Tab 1

# 🧠 What is Object-Oriented Programming in PHP?

OOP lets you organize your code into **classes** (blueprints) and **objects** (real-world things created from blueprints).

## ✅ Step 1: Class and Object in PHP

### **🔧 Real-World Example:**

Let’s say you want to represent a **User** — instead of just using an array, you use a **User class**.

### **📄 User.php**

<?php

// Define a User class (blueprint)

class User {

// Properties (also called data members)

public $name;

public $email;

// Method (also called member function)

public function displayInfo() {

echo "Name: $this->name, Email: $this->email";

}

}

// Create an object from the User class

$user1 = new User();

// Set property values

$user1->name = "Aditya";

$user1->email = "adi@example.com";

// Call method

$user1->displayInfo(); // Output: Name: Aditya, Email: adi@example.com

?>

### **📌 Notes Summary:**

📘 PHP Classes and Objects

✅ Class:

- A blueprint for objects

- Created using class keyword

✅ Object:

- Instance of a class (created using new)

✅ Syntax:

class ClassName {

public $property;

public function method() { ... }

}

$obj = new ClassName();

$obj->property = value;

$obj->method();

✅ $this:

- Refers to the current object inside the class

## **✅ Constructor**

A constructor is a special function that runs **automatically when an object is created.**

### **🔧 Real-World Use Case: Automatically set user details when creating a user object.**

<?php

class User {

public $name;

public $email;

// Constructor runs when object is created

public function \_\_construct($name, $email) {

$this->name = $name; // Assign name to property

$this->email = $email; // Assign email to property

}

public function displayInfo() {

echo "Name: $this->name, Email: $this->email";

}

}

// Create object and set data via constructor

$user1 = new User("Aditya", "adi@example.com");

$user1->displayInfo(); // Output: Name: Aditya, Email: adi@example.com

?>

## 

## 

## 

## **✅ Inheritance**

One class **inherits** properties and methods from another (like a child from a parent).

### **🔧 Real-World Use Case:**

Admin is a special type of User, but with extra privileges.

<?php

// Parent class

class User {

public $name;

public function \_\_construct($name) {

$this->name = $name;

}

public function greet() {

echo "Hello, $this->name";

}

}

// Child class that inherits from User

class Admin extends User {

public function accessPanel() {

echo "$this->name is accessing the admin panel.";

}

}

// Create admin object

$admin = new Admin("Aditya");

// Call method from parent class

$admin->greet(); // Hello, Aditya

echo "<br>";

// Call method from child class

$admin->accessPanel(); // Aditya is accessing the admin panel.

?>

## **✅ Polymorphism**

"One name, many forms" — same method name behaves differently for different classes.

### **🔧 Real-World Use Case:**

Multiple user roles (User, Admin, Moderator) — all greet in their own style.

<?php

class User {

public function greet() {

echo "Hello, user!";

}

}

class Admin extends User {

// Overrides parent greet()

public function greet() {

echo "Welcome back, Admin!";

}

}

class Moderator extends User {

public function greet() {

echo "Greetings, Moderator!";

}

}

// Create polymorphic array

$people = [new User(), new Admin(), new Moderator()];

// Loop through and call greet() — each acts differently

foreach ($people as $person) {

$person->greet();

echo "<br>";

}

?>

### **✅ Output:**

Hello, user!

Welcome back, Admin!

Greetings, Moderator!

## **📝 Summary Notes:**

📘 Constructors

- \_\_construct() is called automatically on object creation

- Used to initialize values

📘 Inheritance

- Use `extends` to inherit from another class

- Child class can access parent’s public & protected methods

📘 Polymorphism

- Child classes override parent methods

- Same method name behaves differently

- Useful in APIs for flexible behavior

## **✅ Visibility in PHP: public, private, protected**

### **🔧 Real-World Scenario:**

You’re building a **Bank Account** class. Some info like **account number** should be protected. Some info like **balance** should be private. Only selected methods should access them securely.

### **📄 BankAccount.php**

<?php

class BankAccount {

// Public property: accessible from anywhere

public $accountHolder;

// Protected: can be accessed only in this class or its children

protected $accountNumber;

// Private: accessible only inside this class

private $balance;

// Constructor to initialize the object

public function \_\_construct($name, $accNo, $initialBalance) {

$this->accountHolder = $name;

$this->accountNumber = $accNo;

$this->balance = $initialBalance;

}

// Public method to show account summary

public function showAccount() {

echo "Account Holder: $this->accountHolder<br>";

echo "Account Number: $this->accountNumber<br>";

echo "Balance: ₹$this->balance<br>";

}

// Public method to deposit amount

public function deposit($amount) {

if ($amount > 0) {

$this->balance += $amount;

echo "Deposited ₹$amount. New Balance: ₹$this->balance<br>";

}

}

// Public method to withdraw

public function withdraw($amount) {

if ($amount <= $this->balance) {

$this->balance -= $amount;

echo "Withdrawn ₹$amount. New Balance: ₹$this->balance<br>";

} else {

echo "Insufficient funds.<br>";

}

}

}

### **✅ Try to Access Outside the Class:**

<?php

$acc = new BankAccount("Aditya", "1234567890", 1000);

// Allowed: public property

echo $acc->accountHolder . "<br>"; // ✅ Works

// Not allowed: protected property

// echo $acc->accountNumber; // ❌ Fatal Error

// Not allowed: private property

// echo $acc->balance; // ❌ Fatal Error

// Use public methods to interact safely

$acc->deposit(500); // ✅ Works

$acc->withdraw(700); // ✅ Works

$acc->showAccount(); // ✅ Works

?>

## **✅ Visibility Levels Summary:**

| **Visibility** | **Accessible in Class** | **Accessible in Child Class** | **Accessible Outside** |
| --- | --- | --- | --- |
| public | ✅ | ✅ | ✅ |
| protected | ✅ | ✅ | ❌ |
| private | ✅ | ❌ | ❌ |

## **📝 Notes Summary**

📘 Visibility in PHP (Access Modifiers)

✅ public

- Accessible from anywhere (inside/outside class)

- Use for safe properties like "name"

✅ protected

- Accessible inside class and subclasses

- Use for sensitive data like account number

✅ private

- Accessible only inside the same class

- Use for very sensitive data like balance

📌 Good Practice:

- Make data private/protected

- Use public methods (getters/setters) to access/update

## **✅ Static vs Non-static in PHP**

| **Aspect** | **Static** | **Non-static** |
| --- | --- | --- |
| Called with | ClassName::method() | $object->method() |
| Memory | Shared across all objects | Separate for each object |
| this keyword | ❌ Not available | ✅ Available |
| Usage | Global counters, config, etc | Data specific to an object |

## **🔧 Real-World Use Case:**

We want:

* **Non-static** methods to handle account operations (deposit(), withdraw())
* A **static** variable to count **how many accounts have been created in total**, regardless of object.

## **✅ Code: BankAccount with Static + Inheritance**

<?php

class BankAccount {

public $accountHolder;

protected $accountNumber;

private $balance;

// Static variable shared by all objects (class-level)

public static $totalAccounts = 0;

// Constructor (non-static): initializes object data

public function \_\_construct($name, $accNo, $initialBalance) {

$this->accountHolder = $name;

$this->accountNumber = $accNo;

$this->balance = $initialBalance;

// Every time an account is created, increment static counter

self::$totalAccounts++;

}

public function showAccount() {

echo "Holder: $this->accountHolder<br>";

echo "Account No: $this->accountNumber<br>";

echo "Balance: ₹$this->balance<br>";

}

public function deposit($amount) {

$this->balance += $amount;

echo "Deposited ₹$amount<br>";

}

public function withdraw($amount) {

if ($amount <= $this->balance) {

$this->balance -= $amount;

echo "Withdrawn ₹$amount<br>";

} else {

echo "Insufficient funds<br>";

}

}

// Static method to get total accounts (no $this allowed here)

public static function getTotalAccounts() {

echo "Total accounts created: " . self::$totalAccounts . "<br>";

}

}

## 

## 

## **✅ SavingAccount (Inheritance + Static Access)**

<?php

// Child class inherits BankAccount

class SavingAccount extends BankAccount {

private $interestRate;

public function \_\_construct($name, $accNo, $initialBalance, $rate) {

// Call parent constructor to set base data

parent::\_\_construct($name, $accNo, $initialBalance);

$this->interestRate = $rate;

}

public function addInterest() {

$interest = $this->interestRate / 100;

$bonus = $this->balance \* $interest;

$this->deposit($bonus);

echo "Interest ₹$bonus added<br>";

}

}

## **✅ Use Static & Non-static Together**

<?php

// Create two accounts

$acc1 = new SavingAccount("Aditya", "ACC001", 1000, 5);

$acc2 = new SavingAccount("John", "ACC002", 2000, 4);

// Use non-static methods

$acc1->deposit(500);

$acc1->withdraw(300);

$acc1->showAccount();

// Add interest using child class method

$acc1->addInterest();

// Static method called without creating object

BankAccount::getTotalAccounts(); // Output: Total accounts created: 2

?>

## **📝 Notes Summary**

📘 Static vs Non-Static in PHP

✅ static

- Belongs to class, not object

- Access with ClassName::$var or ClassName::method()

- Can't use $this

- Example: shared counters, configs

✅ non-static

- Belongs to each object

- Use $this to access properties

- Example: deposit(), withdraw() change object state

📌 Inheritance + static

- Child class can use static methods/vars from parent

- Use parent::\_\_construct() to reuse parent logic

✅ Real Use:

- Static → total accounts

- Non-static → per account data

## **✅ TRAITS**

Traits help avoid **code duplication**. They allow you to "inject" common methods into multiple unrelated classes.

### **🔧 Real-World Use Case:**

Multiple classes (User, Admin, Manager) need logging ability. Instead of duplicating log() in all classes, use a **Trait**.

<?php

// Trait for logging

trait Logger {

public function log($msg) {

echo "[LOG]: $msg<br>";

}

}

// Regular User class using the Logger trait

class User {

use Logger;

public function sayHello() {

$this->log("User says hello");

echo "Hi, I'm a user<br>";

}

}

// Admin class also using Logger

class Admin {

use Logger;

public function deleteUser() {

$this->log("Admin deleted a user");

echo "User deleted<br>";

}

}

$user = new User();

$user->sayHello(); // Logs + message

$admin = new Admin();

$admin->deleteUser(); // Logs + action

?>

✅ **Trait lets you reuse methods without inheritance.** ✅ You can use multiple traits in a single class too.

## **✅ INTERFACES**

An interface defines a **contract**: any class that implements it **must define** the listed methods.

### **🔧 Real-World Use Case:**

Every type of payment class (PayPal, Stripe, UPI) must implement pay() and refund().

<?php

// Define the contract

interface PaymentGateway {

public function pay($amount);

public function refund($amount);

}

// Implementing the interface

class PayPal implements PaymentGateway {

public function pay($amount) {

echo "Paid ₹$amount using PayPal<br>";

}

public function refund($amount) {

echo "Refunded ₹$amount via PayPal<br>";

}

}

class UPI implements PaymentGateway {

public function pay($amount) {

echo "Paid ₹$amount using UPI<br>";

}

public function refund($amount) {

echo "Refunded ₹$amount via UPI<br>";

}

}

// Works polymorphically

function makePayment(PaymentGateway $pg, $amount) {

$pg->pay($amount);

}

makePayment(new PayPal(), 500);

makePayment(new UPI(), 200);

?>

✅ **Interfaces** are used for **polymorphism** (same method, different implementation).

## **✅ ABSTRACT CLASSES**

Abstract classes are **partially implemented classes** — can have both defined and undefined (abstract) methods.

### **🔧 Real-World Use Case:**

A base class for **BankAccount**, where each type of account (Saving, Current) **must** define calculateInterest() but shares common logic.

<?php

// Abstract base class

abstract class BankAccount {

protected $balance;

public function \_\_construct($amount) {

$this->balance = $amount;

}

// Concrete method: shared by all accounts

public function deposit($amt) {

$this->balance += $amt;

echo "Deposited ₹$amt. Balance: ₹$this->balance<br>";

}

// Abstract method: must be defined by subclass

abstract public function calculateInterest();

}

// SavingAccount must implement calculateInterest()

class SavingAccount extends BankAccount {

public function calculateInterest() {

$interest = $this->balance \* 0.05;

echo "Interest on savings: ₹$interest<br>";

}

}

// CurrentAccount must also define it

class CurrentAccount extends BankAccount {

public function calculateInterest() {

echo "No interest for current accounts<br>";

}

}

$acc1 = new SavingAccount(1000);

$acc1->deposit(500);

$acc1->calculateInterest();

$acc2 = new CurrentAccount(3000);

$acc2->calculateInterest();

?>

✅ Abstract class lets you **share logic** + **enforce structure** ✅ Child class **must implement** the abstract methods

## **🧠 Summary Table**

| **Feature** | **Traits** | **Interfaces** | **Abstract Classes** |
| --- | --- | --- | --- |
| Use case | Share methods | Force structure (contract) | Base class with partial implementation |
| Methods | Fully implemented | Only method signatures | Both abstract + implemented methods |
| Multiple use? | Yes (can use many traits) | Yes (can implement many interfaces) | No (can extend only one abstract class) |
| Supports Props | ❌ Traits: methods only | ❌ No properties | ✅ Yes |
| Inheritance | Not related to inheritance | No inheritance, just contract | Inherits with extends |

## **📝 Notes Summary**

📘 Traits:

- Use `trait` keyword

- Use inside class with `use TraitName`

- Good for reusable methods (loggers, helpers)

📘 Interfaces:

- Use `interface` keyword

- All methods must be implemented

- Great for contracts (e.g., PaymentGateway)

📘 Abstract Class:

- Use `abstract class`

- Can define AND force methods

- Child must `extend` and implement abstract methods

## ✅ Namespace

A **namespace** is a way to group related classes so there's **no name conflict**.

### **🧠 Real-World Analogy:**

* Car\Engine and Plane\Engine can both have Engine class, but won’t conflict.

### **✅ Example Without Namespace Conflict**

<?php

// File: Vehicle/Car.php

namespace Vehicle;

class Car {

public function start() {

echo "Car started<br>";

}

}

<?php

// File: Machine/Car.php

namespace Machine;

class Car {

public function run() {

echo "Machine car is running<br>";

}

}

<?php

// File: index.php

require 'Vehicle/Car.php';

require 'Machine/Car.php';

use Vehicle\Car as VehicleCar;

use Machine\Car as MachineCar;

$c1 = new VehicleCar();

$c1->start();

$c2 = new MachineCar();

$c2->run();

✅ Same class name Car, but no conflict due to **namespaces** and use ... as ....

## **✅ 2. What is Autoloading?**

PHP **autoloading** means you don’t have to manually require every class file. PHP will load the file **automatically when you use the class**.

This saves you from:

require 'User.php';

require 'Admin.php';

require 'Database.php';

// and so on...

### 

### 

### 

### 

### **✅ Real-World Folder Structure**

project/

│

├── src/

│ ├── Models/

│ │ └── User.php

│ ├── Services/

│ │ └── AuthService.php

│ └── index.php

│

└── composer.json

## **✅ 3. Full Code Example with Namespaces + Composer Autoload**

### **📄 src/Models/User.php**

<?php

namespace App\Models;

class User {

public function getName() {

return "Aditya Choudhari";

}

}

### 

### 

### 

### 

### **📄 src/Services/AuthService.php**

<?php

namespace App\Services;

use App\Models\User;

class AuthService {

public function login() {

$user = new User();

echo "Logging in " . $user->getName();

}

}

### **📄 src/index.php**

<?php

require \_\_DIR\_\_ . '/../vendor/autoload.php'; // Composer's autoloader

use App\Services\AuthService;

$auth = new AuthService();

$auth->login(); // Output: Logging in Aditya Choudhari

## **✅ 4. Composer Autoload Setup**

### **📄 composer.json**

{

"autoload": {

"psr-4": {

"App\\": "src/"

}

}

}

Then run:

composer dump-autoload

This generates vendor/autoload.php which maps your classes automatically using **PSR-4** standard.

## **✅ Summary**

📘 Namespaces

- Prevent class name conflicts

- Syntax: namespace App\Models;

- Use: use App\Models\User;

📘 Autoloading

- No need to manually include class files

- Use Composer with PSR-4 mapping

- composer.json → autoload → psr-4

- composer dump-autoload (to regenerate map)

📁 Folder Convention

- Namespace: App\Models\User

- Maps to: /src/Models/User.php

📌 Frameworks like Laravel, Symfony rely heavily on this structure

## 

## **🔧 Real-World Use Case:**

A **Banking System** with:

* BankAccount (base class)
* SavingAccount and CurrentAccount (child classes)
* **Only SavingAccount** will implement Taxable interface
* AccountService will **use dependency injection** to operate accounts
* **Namespaces + PSR-4 autoloading**

### **✅ Final Output:**

$account = new SavingAccount(...);

$accountService = new AccountService($account);

$accountService->process();

## **📁 Folder Structure (Backend-Style)**

project/

├── src/

│ ├── Interfaces/

│ │ └── Taxable.php

│ ├── Models/

│ │ ├── BankAccount.php

│ │ ├── SavingAccount.php

│ │ └── CurrentAccount.php

│ ├── Services/

│ │ └── AccountService.php

│ └── index.php

├── composer.json

## **✅ 1. Setup Autoloading (composer.json)**

{

"autoload": {

"psr-4": {

"App\\": "src/"

}

}

}

composer dump-autoload

## **✅ 2. Code Starts Here 🔥**

### **📄 src/Interfaces/Taxable.php**

<?php

namespace App\Interfaces;

// Interface defines contract for taxable accounts

interface Taxable {

public function calculateTax(): float;

}

### **📄 src/Models/BankAccount.php**

<?php

namespace App\Models;

// Base abstract class for all accounts

abstract class BankAccount {

protected string $holder;

protected float $balance;

public function \_\_construct(string $holder, float $balance) {

$this->holder = $holder;

$this->balance = $balance;

}

public function deposit(float $amount): void {

$this->balance += $amount;

}

public function withdraw(float $amount): void {

if ($amount <= $this->balance) {

$this->balance -= $amount;

} else {

echo "Insufficient funds for {$this->holder}<br>";

}

}

public function getBalance(): float {

return $this->balance;

}

abstract public function accountType(): string;

}

### **📄 src/Models/SavingAccount.php**

<?php

namespace App\Models;

use App\Interfaces\Taxable;

// SavingAccount extends base and implements tax

class SavingAccount extends BankAccount implements Taxable {

private float $interestRate;

public function \_\_construct(string $holder, float $balance, float $rate) {

parent::\_\_construct($holder, $balance);

$this->interestRate = $rate;

}

public function addInterest(): void {

$this->balance += $this->balance \* ($this->interestRate / 100);

}

public function calculateTax(): float {

// 10% tax on interest earnings (simplified)

return $this->balance \* 0.10;

}

public function accountType(): string {

return "Saving";

}

}

### **📄 src/Models/CurrentAccount.php**

<?php

namespace App\Models;

// CurrentAccount extends base but is NOT taxable

class CurrentAccount extends BankAccount {

public function accountType(): string {

return "Current";

}

}

### **📄 src/Services/AccountService.php**

<?php

namespace App\Services;

use App\Models\BankAccount;

use App\Interfaces\Taxable;

// Service class to operate on accounts (uses DI)

class AccountService {

private BankAccount $account;

public function \_\_construct(BankAccount $account) {

$this->account = $account;

}

public function process(): void {

echo "Processing " . $this->account->accountType() . " Account<br>";

$this->account->deposit(1000);

$this->account->withdraw(500);

echo "Balance: ₹" . $this->account->getBalance() . "<br>";

// Check if this account is taxable

if ($this->account instanceof Taxable) {

$tax = $this->account->calculateTax();

echo "Tax to be paid: ₹$tax<br>";

} else {

echo "No tax applicable for this account.<br>";

}

}

}

### **📄 src/index.php**

<?php

require \_\_DIR\_\_ . '/../vendor/autoload.php';

use App\Models\SavingAccount;

use App\Models\CurrentAccount;

use App\Services\AccountService;

// Using SavingAccount (taxable)

$saving = new SavingAccount("Aditya", 5000, 4);

$service1 = new AccountService($saving);

$service1->process();

echo "<hr>";

// Using CurrentAccount (non-taxable)

$current = new CurrentAccount("John", 7000);

$service2 = new AccountService($current);

$service2->process();

## **✅ Output:**

Processing Saving Account

Balance: ₹5500

Tax to be paid: ₹550

Processing Current Account

Balance: ₹7500

No tax applicable for this account.

## 

## 

## 

## 

## 

## 

## **🧠 Recap Summary**

### **✅ What You Learned:**

| **Concept** | **Where You Used It** |
| --- | --- |
| ✅ Inheritance | SavingAccount & CurrentAccount → BankAccount |
| ✅ Interface | Taxable → implemented only by SavingAccount |
| ✅ Abstract Class | BankAccount defined deposit, withdraw, etc. |
| ✅ Dependency Injection | AccountService injected with BankAccount |
| ✅ Namespaces & PSR-4 | App\Models, App\Services, App\Interfaces |
| ✅ Autoloading | composer.json + dump-autoload |