello, $name!";
    }
}

$greet = new Greeting();
$greet("John"); // Object called like a function
```

Output:

```
Hello, John!
```

Other Magic Methods

Method	Triggered When	Purpose
`__sleep()`	`serialize()` is used	Prepare object for serialization
`__wakeup()`	`unserialize()` is used	Reinitialize object on unserialize
`__debugInfo()`	`var_dump()` is used	Customize debug output

Practical Applications of OOP in PHP Development

OOP Practical Application

After learning the theory of OOP — **classes, objects, properties, methods, inheritance, polymorphism, abstraction, encapsulation, static, and magic methods** — it's time to see how these concepts are used in real-world PHP development.

Object-Oriented Programming allows developers to build modular, reusable, and maintainable systems — making it easier to manage large applications such as:

- Content Management Systems (CMS)
- E-commerce Platforms
- Student Management Systems
- Authentication System
- Frameworks (like Laravel, Symfony)

Example 1: A Student Management System

Let’s create a small demo project that demonstrates encapsulation, inheritance, static members, and magic methods in action.

Code: Refer to GitHub Repo

OOP Concept	Where Used	Explanation
Class & Object	`Person`, `Student`	Represent people and students
Inheritance	`Student extends Person`	Student inherits properties of Person
Encapsulation	Private/protected properties	Secure and controlled access
Static	`static $studentCount`	Shared property for all objects
Magic Method	`__toString()`	Defines object’s string representation

Example 2: E-commerce Order System

This example demonstrates abstraction and polymorphism.

Code: Refer to GitHub Repo

Note:

Polymorphism: Both CreditCardPayment and PayPalPayment classes use the same pay() method but perform different actions.

Example 3: Logger Class with Magic Methods

Demonstrates dynamic property and string handling.

Code: Refer to GitHub Repo

How OOP is Used in PHP Frameworks

Popular frameworks like Laravel, Symfony, and CodeIgniter heavily use OOP.

Concept	Example in Framework
Class & Object	Controllers, Models, Services
Inheritance	BaseController → UserController
Encapsulation	Private properties in Models
Polymorphism	Different database drivers implementing same interface
Static Methods	Helper classes (`DB::table()`, `Auth::user()`)
Magic Methods	Laravel's Eloquent ORM uses `__get()` and `__call()` for dynamic property access

Any Questions?

Create PHP Project from Scratch

Series: https://laracasts.com/series/php-for-beginners-2023-edition

This programming series is designed for beginner, not just PHP, but programming in general. Here, you will learn to develop web programming from scratch.