**Advanced PHP**

**Exercises**

**OOPS Concepts**

**Q. Define Object-Oriented Programming (OOP) and its four main principles: Encapsulation, Inheritance, Polymorphism, and Abstraction.**

**Object-Oriented Programming (OOP)**

* Object-Oriented Programming (OOP) is a programming paradigm that organizes software design around objects, which are instances of classes.
* Objects encapsulate data (attributes) and behavior (methods) and interact with each other to build complex systems.
* OOP enhances code reusability, scalability, and maintainability.

**Four Main Principles of OOP**

1. **Encapsulation**
   * Encapsulation is the bundling of data (variables) and methods (functions) that operate on the data into a single unit, called a class.
   * It restricts direct access to some components, enforcing data protection and reducing unintended interference.
   * Achieved through access modifiers like private, protected, and public in languages like PHP, Java, and C++.

**Example:**

**Source code:**

**class User {**

**private $password;**

**public function setPassword($pwd) {**

**$this->password=password\_hash($pwd, PASSWORD\_BCRYPT);**

**}**

**public function getPassword() {**

**return "Access Denied";**

**}**

**}**

**$user = new User();**

**$user->setPassword("secret123");**

**echo $user->getPassword();**

**Output: Access Denied**

**2.Inheritance**

* Inheritance allows a class (child) to inherit attributes and methods from another class (parent).
* It promotes code reuse and establishes a relationship between different classes**.**

**Example:**

**Source code:**

**class Animal {**

**public function makeSound() {**

**return "Some generic sound";**

**}**

**}**

**class Dog extends Animal {**

**public function makeSound() {**

**return "Bark!";**

**}**

**}**

**$dog = new Dog();**

**echo $dog->makeSound();**

**Output: Bark!**

**3.Polymorphism**

* Polymorphism allows objects to be treated as instances of their parent class, even when they refer to child class objects.
* It enables method overriding and method overloading, allowing different behaviors for the same method name.

**Example:**

**Source code:**

**class Shape {**

**public function area() {**

**return "Calculating area...";**

**}**

**}**

**class Circle extends Shape {**

**private $radius;**

**public function \_\_construct($radius) {**

**$this->radius = $radius;**

**}**

**public function area() {**

**return pi() \* $this->radius \* $this->radius;**

**}**

**}**

**$shape = new Circle(5);**

**echo $shape->area();**

**Output: 78.54**

**4.Abstraction**

* Abstraction hides complex implementation details and only exposes essential functionalities.
* Achieved using abstract classes and interfaces.

**Example:**

**Source code:**

**abstract class Vehicle {**

**abstract public function start();**

**}**

**class Car extends Vehicle {**

**public function start() {**

**return "Car engine started!";**

**}**

**}**

**$car = new Car ();**

**echo $car->start ();**

**Output: Car engine started!**

**Practical Exercise: Create a simple class in PHP that demonstrates encapsulation by using private and public properties and methods.**

**Source code:**

**class BankAccount**

**{**

**private $accountNumber;**

**private $balance;**

**publicfunction\_\_construct($accountNumber, $initialBalance)**

**{**

**$this->accountNumber = $accountNumber;**

**$this->balance = $initialBalance;**

**}**

**public function deposit($amount)**

**{**

**if ($amount > 0) {**

**$this->balance += $amount;**

**echo "<p>Deposited: $amount.</p>";**

**echo "New Balance: $this->balance</p>";**

**} else {**

**echo "<p>Invalid deposit amount.</p>";**

**}**

**}**

**public function withdraw($amount)**

**{**

**if ($amount > 0 && $amount <= $this->balance) {**

**$this->balance -= $amount;**

**echo "<p>Withdrawn: $amount. Remaining Balance: $this->balance</p>";**

**} else {**

**echo "<p>Invalid withdrawal amount or insufficient balance.</p>";**

**}**

**}**

**public function getBalance()**

**{**

**return $this->balance;**

**}**

**}**

**// Example usage**

**$account = new BankAccount("123456789", 1000);**

**echo "<p> New Account created wih Initial Balance: " . $account->getBalance() . "</p>";**

**// New Account created wih Initial Balance: 1000**

**$account->deposit(500);**

**// Deposited: 500. New Balance: 1500**

**$account->withdraw(200);**

**// Withdrawn: 200. Remaining Balance: 1300**

**echo "<p>Final Balance: " . $account->getBalance () . "</p>";**

**// Final Balance: 1300**

**Class:**

**Explain the structure of a class in PHP, including properties and methods.**

S**tructure of a Class in PHP**

A class in PHP serves as a blueprint for creating objects. It defines properties (variables) and methods (functions) that describe an object's behavior**.**

**Basic Structure of a PHP**

**Example:**

**Source code:**

**class ClassName {**

**// Properties (attributes)**

**public $property1;**

**private $property2;**

**// Constructor (optional)**

**public function \_\_construct($value) {**

**$this->property1 = $value;**

**}**

**// Methods (functions)**

**public function method1() {**

**return "This is a public method.";**

**}**

**private function method2() {**

**return "This is a private method.";**

**}**

**}**

**// Creating an Object**

**$object = new ClassName("Hello");**

**echo $object->method1();**

**Output: This is a public method.**

**Components of a PHP Class**

1. **Class Declaration**
   * Defined using the class keyword, followed by a class name (PascalCase naming convention is preferred).
2. **Properties (Attributes)**
   * Variables that hold the object's data.
   * Can have different access modifiers:
     + public → Accessible from anywhere.
     + private → Accessible only within the class.
     + protected → Accessible within the class and subclasses.
3. **Methods (Functions)**
   * Define the behavior of the object.
   * Can also have public, private, and protected access modifiers.
4. **Constructor (\_\_construct)**
   * A special method that is automatically called when an object is created.
   * Used for initializing properties.
5. **Destructor (\_\_destruct)**
   * A special method that is called when an object is destroyed or script execution ends.
   * Useful for cleanup tasks like closing database connections.

Ex:

Source code:

**public function \_\_destruct() {**

**echo "Object is being destroyed!";**

**}**

**6.Encapsulation with Getters & Setters**

* Used to control access to private properties.

**Ex:**

**class User {**

**private $name;**

**public function setName($name) {**

**$this->name = $name;**

**}**

**public function getName() {**

**return $this->name;**

**}**

**}**

**$user = new User();**

**$user->setName("rimpal");**

**echo $user->getName();**

**Output: rimpal**

**Example: A Real-World Class (Car)**

**class Car {**

**private $brand;**

**private $speed;**

**public function \_\_construct($brand, $speed) {**

**$this->brand = $brand;**

**$this->speed = $speed;**

**}**

**public function getCarInfo() {**

**return "The {$this->brand} is moving at {$this->speed} km/h.";**

**}**

**}**

**// Creating an object**

**$myCar = new Car("Toyota", 120);**

**echo $myCar->getCarInfo();**

**Output: The Toyota is moving at 120 km/h.**

**Conclusion**

* A class acts as a blueprint for objects.
* Properties store object data, and methods define behaviors.
* Encapsulation using private properties and getter/setter methods enhances security.
* Constructors help initialize values at object creation.

This structured approach makes PHP code more scalable, reusable, and maintainable.

**Practical Exercise: Write a PHP script to create a class representing a "Car" with properties like make, model, and year, and a method to display the car details.**

Ex:

Source code:

**class Car**

**{**

**private $make;**

**private $model;**

**private $year;**

**public function \_\_construct($make, $model, $year)**

**{**

**$this->make = $make;**

**$this->model = $model;**

**$this->year = $year;**

**}**

**public function displayDetails()**

**{**

**echo "<p>Car Details: $this->year $this->make $this->model</p>";**

**}**

**}**

**// Example usage**

**$car1 = new Car("Toyota", "Corolla", 2018);**

**$car1->displayDetails(); // Car Details: 2018 Toyota Corolla**

**Object:**

**Q**. **What is an object in OOP? Discuss how objects are instantiated from classes in PHP.**

**What is an Object in OOP?**

* An object in Object-Oriented Programming (OOP) is an instance of a class.
* It represents a real-world entity with properties (attributes) and methods (behaviors) defined by the class.
* Objects encapsulate data and functionality, making the code more modular and reusable.

Instantiating Objects from Classes in PHP

* In PHP, an object is created using the new keyword, which instantiates a class. Here’s how it works:

**Example: Creating and Using an Object in PHP**

**Source code:**

**// Defining a class**

**class Car {**

**public $brand;**

**public $model;**

**// Constructor to initialize properties**

**public function \_\_construct($brand, $model) {**

**$this->brand = $brand;**

**$this->model = $model;**

**}**

**// Method to display car details**

**public function display() {**

**echo "Car: $this->brand $this->model";**

**}**

**}**

**// Creating an object (instantiating a class)**

**$car1 = new Car("Toyota", "Camry");**

**// Accessing object properties and methods**

**echo $car1->display();**

**Output: Car: Toyota Camry**

**Practical Exercise: Instantiate multiple objects of the "Car" class and demonstrate how to access their properties and methods.**

**Source code:**

**class Car**

**{**

**private $make;**

**private $model;**

**private $year;**

**public function \_\_construct($make, $model, $year)**

**{**

**$this->make = $make;**

**$this->model = $model;**

**$this->year = $year;**

**}**

**public function displayDetails()**

**{**

**echo "<p>Car Details: $this->year $this->make $this->model</p>";**

**}**

**}**

**// Example usage**

**$car1 = new Car("Toyota", "Corolla", 2018);**

**$car1->displayDetails(); // Car Details: 2018 Toyota Corolla**

**$car2 = new Car("Honda", "Civic", 2019);**

**$car2->displayDetails(); // Car Details: 2019 Honda Civic**

**$car3 = new Car("Suzuki", "Swift", 2020);**

**$car3->displayDetails(); // Car Details: 2020 Suzuki Swift**

**$car4 = new Car("Hyundai", "Accent", 2021);**

**$car4->displayDetails(); // Car Details: 2021 Hyundai Accent**

**Extends:**

**Q. Explain the concept of inheritance in OOP and how it is implemented in PHP.**

**Concept of Inheritance in OOP**

* Inheritance is a fundamental concept in Object-Oriented Programming (OOP) that allows one class (child/subclass) to inherit properties and methods from another class (parent/superclass).
* It promotes code reusability, hierarchical relationships,and extensibility in software design.
* With inheritance, a child class can:
  + Use methods and properties of the parent class.
  + Override or extend the functionality of inherited methods.
  + Introduce new methods specific to itself.

**Implementing Inheritance in PHP**

* In PHP, inheritance is implemented using the extends keyword.
* The child class inherits all public and protected properties and methods of the parent class.

**Example: Inheritance in PHP**

1. extends Keyword – Used to define a subclass that inherits from a parent class.
2. Method Overriding – A child class can redefine (override) a method from the parent class.
3. Protected Members – Properties/methods marked as protected can be accessed within child classes but not outside.
4. Parent Constructor Call – The child class can call the parent’s constructor using parent::\_\_construct().
5. **Example:**

**Using parent::\_\_construct()**

**Source code:**

**class Vehicle {**

**protected $brand;**

**public function \_\_construct($brand) {**

**$this->brand = $brand;**

**}**

**public function getBrand() {**

**return $this->brand;**

**}**

**}**

**class Car extends Vehicle {**

**private $model;**

**public function \_\_construct($brand, $model) {**

**parent::\_\_construct($brand); // Call parent constructor**

**$this->model = $model;**

**}**

**public function getCarDetails() {**

**return "Car: " . $this->getBrand() . " " . $this->model;**

**}**

**}**

**// Creating an object**

**$car = new Car("Toyota", "Camry");**

**echo $car->getCarDetails();**

**Output: Car: Toyota Camry**

**Benefits of Inheritance**

* **Code Reusability** – Avoids redundant code by reusing parent class logic.
* **Better Organization**– Creates hierarchical relationships among classes.
* **Scalability**– Easily extend and maintain functionality.
* **Encapsulation –** Protects data by allowing controlled access through inherited methods.

**Practical Exercise: Create a "Vehicle" class and extend it with a "Car" class. Include properties and methods inboth classes, demonstrating inherited behavior.**

**Source code:**

**class Vehicle**

**{**

**protected $make;**

**protected $model;**

**protected $year;**

**public function \_\_construct($make, $model, $year)**

**{**

**$this->make = $make;**

**$this->model = $model;**

**$this->year = $year;**

**}**

**public function displayDetails()**

**{**

**echo "<p>Vehicle Details: $this->year $this->make $this->model</p>";**

**}**

**}**

**class Car extends Vehicle**

**{**

**private $color;**

**public function \_\_construct($make, $model, $year, $color)**

**{**

**parent::\_\_construct($make, $model, $year);**

**$this->color = $color;**

**}**

**public function displayDetails()**

**{**

**echo "<p>Car Details: $this->year $this->make $this->model $this->color</p>";**

**}**

**}**

**// Example usage**

**$car1 = new Car("Toyota", "Corolla", 2018, "Red");**

**$car1->displayDetails(); // Car Details: 2018 Toyota Corolla Red**

**$car2 = new Car("Honda", "Civic", 2019, "Blue");**

**$car2->displayDetails(); // Car Details: 2019 Honda Civic Blue**

**$vehicle1 = new Vehicle("Toyota", "Corolla", 2018);**

**$vehicle1->displayDetails(); // Vehicle Details: 2018 Toyota Corolla**

**$vehicle2 = new Vehicle("Honda", "Civic", 2019);**

**$vehicle2->displayDetails(); // Vehicle Details: 2019 Honda Civic**

**Overloading:**

**Q. Discuss method overloading and how it is implemented in PHP.**

**Method Overloading in PHP**

* Method Overloading is a feature in Object-Oriented Programming (OOP) that allows multiple methods with the same name but different parameters.
* While languages like Java and C++ support true method overloading, PHP does not support traditional method overloading directly.
* However, PHP provides magic methods **(\_\_call() and \_\_callStatic())** to achieve dynamic method handling, allowing objects to handle method calls that are not explicitly defined.

**Implementing Method Overloading using \_\_call()**

* PHP uses the \_\_call() method to handle undefined or overloaded instance methods.
* Example: Method Overloading with \_\_call()

**Source code:**

**class Calculator {**

**// Magic method to handle method overloading**

**public function \_\_call($name, $arguments) {**

**if ($name == "add") {**

**switch (count($arguments)) {**

**case 2:**

**return $arguments[0] + $arguments[1]; // Add two numbers**

**case 3:**

**return $arguments[0] + $arguments[1] + $arguments[2]; // Add three numbers**

**default:**

**return "Invalid number of arguments!";**

**}**

**}**

**}**

**}**

**// Creating an object**

**$calc = new Calculator();**

**echo $calc->add(10, 20) . PHP\_EOL;**

**Output: 30**

**echo $calc->add(5, 15, 25) . PHP\_EOL;**

**Output: 45**

**echo $calc->add(5) . PHP\_EOL;**

**Output: Invalid number of arguments!**

**Overloading Static Methods using \_\_callStatic()**

* If a method is called statically but does not exist, \_\_callStatic() is triggered.
* Example: Static Method Overloading\*\*

source code:

**class MathOperations {**

**public static function \_\_callStatic($name, $arguments) {**

**if ($name == "multiply") {**

**return array\_product($arguments); // Multiply all numbers**

**}**

**}**

**}**

**// Calling overloaded static method**

**echo MathOperations::multiply(2, 3) . PHP\_EOL; Output: 6**

**echo MathOperations::multiply(2, 3, 4) . PHP\_EOL; Output: 24**

**Key Points About Method Overloading in PHP**

1. PHP does not support traditional method overloading (same method name with different parameters).
2. Magic methods \_\_call() and \_\_callStatic() handle undefined method calls dynamically.
3. \_\_call() is used for instance methods, while \_\_callStatic() is used for static methods.
4. Allows dynamic method handling, useful for APIs, frameworks, and flexible function parameters.

**Advantages of Method Overloading**

* Provides flexibility by allowing methods with different numbers of arguments.
* Helps create dynamic and scalable class methods.
* Reduces repetitive code by handling similar logic within a single function.

**Practical Exercise: Create a class that demonstrates method overloading by defining multiple methods with the same name but different parameters.**

**Sources code:**

**class AddNumbers**

**{**

**public function \_\_call($methodname, $args)**

**{**

**if ($methodname == 'add') {**

**switch (count($args)) {**

**case 2:**

**return $args[0] + $args[1];**

**case 3:**

**return $args[0] + $args[1] + $args[2];**

**case 4:**

**return $args[0] + $args[1] + $args[2] + $args[3];**

**default:**

**return "Invalid number of arguments";**

**}**

**} else {**

**echo "<p>Method does not exist</p>";**

**}**

**}**

**}**

**// Examples**

**$newadd = new AddNumbers();**

**echo "<p>" . $newadd->add(2) . "</p>"; // Invalid number of arguments**

**echo "<p>" . $newadd->add(25, 35) . "</p>"; // 60**

**echo "<p>" . $newadd->add(10, 20, 30) . "</p>"; // 60**

**echo "<p>" . $newadd->add(20, 40, 60, 80) . "</p>"; // 200**

**echo "<p>" . $newadd->add(15, 25, 35, 45, 55) . "</p>"; // Invalid number of arguments**

**Abstraction Interface**

**Q. Explain the concept of abstraction and the use of interfaces in PHP.**

**Abstraction in PHP**

* Abstraction is an Object-Oriented Programming (OOP) concept that hides implementation details and only exposes relevant functionality.
* It allows developers to define a blueprint for classes without specifying how the methods are implemented.
* In PHP, abstraction is achieved using abstract classes and interfaces.
* An abstract class:
  + Cannot be instantiated.
  + Can have both fully defined and abstract (undefined) methods.
  + Must be extended by a subclass, which provides implementations for abstract methods.
* **Example: Abstract Class in PHP**

**abstract class Animal {**

**// Abstract method (must be implemented in child classes)**

**abstract public function makeSound();**

**// Concrete method (already defined)**

**public function sleep() {**

**return "Sleeping...";**

**}**

**}**

**class Dog extends Animal {**

**public function makeSound() {**

**return "Dog barks";**

**}**

**}**

**$dog = new Dog();**

**echo $dog->makeSound();**

**Output: Dog barks**

**echo $dog->sleep();**

**Output: Sleeping...**

**Key Benefits of Abstract Classes:**

* Enforces a structure for subclasses.
* Allows partial implementation (abstract + concrete methods).
* Useful for defining base classes with shared functionality.

**Interface in PHP**

* **An interface:**
  + Defines a contract for classes without providing any method implementation.
  + Only contains abstract methods (all methods must be public).
  + A class must implement all methods defined in the interface.
* **Example: Interface in PHP**

**interface Vehicle {**

**public function start();**

**public function stop();**

**}**

**class Car implements Vehicle {**

**public function start() {**

**return "Car is starting...";**

**}**

**public function stop() {**

**return "Car is stopping...";**

**}**

**}**

**$myCar = new Car();**

**echo $myCar->start();**

**-->Output: Car is starting...**

**echo $myCar->stop();**

**--->Output: Car is stopping.**

#### **Key Benefits of Interfaces**:

* Supports **multiple inheritance** (a class can implement multiple interfaces).
* Ensures **strict implementation** of required methods.
* Promotes **loose coupling** and **scalability**.

#### Difference Between Abstract Classes and Interfaces\*\*

| **Feature** | **Abstract Class** | **Interface** |
| --- | --- | --- |
| **Can have method implementations?** | ✅ Yes | ❌ No (all methods must be abstract) |
| **Can have properties?** | ✅ Yes | ❌ No (only method definitions) |
| **Supports multiple inheritance?** | ❌ No | ✅ Yes (a class can implement multiple interfaces) |
| **Use case** | Partial implementation (shared functionality) | Enforces method structure (multiple behaviors) |

#### Using Both Abstract Classes and Interfaces Together\*\*

* You can **combine abstract classes and interfaces** for flexibility.
* Example: Combining Abstract Classes and Interfaces

**Source code:**

**interface Engine {**

**public function start();**

**}**

**abstract class Vehicle {**

**abstract public function speed();**

**public function fuel() {**

**return "Using fuel...";**

**}**

**}**

**class Car extends Vehicle implements Engine {**

**public function start() {**

**return "Engine started!";**

**}**

**public function speed() {**

**return "Car is moving at 80 km/h";**

**}**

**}**

**$car = new Car();**

**echo $car->start(); // Output: Engine started!**

**echo $car->speed(); // Output: Car is moving at 80 km/h**

**echo $car->fuel(); // Output: Using fuel…**

* **Abstract class** provides a **base structure** (common functionality).
* **Interface** ensures **specific behaviors** (engine behavior for all vehicles).
* **Abstraction** hides implementation details while providing essential functionality.
* **Abstract classes** are useful when you need **partially implemented base classes.**
* **Interfaces**define**strict contracts**that enforce method implementation.
* **Combining both** provides flexibility and maintainability in large applications.

### **Practical Exercise: Define an interface named VehicleInterface with methods like start(), stop(), and implement this interface in multiple classes.**

**Source code:**

**interface Vehicle**

**{**

**public function start();**

**public function stop();**

**}**

**class Car implements Vehicle**

**{**

**public $brand;**

**public $model;**

**public function \_\_construct($brand, $model)**

**{**

**$this->brand = $brand;**

**$this->model = $model;**

**}**

**public function start()**

**{**

**return "Car is starting...";**

**}**

**public function stop()**

**{**

**return "Car is stopping...";**

**}**

**}**

**class HeavyVehicle implements Vehicle**

**{**

**public $brand;**

**public $model;**

**public $loadcapacity;**

**public function \_\_construct($brand, $model, $loadcapacity)**

**{**

**$this->brand = $brand;**

**$this->model = $model;**

**$this->loadcapacity = $loadcapacity;**

**}**

**public function start()**

**{**

**return "Heavy vehicle starting ...";**

**}**

**public function stop()**

**{**

**return "Heavy vehicle stopping ...";**

**}**

**}**

**// Usage example**

**$myCar = new Car("Maruti", "Swift");**

**echo "<p>" . $myCar->start() . "</p>"; // Output: Car is starting...**

**echo "<p>" . $myCar->stop() . "</p>"; // Output: Car is stopping...**

**$myTruck = new HeavyVehicle("Tata", "4025", "40000");**

**echo "<p>" . $myTruck->start() . "</p>"; // Output: Heavy vehicle starting ...**

**echo "<p>" . $myTruck->stop() . "</p>"; // Output: Heavy vehicle stopping …**

## **Constructor:**

### **Q. What is a constructor in PHP? Discuss its purpose and how it is used.**

* In PHP, a **constructor** is a special method within a class that is automatically called when a new object of that class is created.
* The constructor's primary purpose is to **initialize the object's state** when it is created.
* This means setting values for the object's properties or executing any setup tasks that are required for the object to function properly.
* A constructor is defined using the \_\_construct() method in a class.
* The \_\_construct method is called automatically when a new instance of the class is created using the new keyword.

**Source code:**

**class MyClass {**

**public $name;**

**// Constructor**

**public function \_\_construct($name) {**

**$this->name = $name; // Initialize the property**

**echo "Object created with name: " . $this->name . "\n";**

**}**

**}**

**// Creating an instance of the class**

**$obj = new MyClass("rimi");**

* PHP does not support **constructor overloading**, meaning you cannot have multiple constructors with different parameter types.
* If you need to handle different scenarios, you can use optional parameters or if conditions inside the constructor.

**Source code:**

**class Rectangle {**

**public $width;**

**public $height;**

**// Constructor with optional parameters**

**public function \_\_construct($width = 5, $height = 10) {**

**$this->width = $width;**

**$this->height = $height;**

**}**

**}**

**$rect1 = new Rectangle(); // Uses default values**

**$rect2 = new Rectangle(8, 12); // Custom values**

### **Practical Exercise: Create a class with a constructor that initializes properties when an object is created.**

**Source code:**

**class Car**

**{**

**public $make;**

**public $model;**

**// Constructor with parameters**

**public function \_\_construct($make = "Maruti", $model = "Swift")**

**{**

**$this->make = $make;**

**$this->model = $model;**

**}**

**}**

**// Creating an object with parameters**

**$myCar = new Car("Toyota", "Corolla");**

**echo "<p>" . $myCar->make . "</p>"; // Output: Toyota**

**echo "<p>" . $myCar->model . "</p>"; // Output: Corolla**

**$myDefaultCar = new Car();**

**echo "<p>" . $myDefaultCar->make . "</p>"; // Output: Maruti**

**echo "<p>" . $myDefaultCar->model . "</p>"; // Output: Swift**

## **Destructor:**

### **Q. Explain the role of a destructor in PHP and when it is called.**

* In PHP, a **destructor** is a special method within a class that is automatically called when an object is destroyed, meaning when it is no longer in use or when the script ends.
* The purpose of the destructor is to **clean up** resources, perform final tasks, or release memory that was allocated during the lifetime of the object.
* **Name**: The destructor method in PHP is always named \_\_destruct().
* **Automatic Invocation**: It is automatically invoked when the object goes out of scope or when the script ends.
* **No Parameters**: The destructor method does not accept any parameters.
* **Memory Management**: It helps in cleaning up resources such as closing database connections, releasing file handles, or freeing any other resources that need explicit cleanup.

#### **When is a Destructor Called?**

* The destructor is called when an object is no longer referenced or when it goes out of scope.
* This can happen in various ways:
  + **Unsetting the object** using unset().
  + **When the object goes out of scope**, typically at the end of a function or method.
  + **At the end of the script execution**, PHP will automatically clean up and call the destructor if necessary.
* PHP uses a garbage collector to manage memory.
* When objects are no longer in use (i.e., there are no more references to them), the garbage collector may destroy them, triggering the destructor.
* However, it’s important to note that the exact timing of garbage collection is not guaranteed.
* **Example of a Destructor:**

**Source code:**

**class FileHandler {**

**private $file;**

**public function \_\_construct($filename) {**

**$this->file = fopen($filename, "w"); // Open a file**

**echo "File opened successfully.\n";**

**}**

**public function write($data) {**

**fwrite($this->file, $data); // Write data to the file**

**}**

**// Destructor to close the file**

**public function \_\_destruct() {**

**fclose($this->file); // Close the file**

**echo "File closed successfully.\n";**

**}**

**}**

**$fileHandler = new FileHandler("example.txt");**

**$fileHandler->write("Hello, world!");**

**// Destructor will be called automatically when the object goes out of scope (end of script)**

In the example above:

* The **constructor** opens a file for writing.
* The **destructor** automatically closes the file when the object is no longer needed, freeing up the file handle.

#### Destructor Use Cases:

1. **Closing Open Files**: If your program opens files, you want to close them when the object is destroyed, to ensure that resources are released properly.
2. **Database Connections**: When your object interacts with a database, the destructor can be used to close the database connection once the object is destroyed.
3. **Releasing Other Resources**: If the object uses any other external resources (e.g., network connections, caching mechanisms), the destructor can ensure those resources are properly cleaned up.

#### **Example with Multiple Objects:**

* If you have multiple objects, each with its own destructor, PHP will call the destructors for each object as they are destroyed (typically when the script finishes or objects go out of scope).

**Source code:**

**class DatabaseConnection {**

**public function \_\_construct() {**

**echo "Database connection established.\n";**

**}**

**public function \_\_destruct() {**

**echo "Database connection closed.\n";**

**}**

**}**

**class Cache {**

**public function \_\_construct() {**

**echo "Cache initialized.\n";**

**}**

**public function \_\_destruct() {**

**echo "Cache cleared.\n";**

**}**

**}**

**$db = new DatabaseConnection();**

**$cache = new Cache();**

// Destructor for both objects will be automatically called when the script ends or objects are unset() function:

* **No Return Value**: Destructors cannot return values; they are purely for cleanup.
* **Destructors and Object References**: If there are circular references between objects (e.g., one object references another and vice versa), PHP’s garbage collector will handle this, and destructors will still be called when the objects are no longer needed.
* **Timing**: The exact timing of the destructor is not predictable because PHP uses a reference-counting garbage collector. It is typically called at the end of the script or when an object is no longer referenced.
* If you want to manually destroy an object and invoke its destructor early, you can use the unset() function:

**Source code:**

**$obj = new FileHandler("test.txt");**

**unset($obj); // Destructor is called here to close the file**

### **Practical Exercise: Write a class that implements a destructor to perform cleanup tasks when an object is destroyed.**

**Source code:**

**class DatabaseConnection**

**{**

**public function \_\_construct ()**

**{**

**echo "Database connection established.\n";**

**}**

**public function \_\_destruct ()**

**{**

**echo "Database connection closed.\n";**

**}**

**}**

**class Cache**

**{**

**public function \_\_construct ()**

**{**

**echo "Cache initialized.\n";**

**}**

**public function \_\_destruct()**

**{**

**echo "Cache cleared.\n";**

**}**

**}**

**$db = new DatabaseConnection(); // Database connection established.**

**echo "<br>";**

**unset($db); // Database connection closed.**

**echo "<br>";**

**$cache = new Cache (); // Cache initialized.**

**echo "<br>";**

**unset($cache); // Cache cleared.**

**Magic Methods:**

**Q. Define magic methods in PHP. Discuss commonly used magic methods like \_\_get (), \_\_set (), and \_\_construct ().**

**Magic Methods in PHP**

* **Magic methods** in PHP are special methods that start with double underscores (\_\_) and are automatically called when certain actions are performed on objects.
* These methods allow you to customize the behavior of your classes for common operations like object initialization, property access, method calls, and more.
* Magic methods are part of the **Object-Oriented Programming (OOP)** functionality in PHP, and they help you interact with objects in a dynamic way without needing to explicitly define every single method for each operation.
* Here’s a look at some commonly used magic methods, including \_\_construct (), \_\_get (), and \_\_set():

**1. \_\_construct () — The Constructor**

* Purpose: The \_\_construct () method is the constructor of a class, automatically invoked when a new instance of the class is created.
* It is used to initialize the object and set up initial values for its properties.
* Syntax:

public function \_\_construct([parameters]) { }

* Usage:
  + Initializes object properties when an object is created.
  + Can accept parameters to initialize the object with specific values.
* Example:

**Source code:**

**class User {**

**public $name;**

**public function \_\_construct($name) {**

**$this->name = $name;**

**echo "User {$this->name} created.\n";**

**}**

**}**

**$user = new User("John"); // Calls \_\_construct() and outputs "User John created."**

**2. \_\_get() — Getting Properties Dynamically**

* **Purpose:** The \_\_get() method is called automatically when attempting to access a non-existing or inaccessible property of an object.
* This allows you to define how property values are retrieved, even if the property doesn’t exist or isn’t publicly accessible.
* **Syntax:**

public function \_\_get($name) { }

* **Usage**:
  + Used for **dynamic property access** (i.e., when a property is not directly accessible).
  + Can be used to return values for "virtual" or "computed" properties.

**Example:**

**Source code:**

**class Person {**

**private $data = [**

**'firstName' => 'John',**

**'lastName' => 'Doe'**

**];**

**public function \_\_get($name) {**

**if (isset($this->data[$name])) {**

**return $this->data[$name];**

**}**

**return "Property not found!";**

**}**

**}**

**$person = new Person();**

**echo $person->firstName; // Outputs: John**

**echo $person->lastName; // Outputs: Doe**

**echo $person->age; // Outputs: Property not found!**

**3. \_\_set() — Setting Properties Dynamically**

* **Purpose:** The \_\_set() method is called automatically when attempting to set a non-existing or inaccessible property of an object.
* Like \_\_get(), it allows you to define custom behavior when setting values for properties that do not directly exist.
* **Syntax:**

public function \_\_set($name, $value) { }

* **Usage**:
  + Used for **dynamic property assignment**.
  + Useful for situations where you want to control the assignment of properties (e.g., validation, transformation).
* Example:

**class Product {**

**private $data = [];**

**public function \_\_set($name, $value) {**

**$this->data[$name] = strtoupper($value); // Convert value to uppercase**

**}**

**public function \_\_get($name) {**

**return $this->data[$name] ?? null;**

**}**

**}**

**$product = new Product();**

**$product->name = "laptop"; // Calls \_\_set() and stores "LAPTOP"**

**echo $product->name; // Outputs: LAPTOP**

**4. \_\_call() — Handling Undefined Method Calls**

* **Purpose:** The \_\_call() method is invoked when an attempt is made to call a method that does not exist or is not accessible in the object.
* It allows you to handle such calls dynamically.
* **Syntax:**

**public function \_\_call($name, $arguments) { }**

* **Usage:**
  + Useful for handling dynamic method names or routing method calls to different logic.
* **Example:**

**class Calculator {**

**public function \_\_call($name, $arguments) {**

**if ($name == 'add') {**

**return array\_sum($arguments);**

**}**

**return "Method not found.";**

**}**

**}**

**$calc = new Calculator();**

**echo $calc->add(5, 3, 7); // Outputs: 15**

**5. \_\_toString() — Converting Object to String**

* **Purpose:** The \_\_toString() method is invoked when an object is treated as a string (e.g., when echoing an object).
* It allows you to define a custom string representation for an object.
* **Syntax:**

**public function \_\_toString() { }**

* **Usage:**
  + Enables the object to be printed as a string by defining how it should be represented.
* **Example:**

**class Person {**

**public $name;**

**public $age;**

**public function \_\_toString() {**

**return "Name: {$this->name}, Age: {$this->age}";**

**}**

**}**

**$person = new Person();**

**$person->name = "Alice";**

**$person->age = 30;**

**echo $person; // Outputs: Name: Alice, Age: 30**

**6. \_\_isset() — Checking if a Property is Set**

* **Purpose:** The \_\_isset() method is called when isset() is called on a non-existing or inaccessible property of an object.
* It allows you to define custom logic to check if a property is set.
* **Syntax:**

**public function \_\_isset($name) { }**

* **Usage:**
  + Can be used for custom logic when checking if a property exists.

Example:

**class User {**

**private $data = ['username' => 'admin'];**

**public function \_\_isset($name) {**

**return isset($this->data[$name]);**

**}**

**}**

**$user = new User();**

**var\_dump(isset($user->username)); // Outputs: bool(true)**

**var\_dump(isset($user->password)); // Outputs: bool(false)**

* Magic methods provide a powerful way to interact with objects dynamically. They allow you to:
  + Implement custom behaviors for properties and methods that might not exist at compile time.
  + Enforce validation, logging, or transformations on dynamic data.
  + Create flexible, reusable code that can handle unknown or changing inputs.

### **Practical Exercise: Create a class that uses magic methods to handle property access and modification dynamically.**

**Source code:**

**class User**

**{**

**public $fname;**

**public $lname;**

**public $email;**

**private $pwd;**

**public $dob;**

**public function \_\_construct($fname, $lname, $email, $pwd, $dob)**

**{**

**$this->fname = $fname;**

**$this->lname = $lname;**

**$this->email = $email;**

**$this->pwd = $pwd;**

**$this->dob = $dob;**

**}**

**public function \_\_call($name, $args)**

**{**

**if ($name == 'save') {**

**return "Method not found";**

**}**

**return "Method not found";**

**}**

**public function create()**

**{**

**print("User account created...");**

**}**

**public function display()**

**{**

**print\_r(array(**

**'fname' => $this->fname,**

**'lname' => $this->lname,**

**'email' => $this->email,**

**'dob' => $this->dob**

**));**

**}**

**}**

**<form action="" method="post">**

**<table class="table table-bordered border-dark text-center">**

**<tr>**

**<td><label for="fname" class="form-control">First Name</label></td>**

**<td><input type="text" name="fname" class="form-control" required></td>**

**</tr>**

**<tr>**

**<td><label for="lname" class="form-control">Last Name</label></td>**

**<td><input type="text" name="lname" class="form-control" required></td>**

**</tr>**

**<tr>**

**<td><label for="email" class="form-control">Email Address</label></td>**

**<td><input type="email" name="email" class="form-control" required></td>**

**</tr>**

**<tr>**

**<td><label for="pwd" class="form-control">Password</label></td>**

**<td><input type="password" name="pwd" class="form-control" required></td>**

**</tr>**

**<tr>**

**<td><label for="con-pwd" class="form-control">Confirm Password</label></td>**

**<td><input type="password" name="con-pwd" class="form-control" required></td>**

**</tr>**

**<tr>**

**<td><label for="dob" class="form-control">Date of Birth</label></td>**

**<td><input type="date" name="dob" class="form-control" required></td>**

**</tr>**

**<tr>**

**<td colspan="2">**

**<input name="submit" class="btn btn-primary" type="submit" value="submit"/></td>**

**</tr>**

**</table>**

**</form>**

**if (isset($\_REQUEST['submit'])) {**

**$fname = $\_REQUEST['fname'];**

**$lname = $\_REQUEST['lname'];**

**$email = $\_REQUEST['email'];**

**$pwd = $\_REQUEST['pwd'];**

**$dob = $\_REQUEST['dob'];**

**$user = new User($fname, $lname, $email, $pwd, $dob);**

**$user->create(); // User account created...**

**echo "<br>";**

**echo $user->saved(); // Method not found**

**echo "<br>";**

**$user->display(); // Array ( [fname] => john [lname] => doe [email] => john@doe.com [dob] => 2020-12-25 )**

**}**

**Scope Resolution**

**Q. Explain the scope resolution operator (::) and its use in PHP.**

* The scope resolution operator (::), is used in PHP to access static, constant, and class-level properties and methods.
* It allows you to reference class members (properties and methods) without needing to instantiate an object of the class.

1**. Accessing Static Methods and Properties**

* Static properties and methods belong to the class itself rather than to any specific instance of the class.
* The :: operator is used to access these static members.
* Static Method Example:

**class MyClass {**

**public static function sayHello() {**

**echo "Hello, World!";**

**}**

**}**

**// Accessing a static method using the scope resolution operator**

**MyClass::sayHello(); // Output: Hello, World!**

Static Property Example:

**class MyClass {**

**public static $greeting = "Hello";**

**public static function greet() {**

**echo self::$greeting . ", World!";**

**}**

**}**

**// Accessing a static property using the scope resolution operator**

**echo MyClass::$greeting; // Output: Hello**

**2. Accessing Constants**

* You can use the :: operator to access class constants.
* Constants are defined using the const keyword and are not meant to be changed once set.
* **Class Constant Example:**

**class MyClass {**

**const GREETING = "Hello, World!";**

**}**

**// Accessing a class constant**

**echo MyClass::GREETING; // Output: Hello, World!**

**3. Accessing Parent Class Methods and Properties**

* + The :: operator is also used in inheritance to call methods or access properties of a parent class.
  + You can use parent:: to access the parent class's members.

**Parent Class Access Example**:

**Source code:**

**class ParentClass {**

**protected static $name = "Parent Class";**

**public static function display() {**

**echo "This is the " . self::$name . ".";**

**}**

**}**

**class ChildClass extends ParentClass {**

**public static function show() {**

**echo "This is the " . parent::$name . ".";**

**}**

**}**

**// Accessing parent class static property using the scope resolution operator**

**ChildClass::show(); // Output: This is the Parent Class.**

**4. Accessing Methods in Interfaces and Traits**

* The :: operator is used to refer to methods from interfaces or traits.
* You can call static methods from interfaces directly, as well as call methods from traits when used in a class.
* **Calling Interface Method:**
* **Source code:**

**interface MyInterface {**

**public static function sayHello();**

**}**

**class MyClass implements MyInterface {**

**public static function sayHello() {**

**echo "Hello from Interface!";**

**}**

**}**

**MyClass::sayHello(); // Output: Hello from Interface!**

**Calling Trait Method:**

**Source code:**

**trait MyTrait {**

**public static function sayHello() {**

**echo "Hello from Trait!";**

**}**

**}**

**class MyClass {**

**use MyTrait;**

**}**

**MyClass::sayHello(); // Output: Hello from Trait!**

**5. Accessing the Current Class with self::**

* You can use self:: within a class to refer to static properties and methods of that class, including those that are inherited from parent classes.
* **Example with self::**

**class MyClass {**

**public static $name = "MyClass";**

**public static function display() {**

**echo "Class Name: " . self::$name;**

**}**

**}**

**My Class::display(); // Output: Class Name: My Class**

**Practical Exercise: Create a class with static properties and methods, and demonstrate their access using the scope resolution operator.**

**Source code:**

**class MyClass**

**{**

**public static $greeting = "Hello";**

**public static function greet()**

**{**

**echo self::$greeting . ", World!";**

**}**

**}**

**echo MyClass::$greeting; // Hello**

**MyClass::greet(); // Hello, World!**

**Traits:**

**Q. Define traits in PHP and their purpose in code reuse.**

**Traits in PHP: Purpose and Use in Code Reuse**

* Traits in PHP are a mechanism for code reuse in single inheritance languages, allowing developers to share methods across multiple classes without the need for inheritance.
* They help avoid the problem of code duplication by allowing you to define reusable sets of methods that can be included in one or more classes.
* While PHP supports single inheritance (a class can only inherit from one parent class), traits provide a way to reuse functionality across multiple classes without creating a complex inheritance hierarchy.

**Key Characteristics of Traits:**

* **Code Reuse:** Traits allow for the reuse of methods across different classes.
* **No Inheritance:** Unlike classes, traits cannot be instantiated. They are used to extend functionality, not to create objects.
* **Combining Multiple Traits**: A single class can use multiple traits, allowing it to combine various pieces of functionality.
* **No Constructor or Destructor:** Traits cannot have constructors or destructors, as they are meant to provide reusable functionality, not to manage object lifecycle.

**Defining a Trait**

* A trait is defined using the trait keyword followed by the trait's name.
* Inside the trait, you can define methods, but you cannot define properties or constants (although properties can be declared inside a class using use in combination with traits).

**Example:**

**trait Logger {**

**public function log($message) {**

**echo "Log: $message\n";**

**}**

**}**

**class MyClass {**

**use Logger; // Including the Logger trait**

**public function performAction() {**

**echo "Performing action...\n";**

**$this->log("Action performed"); // Accessing method from the trait**

**}**

**}**

**$obj = new MyClass();**

**$obj->performAction(); // Output: Performing action... Log: Action performed**

**In the example above:**

* The Logger trait defines a log() method.
* The MyClass class uses the Logger trait, allowing it to call log() without defining it in the class.

**Using Multiple Traits**

* A class can use multiple traits by separating them with commas in the use statement.
* This allows a class to combine functionality from several traits.

**Source code:**

**trait A {**

**public function methodA() {**

**echo "Method from trait A\n";**

**}**

**}**

**trait B {**

**public function methodB() {**

**echo "Method from trait B\n";**

**}**

**}**

**class MyClass {**

**use A, B; // Using both traits**

**public function callMethods() {**

**$this->methodA();**

**$this->methodB();**

**}**

**}**

**$obj = new MyClass();**

**$obj->callMethods();**

**// Output:**

**// Method from trait A**

**// Method from trait B**

* MyClass combines methods from both trait A and trait B by using them together.
* The methods methodA() and methodB() can be called on instances of MyClass**.**

**Purpose of Traits: Code Reuse**

1. **Avoiding Code Duplication**
   * One of the main purposes of traits is to avoid code duplication.
   * Rather than repeating the same methods in several classes, you can define them in a trait and simply include that trait in all relevant classes.

**Example:**

**trait DatabaseConnection {**

**public function connect() {**

**echo "Connecting to the database...\n";**

**}**

**}**

**class User {**

**use DatabaseConnection;**

**public function getUserData() {**

**echo "Fetching user data...\n";**

**}**

**}**

**class Admin {**

**use DatabaseConnection;**

**public function getAdminData() {**

**echo "Fetching admin data...\n";**

**}**

**}**

**$user = new User();**

**$user->connect(); // Reused from trait**

**$admin = new Admin();**

**$admin->connect(); // Reused from trait**

* + In this example, both User and Admin classes have a connect() method without having to duplicate the method code.

1. **Combining Functionality**
   * Traits allow you to combine functionality across classes that may not be related by inheritance.
   * You can add specific methods to different classes that need them, even if the classes do not share a common parent.
   * For example, a class can implement logging functionality using a Logger trait, and another unrelated class can also implement caching functionality using a Cache trait, all without forcing them to inherit from a common class.
2. **Separation of Concerns**
   * Traits allow you to separate distinct pieces of functionality into isolated units. For instance:
   * One trait might handle logging, another might handle validation, and another could deal with database access.
   * You can then include these traits in the classes that need that specific functionality.

**Example:**

**trait Validation {**

**public function validate($data) {**

**echo "Validating data...\n";**

**}**

**}**

**trait Caching {**

**public function cache($data) {**

**echo "Caching data...\n";**

**}**

**}**

**class User {**

**use Validation, Caching; // User class needs both validation and caching functionality**

**public function saveUser($data) {**

**$this->validate($data);**

**$this->cache($data);**

**echo "User data saved\n";**

**}**

**}**

**$user = new User();**

**$user->saveUser("Some user data");**

**// Output:**

**// Validating data...**

**// Caching data...**

**// User data saved**

**4.Avoiding Multiple Inheritance Issues**

* PHP does not support multiple inheritance (a class cannot extend more than one class).
* However, traits allow you to mimic multiple inheritance in a controlled manner by letting a class use multiple traits.
* This avoids the complexity and issues that arise with multiple inheritance while still allowing for flexible code reuse.

**Example:**

**trait Validation {**

**public function validate() {**

**echo "Validation\n";**

**}**

**}**

**trait Logging {**

**public function log() {**

**echo "Logging\n";**

**}**

**}**

**class User {**

**use Validation, Logging; // Multiple traits used in a single class**

**}**

**$user = new User ();**

**$user->validate (); // Output: Validation**

**$user->log (); // Output: Logging**

**Important Points About Traits:**

1. **No Instantiation:** Traits cannot be instantiated. They are included in classes using the use keyword.
2. **Conflict Resolution:** If two traits define a method with the same name, a conflict arises. PHP provides a way to resolve conflicts by using insteadof and as operators.

**Conflict Resolution Example**

**trait TraitA {**

**public function greet() {**

**echo "Hello from Trait A\n";**

**}**

**}**

**trait TraitB {**

**public function greet() {**

**echo "Hello from Trait B\n";**

**}**

**}**

**class MyClass {**

**use TraitA, TraitB {**

**TraitB::greet insteadof TraitA; // Resolving conflict**

**TraitA::greet as greetFromA; // Aliasing method**

**}**

**}**

**$obj = new MyClass();**

**$obj->greet(); // Output: Hello from Trait B**

**$obj->greetFromA(); // Output: Hello from Trait A**

**3.No Constructors**: Traits cannot have constructors or destructors, so they can't be used to manage the initialization or cleanup of objects. However, methods inside the trait can be used for such purposes.

**Practical Exercise: Create two traits and use them in a class to demonstrate how to include multiple behaviors.**

**trait TraitA**

**{**

**public function greetA()**

**{**

**echo "Hello from Trait A\n";**

**}**

**}**

**trait TraitB**

**{**

**public function greetB()**

**{**

**echo "Hello from Trait B\n";**

**}**

**}**

**class MyClass**

**{**

**use TraitA, TraitB;**

**}**

**$obj1 = new MyClass();**

**echo $obj1->greetA(); // Hello from Trait A**

**echo $obj1->greetB(); // Hello from Trait B**

**Visibility**

**Q. Discuss the visibility of properties and methods in PHP (public, private, protected).**

In PHP, visibility determines how properties and methods of a class can be accessed. There are three levels of visibility:

**1. Public**

* The property or method is accessible from anywhere, including outside the class.
* This is the default visibility if none is specified.
* **Example:**

**class Example {**

**public $name = "John";**

**public function greet() {**

**return "Hello, " . $this->name;**

**}**

**}**

**$obj = new Example();**

**echo $obj->name; // Accessible**

**echo $obj->greet(); // Accessible**

**2. Private**

* The property or method is accessible only within the class where it is declared.
* It cannot be accessed from outside the class, including child classes.
* **Example:**

**class Example {**

**private $secret = "Hidden";**

**private function getSecret() {**

**return $this->secret;**

**}**

**public function revealSecret() {**

**return $this->getSecret();**

**}**

**}**

**$obj = new Example();**

**// echo $obj->secret; // ❌ Fatal error**

**// echo $obj->getSecret(); // ❌ Fatal error**

**echo $obj->revealSecret(); // ✅ Allowed via a public method**

**3. Protected**

* The property or method is accessible within the class and by derived (child) classes, but not from outside the class.
* Example:

**class ParentClass {**

**protected $message = "Protected Property";**

**protected function showMessage() {**

**return $this->message;**

**}**

**}**

**class ChildClass extends ParentClass {**

**public function getMessage() {**

**return $this->showMessage(); // Allowed in subclass**

**}**

**}**

**$obj = new ChildClass();**

**// echo $obj->message; // ❌ Fatal error**

**// echo $obj->showMessage(); // ❌ Fatal error**

**echo $obj->getMessage(); // ✅ Allowed via public method in subclass**

**Summary Table**

| **Visibility** | **Same Class** | **Derived Class** | **Outside Class** |
| --- | --- | --- | --- |
| **Public** | **✅ Yes** | **✅ Yes** | **✅ Yes** |
| **Private** | **✅ Yes** | **❌ No** | **❌ No** |
| **Protected** | **✅ Yes** | **✅ Yes** | **❌ No** |

**When to Use Which Visibility?**

* **Public →** When you need unrestricted access (e.g., utility methods, getters).
* **Private →** When data should be completely encapsulated (e.g., sensitive information, internal logic).
* **Protected →** When you want to allow access within the class and subclasses, but prevent direct access from outside.

### **Practical Exercise: Write a class that shows examples of each visibility type and how they restrict access to properties and methods.**

**Example**

**Source code:**

**class MyClass**

**{**

**public $propPublic = "This is a public property";**

**protected $propProtected = "This is a protected property";**

**private $propPrivate = "This is a private property";**

**public function metPublic()**

**{**

**return "This is a public method";**

**}**

**protected function metProtected()**

**{**

**return "This is a protected method";**

**}**

**private function metPrivate()**

**{**

**return "This is a private method";**

**}**

**public function getPropProtected()**

**{**

**return $this->propProtected;**

**}**

**public function getPropPrivate()**

**{**

**return $this->propPrivate;**

**}**

**public function runMetProtected()**

**{**

**return $this->metProtected();**

**}**

**public function runMetPrivate()**

**{**

**return $this->metPrivate();**

**}**

**}**

**class MyChildClass extends MyClass {}**

**$obj = new MyClass();**

**$obj2 = new MyChildClass();**

**echo $obj->propPublic; // This is a public property**

**echo $obj->propProtected; // Fatal Error**

**echo $obj->getPropProtected(); // This is a protected property**

**echo $obj->propPrivate; // Fatal Error**

**echo $obj->getPropPrivate(); // This is a private property**

**echo $obj->metPublic(); // This is a public method**

**echo $obj->metProtected(); // Fatal Error**

**echo $obj->runMetProtected(); // This is a protected method**

**echo $obj->metPrivate(); // Fatal Error**

**echo $obj->runMetPrivate(); // This is a private method**

**echo $obj2->propPublic; // This is a public property**

**echo $obj2->propProtected; // Fatal Error**

**echo $obj2->getPropProtected(); // This is a protected property**

**echo $obj2->propPrivate; // Fatal Error**

**echo $obj2->getPropPrivate(); // This is a private property**

**echo $obj2->metPublic(); // This is a public method**

**echo $obj2->metProtected(); // Fatal Error**

**echo $obj2->runMetProtected(); // This is a protected method**

**echo $obj2->metPrivate(); // Fatal Error**

**echo $obj2->runMetPrivate(); // This is a private method**

**Type Hinting**

**THEORY EXERCISE: Explain type hinting in PHP and its benefits.**

**Type Hinting in PHP**

* Type hinting (also called type declarations) in PHP is a feature that allows specifying the expected data type of function parameters, return values, and class properties.
* It ensures that the provided arguments match the expected types, leading to more robust and error-free code.

**1. Type Hinting for Function Parameters**

* You can specify the expected type for function parameters.
* Example: Type Hinting for Basic Data Types

**Example:**

**function addNumbers(int $a, int $b) {**

**return $a + $b;**

**}**

**echo addNumbers(5, 10); // 15**

**// echo addNumbers(5, "10"); // TypeError in strict mode**

**2. Type Hinting for Return Types**

* **Yo**u can also specify the return type of a function.
* Example: Function Return Type Declaration

**Example:**

**function getGreeting(string $name): string {**

**return "Hello, " . $name;**

**}**

**echo getGreeting("Ramesh"); // Hello, Ramesh**

**3. Type Hinting for Class Properties (PHP 7.4+)**

* PHP 7.4 introduced typed properties, allowing type hints for class attributes.
* Example: Typed Class Properties

**class User {**

**public int $id;**

**public string $name;**

**public function \_\_construct(int $id, string $name) {**

**$this->id = $id;**

**$this->name = $name;**

**}**

**}**

**$user = new User(1, "Ramesh");**

**echo $user->name; // Ramesh**

**4. Type Hinting for Objects and Arrays**

* You can enforce that parameters must be specific objects or arrays.
* Example: Object Type Hinting

**class Order {**

**public function process(User $user) {**

**echo "Processing order for " . $user->name;**

**}**

**}**

**$order = new Order();**

**$user = new User(2, "Ramesh");**

**$order->process($user); // Processing order for Ramesh**

Example: Array Type Hinting

**function printNames(array $names) {**

**foreach ($names as $name) {**

**echo $name . "\n";**

**}**

**}**

**printNames(["Nirav", "More"]); // Works fine**

**5. Nullable Types (PHP 7.1+)**

* You can allow null values by using a question mark ? before the type.
* Example: Nullable Type Hinting

**Example:**

**function setAge(?int $age) {**

**return $age ?? "Age not provided";**

**}**

**echo setAge(25); // 25**

**echo setAge(null); // Age not provided**

**6. Union Types (PHP 8.0+)**

* PHP 8.0 introduced union types, allowing multiple possible types.
* Example: Union Type Hinting

**function getData(int|string $value): string {**

**return "Value: " . $value;**

**}**

**echo getData(10); // Value: 10**

**echo getData("Hello"); // Value: Hello**

**7. Mixed Type (PHP 8.0+)**

* mixed is a special type that accepts any value.
* Example: Using mixed

**function processData(mixed $data) {**

**if (is\_array($data)) {**

**return "Array provided";**

**}**

**return "Data: " . $data;**

**}**

**echo processData(["PHP", "Type Hinting"]); // Array provided**

**echo processData(100); // Data: 100**

**Benefits of Type Hinting**

* **Error Prevention:**Prevents unintended data type mismatches.
* **Code Readability:**Makes it clear what data type is expected.
* **Better Debugging:**Catches type errors early.
* **Performance Optimization**: Helps PHP optimize function calls.
* **Easier Refactoring**: Reduces the risk of breaking changes.

### **Practical Exercise: Write a method in a class that accepts type-hinted parameters and demonstrate how it works with different data types.**

**class User**

**{**

**public int $id;**

**public string $name;**

**public string $email;**

**public int $phone;**

**public float $weight;**

**public float $height;**

**public function \_\_construct(int $id, string $name, string $email, int $phone, float $weight, float $height)**

**{**

**$this->id = $id;**

**$this->name = $name;**

**$this->email = $email;**

**$this->phone = $phone;**

**$this->weight = $weight;**

**$this->height = $height;**

**}**

**}**

**$user = new User(1, "Tom", "tom@email.com", 9998887777, 65.84, 143.29);**

**echo $user->id; // 1**

**echo $user->name; // Tom**

**echo $user->email; // tom@email.com**

**echo $user->phone; // 9998887777**

**echo $user->weight; // 65.84**

**echo $user->height; // 143.29**

**Final Keyword**

**THEORY EXERCISE: Discuss the purpose of the final keyword in PHP and how it affects classes and methods.**

* **I**n PHP, the final keyword is used to prevent further modification of classes and methods.
* It ensures that certain parts of the code remain unchanged by subclasses, providing a way to enforce immutability and maintain code integrity.

**Usage of final in PHP**

**1. final with Methods**

* When a method is declared as final, it cannot be overridden by any subclass.
* This is useful when you want to enforce specific behavior in a parent class that must remain unchanged.
* Example:

**final class Utility {**

**public function helperFunction() {**

**echo "This class cannot be extended.";**

**}**

**}**

**// This will cause a Fatal Error**

**// class ExtendedUtility extends Utility {**

**// }**

* **Since Utility is marked as final, attempting to create a subclass (ExtendedUtility) results in a Fatal Error.**

**Why Use final?**

* **Prevents Inheritance Issues:** Stops unintended modifications in child classes that could break functionality.
* **Enhances Security:** Ensures that critical methods remain unchanged.
* **Improves Performance:** Helps PHP's optimizer by reducingthe complexity of method resolution.

However, overusing final can reduce flexibility, making code less extendable. It is best used when you want to enforce strict behavior in core logic.

**Practical Exercise: Create a class marked as final and attempt to extend it to show the restriction.**

**final class RestClass**

**{**

**function message()**

**{**

**echo "This is a method from final class";**

**}**

**}**

**class DerivedClass extends RestClass**

**{**

**function message()**

**{**

**echo "changes with child class of a final class";**

**}**

**}**

**$obj = new RestClass();**

**$obj1 = new DerivedClass();**

**echo $obj->message(); // This is a method from final class**

**echo $obj1->message(); // Fatal error: Class DerivedClass cannot extend final class RestClass**

**Email Security Function**

**THEORY EXERCISE: Explain the importance of email security and common practices to ensure secure email transmission.**

**Importance of Email Security**

* Email security is crucial because emails are a primary communication method for individuals and businesses, often containing sensitive data, financial information, or authentication credentials.
* **Poor email security can lead to:**
  + **Phishing Attacks –** Hackers use fake emails to steal sensitive information**.**
  + **Data Breaches –** Unencrypted emails can be intercepted, exposing confidential data.
  + **Malware & Ransomware –** Malicious attachments or links can infect systems.
  + **Spoofing & Identity Theft –** Attackers can forge email addresses to impersonate legitimate senders.

**Common Practices for Secure Email Transmission**

**1. Use Strong Authentication Methods**

* Implement Multi-Factor Authentication (MFA) to add an extra security layer beyond passwords.
* Use strong, unique passwords and update them regularly.

**2. Encrypt Emails for Confidentiality**

* **TLS Encryption (Transport Layer Security):** Ensures emails are encrypted during transmission.
* **End-to-End Encryption (PGP or S/MIME):** Protects emails from being read by unauthorized parties.

**3. Enable Email Security Protocols**

* **SPF (Sender Policy Framework):** Prevents spoofing by verifying sender IP addresses.
* **DKIM (DomainKeys Identified Mail):** Ensures email integrity by adding a digital signature.
* **DMARC (Domain-based Message Authentication, Reporting & Conformance):** Protects against phishing by enforcing SPF and DKIM policies.

**4. Beware of Phishing and Social Engineering**

* Do not click on suspicious links or attachments from unknown senders.
* Verify email sender authenticity, especially for financial transactions.

**5. Secure Your Email Server**

* Keep mail server software updated to patch security vulnerabilities.
* Use firewalls and anti-malware tools to protect email infrastructure.
* Implement rate-limiting and filtering to prevent spam and DDoS attacks.

**6. Use a Secure Email Provider**

* Choose an email provider that supports strong encryption and advanced security features like Google Workspace or ProtonMail.

**7. Regularly Monitor and Train Employees**

* Conduct security awareness training to help users identify threats.
* Implement email logging and monitoring for suspicious activities.

By following these best practices, organizations and individuals can significantly reduce the risks associated with email-based threats.

**Practical Exercise: Write a function that sanitizes email input and validates it before sending.**

**Example:**

**Source code:**

**function sendSafeEmail($email, $subject, $message)**

**{**

**// Step 1: Sanitize email input**

**$email = filter\_var(trim($email), FILTER\_SANITIZE\_EMAIL);**

**// Step 2: Validate email format**

**if (!filter\_var($email, FILTER\_VALIDATE\_EMAIL)) {**

**return "Invalid email format.";**

**}**

**// Step 3: Prevent Header Injection**

**if (preg\_match("/(\r|\n)/", $email) || preg\_match("/(\r|\n)/", $subject)) {**

**return "Email headers contain invalid characters.";**

**}**

**// Step 4: Prepare email headers**

**$headers = "From: no-reply@example.com\r\n";**

**$headers .= "Reply-To: no-reply@example.com\r\n";**

**$headers .= "MIME-Version: 1.0\r\n";**

**$headers .= "Content-Type: text/plain; charset=UTF-8\r\n";**

**// Step 5: Send email securely**

**if (mail($email, $subject, $message, $headers)) {**

**return "Email sent successfully to $email.";**

**} else {**

**return "Failed to send email.";**

**}**

**}**

**// Example usage**

**echo sendSafeEmail("test@example.com", "Welcome!", "Hello, welcome to our platform!");**

**// To: test@example.com**

**// Subject: Welcome!**

**// From: no-reply@example.com**

**// Reply-To: no-reply@example.com**

**// MIME-Version: 1.0**

**// Content-Type: text/plain; charset=UTF-8**

**// Hello, welcome to our platform!**

**File Handling**

**THEORY EXERCISE: Discuss file handling in PHP, including opening, reading, writing, and closing files.**

**File Handling in PHP**

* File handling in PHP allows you to create, read, write, and manage files on the server.
* PHP provides built-in functions to perform these operations efficiently.

**1. Opening a File (fopen())**

* To work with a file, you first need to open it using fopen().
* **This function requires two parameters:**
  + **Filename:** The file to be opened.
  + **Mode:** The file operation mode (read, write, append, etc.).

**File Opening Modes:**

| Mode | Description |
| --- | --- |
| "r" | Read-only. File must exist. |
| "r+" | Read & write. File must exist. |
| "w" | Write-only. Creates a new file or truncates an existing one. |
| "w+" | Read & write. Creates a new file or truncates an existing one. |
| "a" | Append. Creates a file if it doesn’t exist. |
| "a+" | Read & append. Creates a file if it doesn’t exist. |
| "x" | Write-only. Fails if the file exists. |
| "x+" | Read & write. Fails if the file exists. |

Example: Opening a File

**<?php**

**$file = fopen("example.txt", "r") or die("Unable to open file!");**

**?>**

**2. Reading a File**

* PHP provides several functions to read file content:
* fread() – Read a Specific Number of Bytes

**<?php**

**$file = fopen("example.txt", "r");**

**$content = fread($file, filesize("example.txt"));**

**echo $content;**

**fclose($file);**

**?>**

**file\_get\_contents() – Read the Whole File into a String**

**<?php**

**$content = file\_get\_contents("example.txt");**

**echo $content;**

**?>**

**3. Writing to a File**

To write to a file, use fwrite() or file\_put\_contents().

fwrite() – Write Data to a File

<?php

$file = fopen("example.txt", "w");

fwrite($file, "Hello, World!\n");

fclose($file);

?>

**"w" mode will erase existing content. Use "a" mode to append.**

#### file\_put\_contents()**– Write Directly**

**<?php**

**file\_put\_contents("example.txt", "New content here!");**

**?>**

**4. Closing a File (fclose())**

Always close a file after reading or writing to free system resources.

<?php

$file = fopen("example.txt", "r");

fclose($file);

?>

**5. Checking if a File Exists (file\_exists())**

<?php

if (file\_exists("example.txt")) {

echo "File exists.";

} else {

echo "File does not exist.";

}

?>

**6. Deleting a File (unlink())**

<?php

if (file\_exists("example.txt")) {

unlink("example.txt");

echo "File deleted.";

} else {

echo "File does not exist.";

}

?>

* PHP provides powerful file-handling functions to open, read, write, and close files efficiently.
* Always handle files carefully, especially when writing or deleting, to avoid data loss.

### **Practical Exercise: Create a script that reads from a text file and displays its content on a web page.**

// Define the file path

$file = 'index.txt';

// Check if the file exists

if (file\_exists($file)) {

// Read the file content

$content = file\_get\_contents($file);

} else {

$content = "File not found.";

}

?>

echo htmlspecialchars($content); // This is text from index.txt file.

**Handling Emails**

**THEORY EXERCISE: Explain how to send emails in PHP using the mail() function and the importance of validating email addresses.**

**Sending Emails in PHP Using the mail() Function**

* PHP provides the mail() function for sending emails, but it requires a properly configured mail server (such as **Sendmail**, **Postfix**, or **SMTP**) to work correctly.

**1. Basic Syntax of mail()**

mail(to, subject, message, headers, parameters);

* **to** – Recipient's email address
* **subject** – Subject of the email
* **message** – Email body
* **headers** – Additional email headers (e.g., From, Reply-To)
* **parameters** – Optional, used to specify additional options

**2. Example: Sending a Basic Email**

<?php

$to = "recipient@example.com";

$subject = "Test Email";

$message = "Hello, this is a test email from PHP.";

$headers = "From: sender@example.com";

if (mail($to, $subject, $message, $headers)) {

echo "Email sent successfully!";

} else {

echo "Failed to send email.";

}

?>

* The mail() function does not return an error message if the email fails to send.
* It requires a working mail server.

**3. Using Additional Headers**

* You can add extra headers, such as Reply-To, CC, BCC, and Content-Type:

<?php

$to = "recipient@example.com";

$subject = "HTML Email Test";

$message = "<h1>Hello!</h1><p>This is an <b>HTML</b> email.</p>";

$headers = "MIME-Version: 1.0" . "\r\n";

$headers .= "Content-type: text/html; charset=UTF-8" . "\r\n";

$headers .= "From: sender@example.com" . "\r\n";

$headers .= "Reply-To: reply@example.com" . "\r\n";

$headers .= "CC: cc@example.com" . "\r\n";

$headers .= "BCC: bcc@example.com" . "\r\n";

if (mail($to, $subject, $message, $headers)) {

echo "HTML Email sent successfully!";

} else {

echo "Failed to send email.";

}

?>

**4. Validating Email Addresses**

* To prevent email spoofing and errors, always validate email addresses before sending.

**Using filter\_var() to Validate an Email**

<?php

$email = "user@example.com";

if (filter\_var($email, FILTER\_VALIDATE\_EMAIL)) {

echo "Valid email address.";

} else {

echo "Invalid email address.";

}

?>

**Why is Email Validation Important?**

* Prevents sending emails to invalid addresses.
* Reduces the chance of being marked as spam.
* Improves reliability and security.

**5. Using PHPMailer for Advanced Emailing (Recommended)**

* The mail() function has limitations, especially for sending emails via SMTP. - A better alternative is **PHPMailer**, which supports SMTP authentication and attachments.

**Install PHPMailer via Composer**

**Send Email Using PHPMailer**

<?php

use PHPMailer\PHPMailer\PHPMailer;

use PHPMailer\PHPMailer\Exception;

require 'vendor/autoload.php';

$mail = new PHPMailer(true);

try {

$mail->isSMTP();

$mail->Host = 'smtp.example.com';

$mail->SMTPAuth = true;

$mail->Username = 'your\_email@example.com';

$mail->Password = 'your\_password';

$mail->SMTPSecure = PHPMailer::ENCRYPTION\_STARTTLS;

$mail->Port = 587;

$mail->setFrom('your\_email@example.com', 'Your Name');

$mail->addAddress('recipient@example.com');

$mail->isHTML(true);

$mail->Subject = 'Test Email with PHPMailer';

$mail->Body = '<h1>Hello!</h1><p>This is a test email.</p>';

$mail->send();

echo "Email sent successfully!";

} catch (Exception $e) {

echo "Email could not be sent. Error: {$mail->ErrorInfo}";

}

?>

**Why Use PHPMailer?**

* Supports SMTP authentication
* Allows sending attachments
* Better error handling
* More secure than mail()

**Conclusion**

* The mail() function is useful for simple emails but requires a mail server.
* Always **validate email addresses** before sending.
* Use **PHPMailer** for advanced email handling, including SMTP authentication and HTML emails.

### **Practical Exercise: Write a PHP script to send a test email to a user using the mail() function.**

$to = "recipient@example.com";

$subject = "Test Email from PHP";

$message = "This is a test email sent using PHP's mail() function.";

$headers = "From: sender@example.com" . "\r\n" .

"Reply-To: sender@example.com" . "\r\n" .

"X-Mailer: PHP/" . phpversion();

if(mail($to, $subject, $message, $headers)) {

echo "Email sent successfully to $to";

} else {

echo "Failed to send email.";

}

**MVC Architecture**

**THEORY EXERCISE: Discuss the Model-View-Controller (MVC) architecture and its advantages in web development.**

* **The Model-View-Controller (MVC)**architecture is a design pattern used in web development to separate application logic into three interconnected components**:**
  1. Model – Handles data and business logic.
  2. View – Manages the user interface (UI).
  3. Controller – Handles user input and interacts with the Model and View.
* This separation makes code modular, maintainable, and scalable**.**

**Model (Data & Business Logic)**

* Represents the application's data structure.
* Interacts with the database (CRUD operations: Create, Read, Update, Delete).
* Implements business logic and validation.
* Example (Model in PHP)

**class UserModel {**

**private $db;**

**public function \_\_construct($dbConnection) {**

**$this->db = $dbConnection;**

**}**

**public function getUser($id) {**

**$stmt = $this->db->prepare("SELECT \* FROM users WHERE id = ?");**

**$stmt->execute([$id]);**

**return $stmt->fetch();**

**}**

**}**

**View (User Interface)**

* Responsible for displaying data to the user.
* Renders HTML, CSS, and JavaScript.
* Does not contain business logic.
* **Example** (View in PHP)

**<!-- user\_view.php -->**

**<h1>User Profile</h1>**

**<p>Name: <?php echo $user['name']; ?></p>**

**<p>Email: <?php echo $user['email']; ?></p>**

**Controller (Request Handling)**

* Accepts user input (e.g., form submissions, URL requests).
* Calls the appropriate Model to fetch or modify data.
* Loads the View to display the data.
* Example (Controller in PHP)

// **user\_controller.php**

**require 'UserModel.php';**

**require 'db.php'; // Database connection**

**$userModel = new UserModel($db);**

**$user = $userModel->getUser($\_GET['id']);**

**require 'user\_view.php'; // Load the view**

**How MVC Works in Web Applications**

1. User requests a URL → example.com/user/1
2. Controller processes the request and interacts with the Model.
3. Model fetches data from the database.
4. Controller sends data to the View.
5. View renders the response (HTML page).

**Advantages of MVC in Web Development**

* Separation of Concerns (SoC) – Each component handles a specific role, improving maintainability.
* Code Reusability – Models and Views can be reused in multiple parts of the application.
* Scalability – Easier to scale applications by adding new models, views, and controllers.
* Easier Collaboration – Developers can work on different parts of the application simultaneously (e.g., frontend & backend teams).
* Security – Business logic is separated from the view, reducing direct access to data.
* Many PHP frameworks use MVC, such as: Laravel, CodeIgniter, Symfony, CakePHP

**Conclusion**

* MVC is a powerful architecture that enhances code organization, maintainability, and scalability in web development.
* If you're building large applications, using an MVC framework like Laravel can significantly improve development.

**Practical Exercise: Create a simple MVC application that demonstrates the separation of concerns by implementing a basic "User" module with a model, view, and controller.**

**Source code:**

**├── mvc\_app/**

**│ ├── index.php**

**│ ├── controllers/**

**│ │ ├── UserController.php**

**│ ├── models/**

**│ │ ├── User.php**

**│ ├── views/**

**│ │ ├── user\_view.php**

**// index.php**

**<?php**

**require\_once 'controllers/UserController.php';**

**$controller = new UserController();**

**$controller->index();**

**?>**

**// models/User.php**

**<?php**

**class User {**

**public function getUsers() {**

**return [**

**["id" => 1, "name" => "John Doe", "email" => "john@example.com"],**

**["id" => 2, "name" => "Jane Doe", "email" => "jane@example.com"]**

**];**

**}**

**}**

**?>**

**// controllers/UserController.php**

**<?php**

**require\_once 'models/User.php';**

**class UserController {**

**public function index() {**

**$userModel = new User();**

**$users = $userModel->getUsers();**

**require 'views/user\_view.php';**

**}**

**}**

**?>**

**// views/user\_view.php**

**<!DOCTYPE html>**

**<html>**

**<head>**

**<title>User List</title>**

**</head>**

**<body>**

**<h1>Users</h1>**

**<ul>**

**<?php foreach ($users as $user): ?>**

**<li><?php echo $user['name'] . " (" . $user['email'] . ")"; ?></li>**

**<?php endforeach; ?>**

**</ul>**

**</body>**

**</html>**

**Practical Example: Implementation of all the OOPs Concepts**

**Practical Exercise: Develop a mini project (e.g., a Library Management System) that utilizes all OOP concepts like classes, inheritance, interfaces, magic methods, etc.**

**Source code:**

**├── library\_system/**

**│ ├── index.php**

**│ ├── classes/**

**│ │ ├── Book.php**

**│ │ ├── Member.php**

**│ │ ├── Library.php**

**│ │ ├── LibraryItem.php**

**│ │ ├── Borrowable.php**

**// classes/LibraryItem.php**

**abstract class LibraryItem {**

**protected $title;**

**protected $author;**

**public function \_\_construct($title, $author) {**

**$this->title = $title;**

**$this->author = $author;**

**}**

**abstract public function getDescription();**

**}**

**// classes/Borrowable.php**

**interface Borrowable {**

**public function borrowItem($memberName);**

**public function returnItem();**

**}**

**// classes/Book.php**

**require\_once 'LibraryItem.php';**

**require\_once 'Borrowable.php';**

**class Book extends LibraryItem implements Borrowable {**

**private $isBorrowed = false;**

**public function getDescription() {**

**return "Book: {$this->title} by {$this->author}";**

**}**

**public function borrowItem($memberName) {**

**if (!$this->isBorrowed) {**

**$this->isBorrowed = true;**

**echo "$memberName borrowed '{$this->title}'.\n";**

**} else {**

**echo "This book is already borrowed.\n";**

**}**

**}**

**public function returnItem() {**

**$this->isBorrowed = false;**

**echo "Book '{$this->title}' returned.\n";**

**}**

**}**

**// classes/Member.php**

**class Member {**

**private $name;**

**public function \_\_construct($name) {**

**$this->name = $name;**

**}**

**public function getName() {**

**return $this->name;**

**}**

**}**

**// classes/Library.php**

**class Library {**

**private $items = [];**

**public function addItem(LibraryItem $item) {**

**$this->items[] = $item;**

**}**

**public function listItems() {**

**foreach ($this->items as $item) {**

**echo $item->getDescription() . "\n";**

**}**

**}**

**}**

**// index.php**

**require\_once 'classes/Book.php';**

**require\_once 'classes/Member.php';**

**require\_once 'classes/Library.php';**

**$library = new Library();**

**$book1 = new Book("1984", "George Orwell");**

**$book2 = new Book("To Kill a Mockingbird", "Harper Lee");**

**$member = new Member("Alice");**

**$library->addItem($book1);**

**$library->addItem($book2);**

**$library->listItems();**

**$book1->borrowItem($member->getName());**

**$book1->returnItem();**

**Connection with MySQL Database**

**THEORY EXERCISE: Explain how to connect PHP to a MySQL database using mysqli or PDO.**

**Connecting PHP to a MySQL Database Using mysqli and PDO**

* To connect PHP to a MySQL database, you can use mysqli (MySQL Improved) or PDO (PHP Data Objects).
* Both approaches allow executing SQL queries, retrieving data, and managing transactions.

**1. Connecting Using mysqli (Procedural & Object-Oriented)**

* The mysqli extension provides two ways to connect:
  + Procedural Style
  + Object-Oriented Style

1. **mysqli Procedural Connection**

**$servername = "localhost";**

**$username = "root";**

**$password = "";**

**$database = "test\_db";**

**// Create connection**

**$conn = mysqli\_connect($servername, $username, $password, $database);**

**// Check connection**

**if (!$conn) {**

**die("Connection failed: " . mysqli\_connect\_error());**

**}**

**echo "Connected successfully!";**

**✅** Pros: Simple and easy to use.  
❌ Cons: Less secure and harder to maintain in large projects.

1. **mysqli Object-Oriented Connection**

**$servername = "localhost";**

**$username = "root";**

**$password = "";**

**$database = "test\_db";**

**// Create connection**

**$conn = new mysqli($servername, $username, $password, $database);**

**// Check connection**

**if ($conn->connect\_error) {**

**die("Connection failed: " . $conn->connect\_error);**

**}**

**echo "Connected successfully!";**

✅ Pros: More structured and reusable.  
❌ Cons: Still limited compared to PDO.

**2. Connecting Using PDO (Preferred for Security & Flexibility)**

* The PDO (PHP Data Objects) extension provides a more secure, flexible, and object-oriented approach.

1. **PDO Connection**

$dsn = "mysql:host=localhost;dbname=test\_db;charset=utf8mb4";

$username = "root";

$password = "";

try {

$pdo = new PDO($dsn, $username, $password, [

PDO::ATTR\_ERRMODE => PDO::ERRMODE\_EXCEPTION, // Enable error reporting

PDO::ATTR\_DEFAULT\_FETCH\_MODE => PDO::FETCH\_ASSOC, // Fetch results as associative array

PDO::ATTR\_EMULATE\_PREPARES => false, // Prevent SQL injection

]);

echo "Connected successfully!";

} catch (PDOException $e) {

die("Connection failed: " . $e->getMessage());

}

**3. Executing Queries (mysqli vs. PDO)**

**(A) Inserting Data**

* + **mysqli Procedural**

$sql = "INSERT INTO users (name, email) VALUES ('John Doe', 'john@example.com')";

mysqli\_query($conn, $sql);

* **mysqli Object-Oriented**

$sql = "INSERT INTO users (name, email) VALUES ('John Doe', 'john@example.com')";

$conn->query($sql);

* **PDO (More Secure)**

$stmt = $pdo->prepare("INSERT INTO users (name, email) VALUES (:name, :email)");

$stmt->execute(['name' => 'John Doe', 'email' => 'john@example.com']);

**(B) Fetching Data**

* **mysqli Procedural**

$result = mysqli\_query($conn, "SELECT \* FROM users");

while ($row = mysqli\_fetch\_assoc($result)) {

echo $row['name'] . "<br>";

}

* **mysqli Object-Oriented**

$result = $conn->query("SELECT \* FROM users");

while ($row = $result->fetch\_assoc()) {

echo $row['name'] . "<br>";

}

* **PDO (Preferred)**

$stmt = $pdo->query("SELECT \* FROM users");

foreach ($stmt as $row) {

echo $row['name'] . "<br>";

}

**4. Closing the Connection**

* **mysqli**

mysqli\_close($conn); // Procedural

$conn->close(); // Object-Oriented

* **PDO**

$pdo = null; // Simply set it to null

**5. Why Choose PDO Over mysqli?**

| **Feature** | **mysqli** | **PDO** |
| --- | --- | --- |
| Database Support | Only MySQL | Supports multiple databases (MySQL, PostgreSQL, SQLite, etc.) |
| Security | Requires manual escaping | Uses prepared statements (prevents SQL injection) |
| Object-Oriented | Yes | Yes (More structured) |
| Error Handling | Basic error handling | Exception-based error handling |
| Performance | Similar for MySQL | Slightly better for complex queries |

* Use PDO for better security, flexibility, and multi-database support.\*\*
* **mysqli** is good for simple MySQL applications but lacks flexibility.
* **PDO** is the recommended choice due to **better security, multi-database support, and prepared statements**.

**Practical Exercise: Write a script to establish a database connection and handle any errors during the connection process.**

**Source code:**

**$servername = "localhost";**

**$username = "root";**

**$password = "";**

**$database = "test\_db";**

**// Create connection**

**$conn = new mysqli($servername, $username, $password, $database);**

**// Check connection**

**if ($conn->connect\_error) {**

**die("Connection failed: " . $conn->connect\_error);**

**} else {**

**echo "Connected successfully";**

**}**

**// Close connection**

**$conn->close();**

**SQL Injection**

**THEORY EXERCISE: Define SQL injection and its implications on security.**

**SQL Injection and Its Security Implications**

* SQL Injection (**SQLi**) is a **cyber attack** where an attacker manipulates SQL queries by injecting **malicious SQL code** into input fields (such as login forms, search boxes, or URL parameters).
* This allows attackers to **bypass authentication**, **access sensitive data**, **modify databases**, and even **execute system commands**.

**2. How SQL Injection Works**

* SQL Injection occurs when user inputs are **directly included in SQL queries** without proper validation or sanitization.
* Attackers exploit **vulnerable input fields** to inject malicious SQL statements.

**Example of a Vulnerable PHP Code (Using mysqli)**

**$conn = new mysqli("localhost", "root", "", "test\_db")**

**$username = $\_GET['username']; // User input from URL: ?username=admin**

**$password = $\_GET['password'];**

**$sql = "SELECT \* FROM users WHERE username = '$username' AND password = '$password'";**

**$result = $conn->query($sql);**

**if ($result->num\_rows > 0) {**

**echo "Login successful!";**

**} else {**

**echo "Invalid credentials!";**

**}**

**SQL Injection Attack**

* **An attacker can input:**

username=admin' --

password=anything

* **This modifies the SQL query as:**

**SELECT \* FROM users WHERE username = 'admin' -- ' AND password = 'anything'**

* **The -- comments out the rest of the query, allowing the attacker to log in without a valid password.**

**3. Consequences of SQL Injection**

**✅ Bypassing Authentication** – Attackers can gain unauthorized access.  
✅ **Data Theft** – Sensitive information (emails, passwords, credit card details) can be exposed.  
**✅ Data Manipulation** – Hackers can modify, delete, or insert data.  
✅ **Database Corruption** – Attackers can delete entire tables (DROP TABLE users).  
**✅ System Takeover** – In extreme cases, attackers can execute system commands, gaining full control of the server.

**4. Preventing SQL Injection**

* (A) Use Prepared Statements (Parameterized Queries)
  + Prepared statements separate SQL logic from user input, preventing malicious injection.
  + ✅ Secure PHP Code (Using mysqli Prepared Statements)

<?php

$conn = new mysqli("localhost", "root", "", "test\_db");

$stmt = $conn->prepare("SELECT \* FROM users WHERE username = ? AND password = ?");

$stmt->bind\_param("ss", $username, $password);

$username = $\_GET['username'];

$password = $\_GET['password'];

$stmt->execute();

$result = $stmt->get\_result();

if ($result->num\_rows > 0) {

echo "Login successful!";

} else {

echo "Invalid credentials!";

}

?>

* **✅ Secure PHP Code (Using PDO Prepared Statements)**

**<?php**

**$pdo = new PDO("mysql:host=localhost;dbname=test\_db", "root", "");**

**$pdo->setAttribute(PDO::ATTR\_ERRMODE, PDO::ERRMODE\_EXCEPTION);**

**$stmt = $pdo->prepare("SELECT \* FROM users WHERE username = :username AND password = :password");**

**$stmt->execute([**

**'username' => $\_GET['username'],**

**'password' => $\_GET['password']**

**]);**

**if ($stmt->rowCount() > 0) {**

**echo "Login successful!";**

**} else {**

**echo "Invalid credentials!";**

**}**

**?>**

**(B) Use Input Validation and Sanitization**

* Ensure input fields contain only expected values (e.g., letters and numbers).
* Remove special characters using filter\_var().

**$username=filter\_var($\_GET['username'],FILTER\_SANITIZE\_STRING);**

* **(C) Use Least Privilege Database Accounts**
  + Create database users with minimal privileges to reduce the impact of an attack.
  + Avoid using root access in your database connections.
* **🔹 (D) Disable Display of SQL Errors**
  + Attackers can exploit detailed error messages. Hide SQL errors in production:

**ini\_set('display\_errors', 0);**

**error\_reporting(0);**

* **SQL Injection is one of the most dangerous security threats that can lead to data theft, unauthorized access, and database destruction**.

**Prevention Strategies:**

* Always use prepared statements (mysqli or PDO).
* Validate and sanitize user input.
* Use least privilege database accounts.
* Disable error messages in production.

**Practical Exercise: Demonstrate a vulnerable SQL query and then show how to prevent SQL injection using prepared statements.**

**// Vulnerable SQL query (Prone to SQL Injection)**

**$servername = "localhost";**

**$username = "root";**

**$password = "";**

**$database = "test\_db";**

**$conn = new mysqli($servername, $username, $password, $database);**

**if ($conn->connect\_error) {**

**die("Connection failed: " . $conn->connect\_error);**

**}**

**$user\_input = $\_GET['username']; // Simulated user input**

**$query = "SELECT \* FROM users WHERE username = '$user\_input'";**

**$result = $conn->query($query);**

**if ($result->num\_rows > 0) {**

**while ($row = $result->fetch\_assoc()) {**

**echo "User: " . $row['username'] . "<br>";**

**}**

**} else {**

**echo "No user found.";**

**}**

**$conn->close();**

**// Secure version using prepared statements**

**$conn = new mysqli($servername, $username, $password, $database);**

**if ($conn->connect\_error) {**

**die("Connection failed: " . $conn->connect\_error);**

**}**

**$stmt = $conn->prepare("SELECT \* FROM users WHERE username = ?");**

**$stmt->bind\_param("s", $user\_input);**

**$stmt->execute();**

**$result = $stmt->get\_result();**

**if ($result->num\_rows > 0) {**

**while ($row = $result->fetch\_assoc()) {**

**echo "User: " . $row['username'] . "<br>";**

**}**

**} else {**

**echo "No user found.";**

**}**

**$stmt->close();**

**$conn->close();**

**Practical: Exception Handling with Try-Catch for Database Connection and Queries**

**$servername = "localhost";**

**$username = "root";**

**$password = "";**

**$database = "test\_db";**

**try {**

**// Create connection using PDO**

**$conn = new PDO("mysql:host=$servername;dbname=$database", $username, $password);**

**// Set the PDO error mode to exception**

**$conn->setAttribute(PDO::ATTR\_ERRMODE, PDO::ERRMODE\_EXCEPTION);**

**echo "Connected successfully";**

**// Example query execution with exception handling**

**try {**

**$stmt = $conn->prepare("SELECT \* FROM users");**

**$stmt->execute();**

**$result = $stmt->fetchAll(PDO::FETCH\_ASSOC);**

**foreach ($result as $row) {**

**echo "User: " . $row['username'] . "<br>";**

**}**

**} catch (PDOException $e) {**

**echo "Query failed: " . $e->getMessage();**

**}**

**} catch (PDOException $e) {**

**echo "Connection failed: " . $e->getMessage();**

**}**

**Practical Exercise: Implement try-catch blocks in a PHP script to handle exceptions for database connection and query execution.**

**$servername = "localhost";**

**$username = "root";**

**$password = "";**

**$database = "test\_db";**

**try {**

**// Create connection using PDO**

**$conn = new PDO("mysql:host=$servername;dbname=$database", $username, $password);**

**// Set the PDO error mode to exception**

**$conn->setAttribute(PDO::ATTR\_ERRMODE, PDO::ERRMODE\_EXCEPTION);**

**echo "Connected successfully";**

**try {**

**// Example query execution with exception handling**

**$stmt = $conn->prepare("SELECT \* FROM users");**

**$stmt->execute();**

**$result = $stmt->fetchAll(PDO::FETCH\_ASSOC);**

**if ($result) {**

**foreach ($result as $row) {**

**echo "User: " . htmlspecialchars($row['username']) . "<br>";**

**}**

**} else {**

**echo "No users found.";**

**}**

**} catch (PDOException $e) {**

**echo "Query failed: " . $e->getMessage();**

**}**

**} catch (PDOException $e) {**

**echo "Connection failed: " . $e->getMessage();**

**} finally {**

**$conn = null; // Close the connection**

**}**

**Server-Side Validation while Registration using Regular Expressions**

**Practical Exercise: Write a registration form that validates user input (e.g., email, password) using regular expressions before submission.**

**<!DOCTYPE html>**

**<html>**

**<head>**

**<title>Registration Form</title>**

**</head>**

**<body>**

**<form method="post" action="">**

**Email: <input type="text" name="email" required><br>**

**Password: <input type="password" name="password" required><br>**

**<input type="submit" name="submit" value="Register">**

**</form>**

**<?php**

**if ($\_SERVER["REQUEST\_METHOD"] == "POST") {**

**$email = $\_POST['email'];**

**$password = $\_POST['password'];**

**$email\_pattern = "/^[a-zA-Z0-9.\_%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$/";**

**$password\_pattern = "/^(?=.\*[A-Za-z])(?=.\*\d)[A-Za-z\d]{8,}$/"; // Min 8 chars, 1 letter, 1 number**

**if (!preg\_match($email\_pattern, $email)) {**

**echo "Invalid email format.<br>";**

**}**

**if (!preg\_match($password\_pattern, $password)) {**

**echo "Password must be at least 8 characters long and contain at least one letter and one number.<br>";**

**}**

**if (preg\_match($email\_pattern, $email) && preg\_match($password\_pattern, $password)) {**

**echo "Registration successful!";**

**}**

**}**

**?>**

**</body>**

**</html>**

**Send Mail While Registration**

**Practical Exercise: Extend the registration form to send a confirmation email upon successful registration.**

**<!DOCTYPE html>**

**<html>**

**<head>**

**<title>Registration Form</title>**

**</head>**

**<body>**

**<form method="post" action="">**

**Email: <input type="text" name="email" required><br>**

**Password: <input type="password" name="password" required><br>**

**<input type="submit" name="submit" value="Register">**

**</form>**

**<?php**

**if ($\_SERVER["REQUEST\_METHOD"] == "POST") {**

**$email = $\_POST['email'];**

**$password = $\_POST['password'];**

**$email\_pattern = "/^[a-zA-Z0-9.\_%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$/";**

**$password\_pattern = "/^(?=.\*[A-Za-z])(?=.\*\d)[A-Za-z\d]{8,}$/"; // Min 8 chars, 1 letter, 1 number**

**if (!preg\_match($email\_pattern, $email)) {**

**echo "Invalid email format.<br>";**

**}**

**if (!preg\_match($password\_pattern, $password)) {**

**echo "Password must be at least 8 characters long and contain at least one letter and one number.<br>";**

**}**

**if (preg\_match($email\_pattern, $email) && preg\_match($password\_pattern, $password)) {**

**echo "Registration successful!";**

**// Send confirmation email**

**$to = $email;**

**$subject = "Registration Confirmation";**

**$message = "Thank you for registering. Your registration is successful.";**

**$headers = "From: noreply@example.com";**

**if (mail($to, $subject, $message, $headers)) {**

**echo " A confirmation email has been sent to your email address.";**

**} else {**

**echo " Failed to send confirmation email.";**

**}**

**}**

**}**

**?>**

**</body>**

**</html>**

**Session and Cookies**

**THEORY EXERCISE: Explain the differences between sessions and cookies in PHP.**

**Differences Between Sessions and Cookies in PHP**

Sessions and cookies are both used to store user-related data, but they work in different ways and serve different purposes.

| **Feature** | **Sessions** | **Cookies** |
| --- | --- | --- |
| **Storage Location** | Stored on the server | Stored on the user's browser |
| **Persistence** | Ends when the browser is closed (unless configured otherwise) | Can persist for a long time based on expiration settings |
| **Security** | More secure as data is stored on the server | Less secure as data is stored on the client-side and can be modified |
| **Capacity** | No storage limit (depends on server memory) | Typically limited to 4KB per cookie |
| **Access Method** | Accessed via $\_SESSION | Accessed via $\_COOKIE |
| **Lifetime Control** | Can be destroyed using session\_destroy() | Can be set to expire at a specific time using setcookie() |
| **Use Case** | Used for storing sensitive data like login credentials or user preferences | Used for storing less critical data like user preferences, tracking, or remembering login sessions |

**How They Work in PHP**

**1. Sessions** Sessions store data on the server and assign a unique session ID to each user. This session ID is sent to the user's browser via a cookie (PHPSESSID) to maintain state.

**Example of Using a Session:**

**session\_start(); // Start the session**

**$\_SESSION['user'] = 'Nirav'; // Store a value**

**echo $\_SESSION['user']; // Retrieve session data**

2. Cookies Cookies store data in the user's browser and can be accessed on subsequent visits**.**

**Example of Setting a Cookie:**

**setcookie("user", "Nirav", time() + 3600, "/"); // Expires in 1 hour**

**echo $\_COOKIE['user']; // Retrieve cookie data**

**When to Use**

* Use sessions when you need to store sensitive data like user authentication details.
* Use cookies when you need to store non-sensitive, persistent data like user preferences.

**Practical Exercise: Write a script to create a session and store user data, and then retrieve it on a different page. Also, demonstrate how to set and retrieve a cookie.**

**<?php**

**session\_start();**

**// Set session and cookie**

**$\_SESSION['user'] = "John Doe";**

**setcookie("user", "John Doe", time() + 3600, "/"); // 1-hour expiry**

**// Retrieve and display session and cookie data**

**echo "<h2>Session and Cookie Data</h2>";**

**if (isset($\_SESSION['user'])) {**

**echo "Session User: " . $\_SESSION['user'] . "<br>";**

**} else {**

**echo "No session data found.<br>";**

**}**

**if (isset($\_COOKIE['user'])) {**

**echo "Cookie User: " . $\_COOKIE['user'] . "<br>";**

**} else {**

**echo "No cookie found.<br>";**

**}**

**?>**

**<a href="<?php echo $\_SERVER['PHP\_SELF']; ?>">Refresh Page</a>**

**File Upload**

**THEORY EXERCISE: Discuss file upload functionality in PHP and its security implications.**

**File Upload Functionality in PHP**

Uploading files in PHP involves handling form submissions and moving files to a designated directory. However, improper handling can lead to security vulnerabilities.

**Basic File Upload Process**

1. Create an HTML form with enctype="multipart/form-data":

**<form action="upload.php"method="post"enctype="multipart/form-data">**

**<input type="file" name="file">**

**<input type="submit" value="Upload">**

**</form>**

2.Handle the file upload in upload.php:

**<?php**

**if ($\_SERVER["REQUEST\_METHOD"] == "POST") {**

**$target\_dir = "uploads/";**

**$target\_file=$target\_di.basename($\_FILES["file"]["name"]);**

**if(move\_uploaded\_file($\_FILES["file"]["tmp\_name"], $target\_file)) {**

**echo "File uploaded successfully.";**

**} else {**

**echo "File upload failed.";**

**}**

**}**

**?>**

**Security Implications & Best Practices**

**1. Restrict File Types**

Attackerscan upload executable scripts (e.g., .php, .exe), leading to Remote Code Execution (RCE).

**$allowed\_types=['image/jpeg', 'image/png', 'application/pdf'];**

**$file\_type=mime\_content\_type($\_FILES["file"]["tmp\_name"];**

**if (!in\_array($file\_type, $allowed\_types)) {**

**die("Invalid file type.");**

**}**

**2. Limit File Size**

Large files can lead to Denial of Service (DoS) attacks.

* Set limits in php.ini:

upload\_max\_filesize = 2M

post\_max\_size = 2M

* Check size in PHP:

if ($\_FILES["file"]["size"] > 2097152) { // 2MB limit

die("File is too large.");

}

**3. Store Files Securely**

Never store uploaded files in a publicly accessible directory.

* Store outside the document root (/var/www/uploads instead of /var/www/html/uploads).
* Use random names for uploaded files:

$new\_filename=uniqid()."\_".basename($\_FILES["file"]["name"]);

**4. Prevent Execution of Uploaded Files**

Even if an attacker uploads a PHP script, prevent execution by:

* Storing files outside public\_html.
* Disabling execution in .htaccess:

<FilesMatch "\.(php|pl|py|jsp|asp|exe|sh)$">

Order Allow,Deny

Deny from all

</FilesMatch>

**5. Validate File Extensions**

Attackers can rename .php files to .jpg and still execute them.

Use pathinfo() to check extensions.

$allowed\_extensions = ['jpg', 'png', 'pdf'];

$file\_ext=strtolower(pathinfo($\_FILES["file"]["name"],PATHINFO\_EXTEON));

if (!in\_array($file\_ext, $allowed\_extensions)) {

die("Invalid file extension.");

}

**6. Rename and Move Files Securely**

Instead of keeping the original filename:

$safe\_name = uniqid() . "." . $file\_ext;

move\_uploaded\_file($\_FILES["file"]["tmp\_name"], "uploads/" $safe\_name);

**Practical Exercise: Create a file upload form that allows users to upload files and handle the uploaded files safely on the server.**

<!DOCTYPE html>

<html>

<head>

<title>File Upload</title>

</head>

<body>

<form action="upload.php" method="post" enctype="multipart/form-data">

Select file to upload:

<input type="file" name="fileToUpload" required>

<input type="submit" value="Upload File" name="submit">

</form>

</body>

</html>

<?php

if ($\_SERVER["REQUEST\_METHOD"] == "POST") {

$target\_dir = "uploads/";

$target\_file = $target\_dir . basename($\_FILES["fileToUpload"]["name"]);

$uploadOk = 1;

$fileType = strtolower(pathinfo($target\_file, PATHINFO\_EXTENSION));

// Check if file already exists

if (file\_exists($target\_file)) {

echo "Sorry, file already exists.";

$uploadOk = 0;

}

// Allow certain file formats

$allowed\_types = ["jpg", "png", "jpeg", "gif", "pdf", "txt"];

if (!in\_array($fileType, $allowed\_types)) {

echo "Sorry, only JPG, JPEG, PNG, GIF, PDF, and TXT files are allowed.";

$uploadOk = 0;

}

// Check file size (limit: 2MB)

if ($\_FILES["fileToUpload"]["size"] > 2000000) {

echo "Sorry, your file is too large.";

$uploadOk = 0;

}

// If all checks pass, move the uploaded file

if ($uploadOk == 1) {

if (!file\_exists($target\_dir)) {

mkdir($target\_dir, 0777, true);

}

if (move\_uploaded\_file($\_FILES["fileToUpload"]["tmp\_name"], $target\_file)) {

echo "The file " . htmlspecialchars(basename($\_FILES["fileToUpload"]["name"])) . " has been uploaded.";

} else {

echo "Sorry, there was an error uploading your file.";

}

}

}

?>

**PHP with MVC Architecture**

**Practical Exercise: Implement a CRUD application (Create, Read, Update, Delete) using the MVC architecture to manage user data.**

<?php

// models/User.php

class User {

private $conn;

public function \_\_construct($db) {

$this->conn = $db;

}

public function getAllUsers() {

$stmt = $this->conn->prepare("SELECT \* FROM users");

$stmt->execute();

return $stmt->fetchAll(PDO::FETCH\_ASSOC);

}

public function getUser($id) {

$stmt = $this->conn->prepare("SELECT \* FROM users WHERE id = ?");

$stmt->execute([$id]);

return $stmt->fetch(PDO::FETCH\_ASSOC);

}

public function createUser($name, $email) {

$stmt = $this->conn->prepare("INSERT INTO users (name, email) VALUES (?, ?)");

return $stmt->execute([$name, $email]);

}

public function updateUser($id, $name, $email) {

$stmt = $this->conn->prepare("UPDATE users SET name = ?, email = ? WHERE id = ?");

return $stmt->execute([$name, $email, $id]);

}

public function deleteUser($id) {

$stmt = $this->conn->prepare("DELETE FROM users WHERE id = ?");

return $stmt->execute([$id]);

}

}

// controllers/UserController.php

require\_once "models/User.php";

require\_once "config/database.php";

class UserController {

private $user;

public function \_\_construct($db) {

$this->user = new User($db);

}

public function index() {

$users = $this->user->getAllUsers();

include "views/user\_list.php";

}

public function create() {

if ($\_POST) {

$this->user->createUser($\_POST['name'], $\_POST['email']);

header("Location: index.php");

}

include "views/user\_form.php";

}

public function update($id) {

$user = $this->user->getUser($id);

if ($\_POST) {

$this->user->updateUser($id, $\_POST['name'], $\_POST['email']);

header("Location: index.php");

}

include "views/user\_form.php";

}

public function delete($id) {

$this->user->deleteUser($id);

header("Location: index.php");

}

}

// index.php (Front Controller)

require\_once "controllers/UserController.php";

$db = new PDO("mysql:host=localhost;dbname=test\_db", "root", "");

$userController = new UserController($db);

$action = $\_GET['action'] ?? 'index';

$id = $\_GET['id'] ?? null;

if (method\_exists($userController, $action)) {

$userController->$action($id);

} else {

echo "Invalid action.";

}

// views/user\_list.php

foreach ($users as $user) {

echo "<p>{$user['name']} ({$user['email']})

<a href='index.php?action=update&id={$user['id']}'>Edit</a> |

<a href='index.php?action=delete&id={$user['id']}'>Delete</a></p>";

}

echo "<a href='index.php?action=create'>Add New User</a>";

// views/user\_form.php

?>

<form method="POST">

<input type="text" name="name" value="<?= $user['name'] ?? '' ?>" placeholder="Name" required>

<input type="email" name="email" value="<?= $user['email'] ?? '' ?>" placeholder="Email" required>

<input type="submit" value="Save">

</form>

**Insert, Update, Delete MVC**

**Practical Exercise: Extend the CRUD application to include functionalities for inserting, updating, and deleting user records, ensuring proper separation of concerns in the MVC structure.**

<?php

// models/User.php

class User {

private $conn;

public function \_\_construct($db) {

$this->conn = $db;

}

public function getAllUsers() {

$stmt = $this->conn->prepare("SELECT \* FROM users");

$stmt->execute();

return $stmt->fetchAll(PDO::FETCH\_ASSOC);

}

public function getUser($id) {

$stmt = $this->conn->prepare("SELECT \* FROM users WHERE id = ?");

$stmt->execute([$id]);

return $stmt->fetch(PDO::FETCH\_ASSOC);

}

public function createUser($name, $email) {

$stmt = $this->conn->prepare("INSERT INTO users (name, email) VALUES (?, ?)");

return $stmt->execute([$name, $email]);

}

public function updateUser($id, $name, $email) {

$stmt = $this->conn->prepare("UPDATE users SET name = ?, email = ? WHERE id = ?");

return $stmt->execute([$name, $email, $id]);

}

public function deleteUser($id) {

$stmt = $this->conn->prepare("DELETE FROM users WHERE id = ?");

return $stmt->execute([$id]);

}

}

// controllers/UserController.php

require\_once "models/User.php";

require\_once "config/database.php";

class UserController {

private $user;

public function \_\_construct($db) {

$this->user = new User($db);

}

public function index() {

$users = $this->user->getAllUsers();

include "views/user\_list.php";

}

public function create() {

if ($\_POST) {

$this->user->createUser($\_POST['name'], $\_POST['email']);

header("Location: index.php");

exit();

}

include "views/user\_form.php";

}

public function update($id) {

$user = $this->user->getUser($id);

if ($\_POST) {

$this->user->updateUser($id, $\_POST['name'], $\_POST['email']);

header("Location: index.php");

exit();

}

include "views/user\_form.php";

}

public function delete($id) {

$this->user->deleteUser($id);

header("Location: index.php");

exit();

}

}

// index.php (Front Controller)

require\_once "controllers/UserController.php";

$db = new PDO("mysql:host=localhost;dbname=test\_db", "root", "");

$userController = new UserController($db);

$action = $\_GET['action'] ?? 'index';

$id = $\_GET['id'] ?? null;

if (method\_exists($userController, $action)) {

$userController->$action($id);

} else {

echo "Invalid action.";

}

// views/user\_list.php

?>

<h2>User List</h2>

<a href='index.php?action=create'>Add New User</a>

<?php foreach ($users as $user): ?>

<p>

<?= htmlspecialchars($user['name']) ?> (<?= htmlspecialchars($user['email']) ?>)

<a href='index.php?action=update&id=<?= $user['id'] ?>'>Edit</a> |

<a href='index.php?action=delete&id=<?= $user['id'] ?>' onclick='return confirm("Are you sure?")'>Delete</a>

</p>

<?php endforeach;

// views/user\_form.php

?>

<h2><?= isset($user) ? "Edit User" : "Add New User" ?></h2>

<form method="POST">

<input type="text" name="name" value="<?= htmlspecialchars($user['name'] ?? '') ?>" placeholder="Name" required>

<input type="email" name="email" value="<?= htmlspecialchars($user['email'] ?? '') ?>" placeholder="Email" required>

<input type="submit" value="Save">

</form>

<a href='index.php'>Back to List</a>

**Extra Practise for Grade A**

**1. Practical Exercise: Develop a class hierarchy for a simple e-commerce system with classes like Product, Category, and Order. Implement encapsulation by using private properties and public methods to access them.**

**<?php**

**// Product class**

**class Product {**

**private $id;**

**private $name;**

**private $price;**

**private $category;**

**public function \_\_construct($id, $name, $price, Category $category) {**

**$this->id = $id;**

**$this->name = $name;**

**$this->price = $price;**

**$this->category = $category;**

**}**

**public function getId() {**

**return $this->id;**

**}**

**public function getName() {**

**return $this->name;**

**}**

**public function getPrice() {**

**return $this->price;**

**}**

**public function getCategory() {**

**return $this->category;**

**}**

**public function setPrice($price) {**

**if ($price > 0) {**

**$this->price = $price;**

**}**

**}**

**}**

**// Category class**

**class Category {**

**private $id;**

**private $name;**

**public function \_\_construct($id, $name) {**

**$this->id = $id;**

**$this->name = $name;**

**}**

**public function getId() {**

**return $this->id;**

**}**

**public function getName() {**

**return $this->name;**

**}**

**}**

**// Order class**

**class Order {**

**private $id;**

**private $products = [];**

**private $totalAmount = 0;**

**public function \_\_construct($id) {**

**$this->id = $id;**

**}**

**public function addProduct(Product $product) {**

**$this->products[] = $product;**

**$this->totalAmount += $product->getPrice();**

**}**

**public function getTotalAmount() {**

**return $this->totalAmount;**

**}**

**public function getProducts() {**

**return $this->products;**

**}**

**}**

**// Example usage**

**$category = new Category(1, "Electronics");**

**$product1 = new Product(101, "Smartphone", 500, $category);**

**$product2 = new Product(102, "Laptop", 1200, $category);**

**$order = new Order(1);**

**$order->addProduct($product1);**

**$order->addProduct($product2);**

**echo "Total Order Amount: $" . $order->getTotalAmount();**

**?>**

**Class**

**2. Practical Exercise: Create a class called Book with properties like title, author, and price. Implement a method to apply a discount to the book's price and return the new price.**

**<?php**

**class Book {**

**private $title;**

**private $author;**

**private $price;**

**public function \_\_construct($title, $author, $price) {**

**$this->title = $title;**

**$this->author = $author;**

**$this->price = $price;**

**}**

**public function getTitle() {**

**return $this->title;**

**}**

**public function getAuthor() {**

**return $this->author;**

**}**

**public function getPrice() {**

**return $this->price;**

**}**

**public function applyDiscount($percentage) {**

**if ($percentage > 0 && $percentage <= 100) {**

**$discountAmount = ($this->price \* $percentage) / 100;**

**$this->price -= $discountAmount;**

**}**

**return $this->price;**

**}**

**}**

**// Example usage**

**$book = new Book("The Great Gatsby", "F. Scott Fitzgerald", 20);**

**echo "Original Price: $" . $book->getPrice() . "\n";**

**$newPrice = $book->applyDiscount(10);**

**echo "Discounted Price: $" . $newPrice . "\n";**

**?>**

**Object**

**3. Practical Exercise: Instantiate an object of the Book class and demonstrate the usage of its methods. Create multiple instances of Book and display their details in a formatted manner.**

**<?php**

**class Book {**

**private $title;**

**private $author;**

**private $price;**

**public function \_\_construct($title, $author, $price) {**

**$this->title = $title;**

**$this->author = $author;**

**$this->price = $price;**

**}**

**public function getTitle() {**

**return $this->title;**

**}**

**public function getAuthor() {**

**return $this->author;**

**}**

**public function getPrice() {**

**return $this->price;**

**}**

**public function applyDiscount($percentage) {**

**if ($percentage > 0 && $percentage <= 100) {**

**$discountAmount = ($this->price \* $percentage) / 100;**

**$this->price -= $discountAmount;**

**}**

**return $this->price;**

**}**

**}**

**// Instantiate multiple books**

**$book1 = new Book("The Great Gatsby", "F. Scott Fitzgerald", 20);**

**$book2 = new Book("1984", "George Orwell", 15);**

**$book3 = new Book("To Kill a Mockingbird", "Harper Lee", 18);**

**// Apply discounts**

**$book1->applyDiscount(10);**

**$book2->applyDiscount(15);**

**$book3->applyDiscount(5);**

**// Display book details**

**$books = [$book1, $book2, $book3];**

**foreach ($books as $book) {**

**echo "Title: " . $book->getTitle() . "\n";**

**echo "Author: " . $book->getAuthor() . "\n";**

**echo "Price after discount: $" . $book->getPrice() . "\n";**

**echo "-----------------------------\n";**

**}**

**?>**

**Extends**

**4. Practical Exercise: Create a base class called Employee with properties like name and salary. Extend it with subclasses FullTimeEmployee and PartTimeEmployee, each having specific methods to calculate bonuses.**

<?php

// Base class: Employee

class Employee {

protected $name;

protected $salary;

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

$this->name = $name;

$this->salary = $salary;

}

public function getName() {

return $this->name;

}

public function getSalary() {

return $this->salary;

}

}

// Subclass: FullTimeEmployee

class FullTimeEmployee extends Employee {

public function calculateBonus() {

return $this->salary \* 0.10; // 10% bonus

}

}

// Subclass: PartTimeEmployee

class PartTimeEmployee extends Employee {

public function calculateBonus() {

return $this->salary \* 0.05; // 5% bonus

}

}

// Example usage

$fullTimeEmp = new FullTimeEmployee("Alice", 5000);

$partTimeEmp = new PartTimeEmployee("Bob", 2000);

echo "Employee: " . $fullTimeEmp->getName() . " - Salary: $" .

$fullTimeEmp->getSalary()."-Bonus:$" .$fullTimeEmp>calculateBonus() . "\n";

echo "Employee: " . $partTimeEmp->getName() . " - Salary: $" .

$partTimeEmp->getSalary() . " - Bonus: $" . $partTimeEmp->calculateBonus() . "\n";

?>

**Overloading**

1. **Practical Exercise: Create a Calculator class with a method calculate that can add, subtract, or multiply based on the number and type of arguments passed.**

<?php

class Calculator {

public function calculate(...$args) {

$operation = array\_shift($args);

if (empty($args)) {

throw new InvalidArgumentException("No numbers provided for calculation.");

}

switch ($operation) {

case 'add':

return array\_sum($args);

case 'subtract':

return array\_shift($args) - array\_sum($args);

case 'multiply':

return array\_product($args);

default:

throw new InvalidArgumentException("Invalid operation. Use 'add', 'subtract', or 'multiply'.");

}

}

}

// Example usage

$calc = new Calculator();

echo "Addition: " . $calc->calculate('add', 5, 10, 15) . "\n";

echo "Subtraction: " . $calc->calculate('subtract', 50, 20, 10) . "\n";

echo "Multiplication: " . $calc->calculate('multiply', 2, 3, 4) . "\n";

?>

**Abstraction Interface**

**6.Practical Exercise: Define an interface PaymentInterface with methods like processPayment(), refund(), and implement it in classes like CreditCardPayment and PaypalPayment.**

<?php

// Define the PaymentInterface

interface PaymentInterface {

public function processPayment($amount);

public function refund($amount);

}

// Implement CreditCardPayment class

class CreditCardPayment implements PaymentInterface {

public function processPayment($amount) {

return "Processing credit card payment of $$amount";

}

public function refund($amount) {

return "Refunding $$amount to credit card";

}

}

// Implement PaypalPayment class

class PaypalPayment implements PaymentInterface {

public function processPayment($amount) {

return "Processing PayPal payment of $$amount";

}

public function refund($amount) {

return "Refunding $$amount via PayPal";

}

}

// Example usage

$creditCardPayment = new CreditCardPayment();

echo $creditCardPayment->processPayment(100) . "\n";

echo $creditCardPayment->refund(50) . "\n";

$paypalPayment = new PaypalPayment();

echo $paypalPayment->processPayment(200) . "\n";

echo $paypalPayment->refund(75) . "\n";

?>

**Constructor**

**7. Practical Exercise: Create a class Student with properties like name, age, and grade. Use a constructor to initialize these properties and a method to display student details.**

<?php

class Student {

private $name;

private $age;

private $grade;

// Constructor to initialize properties

public function \_\_construct($name, $age, $grade) {

$this->name = $name;

$this->age = $age;

$this->grade = $grade;

}

// Method to display student details

public function displayDetails() {

echo "Student Name: " . $this->name . "\n";

echo "Age: " . $this->age . "\n";

echo "Grade: " . $this->grade . "\n";

}

}

// Example usage

$student1 = new Student("Alice", 20, "A");

$student1->displayDetails();

?>

**Destructor**

1. **Practical Exercise: Write a class that connects to a database, with a destructor that closes the connection when the object is destroyed.**

<?php

class DatabaseConnection {

private $host = "localhost";

private $username = "root";

private $password = "";

private $dbname = "test\_db";

private $conn;

// Constructor to establish database connection

public function \_\_construct() {

try {

$this->conn = new PDO("mysql:host={$this->host};dbname={$this->dbname}", $this->username, $this->password);

$this->conn->setAttribute(PDO::ATTR\_ERRMODE, PDO::ERRMODE\_EXCEPTION);

echo "Connected successfully\n";

} catch (PDOException $e) {

echo "Connection failed: " . $e->getMessage() . "\n";

}

}

// Destructor to close the connection

public function \_\_destruct() {

$this->conn = null;

echo "Connection closed\n";

}

}

// Example usage

$db = new DatabaseConnection();

?>

**Magic Methods**

**9. Practical Exercise: Create a class that uses the \_\_set() and \_\_get() magic methods to dynamically create and access properties based on user input.**

<?php

class DynamicProperties {

private $properties = [];

// Magic method to set properties dynamically

public function \_\_set($name, $value) {

$this->properties[$name] = $value;

}

// Magic method to get properties dynamically

public function \_\_get($name) {

return $this->properties[$name] ?? "Property '$name' does not exist.";

}

}

// Example usage

$obj = new DynamicProperties();

$obj->name = "Alice";

$obj->age = 25;

echo "Name: " . $obj->name . "\n";

echo "Age: " . $obj->age . "\n";

echo "Grade: " . $obj->grade . "\n";

?>

**Scope Resolution**

**10. Practical Exercise: Define a class with static properties and methods to keep track of the number of instances created. Use the scope resolution operator to access these static members.**

<?php

class InstanceTracker {

private static $instanceCount = 0;

public function \_\_construct() {

self::$instanceCount++;

}

public static function getInstanceCount() {

return self::$instanceCount;

}

}

// Example usage

$instance1 = new InstanceTracker();

$instance2 = new InstanceTracker();

$instance3 = new InstanceTracker();

echo "Number of instances created: " . InstanceTracker::getInstanceCount() . "\n";

?>

**Traits**

**11. Practical Exercise: Create two traits: Logger and Notifier. Use these traits in a class User to log user activities and send notifications.**

<?php

trait Logger {

public function log($message) {

echo "[LOG]: " . $message . "\n";

}

}

trait Notifier {

public function notify($message) {

echo "[NOTIFICATION]: " . $message . "\n";

}

}

class User {

use Logger, Notifier;

private $name;

public function \_\_construct($name) {

$this->name = $name;

$this->log("User '$name' created.");

}

public function performAction($action) {

$this->log("User '$this->name' performed action: $action");

$this->notify("Hey $this->name, you have a new notification about: $action");

}

}

// Example usage

$user = new User("Alice");

$user->performAction("Logged in");

?>

**Visibility**

**12. Practical Exercise: Develop a class Account with properties for username (public), password(private), and accountBalance (protected). Demonstrate how to access these properties in a derived class.**

<?php

class Account {

public $username;

private $password;

protected $accountBalance;

public function \_\_construct($username, $password, $accountBalance) {

$this->username = $username;

$this->password = $password;

$this->accountBalance = $accountBalance;

}

public function getUsername() {

return $this->username;

}

protected function getBalance() {

return $this->accountBalance;

}

}

class SavingsAccount extends Account {

public function displayBalance() {

return "User: {$this->username}, Balance: {$this->getBalance()}";

}

}

// Example usage

$account = new SavingsAccount("JohnDoe", "securePass123", 5000);

echo $account->getUsername() . "\n";

echo $account->displayBalance() . "\n";

?>

**Type Hinting**

**13. Practical Exercise: Write a method in a class Order that accepts an array of products (type-hinted)and calculates the total order amount.**

<?php

class Product {

public $name;

public $price;

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

$this->name = $name;

$this->price = $price;

}

}

class Order {

public function calculateTotal(array $products): float {

$total = 0;

foreach ($products as $product) {

$total += $product->price;

}

return $total;

}

}

// Example usage

$product1 = new Product("Laptop", 1200.50);

$product2 = new Product("Mouse", 25.75);

$product3 = new Product("Keyboard", 45.30);

$order = new Order();

echo "Total Order Amount: " . $order->calculateTotal([$product1, $product2, $product3]) . "\n";

?>

Final Keyword

14. Practical Exercise: Create a base class Animal and a final class Dog. Attempt to extend Dog and demonstrate the restriction imposed by the final keyword.

<?php

class Animal {

public function makeSound() {

return "Some generic animal sound";

}

}

final class Dog extends Animal {

public function makeSound() {

return "Bark";

}

}

// Attempting to extend the final class will result in an error

/\*

class Puppy extends Dog {

// This will cause a fatal error

}

\*/

// Example usage

$dog = new Dog();

echo $dog->makeSound() . "\n";

?>

**Email Security Function**

**15. Practical Exercise: Write a function that sanitizes user input for an email address, validates it, and throws an exception if it fails validation.**

<?php

// Function to sanitize and validate email

function validateEmail(string $email): string {

$sanitizedEmail = filter\_var($email, FILTER\_SANITIZE\_EMAIL);

if (!filter\_var($sanitizedEmail, FILTER\_VALIDATE\_EMAIL)) {

throw new Exception("Invalid email address.");

}

return $sanitizedEmail;

}

// Example usage

try {

$email = "test@example.com";

echo "Valid Email: " . validateEmail($email) . "\n";

} catch (Exception $e) {

echo "Error: " . $e->getMessage() . "\n";

}

?>

**File Handling**

**16. Practical Exercise: Create a script that uploads a file and reads its content. Implement error handling to manage any file-related exceptions.**

<?php

// File upload and read script

if ($\_SERVER['REQUEST\_METHOD'] == 'POST' && isset($\_FILES['file'])) {

try {

if ($\_FILES['file']['error'] !== UPLOAD\_ERR\_OK) {

throw new Exception("File upload error.");

}

$filePath = \_\_DIR\_\_ . '/uploads/' . basename($\_FILES['file']['name']);

if (!move\_uploaded\_file($\_FILES['file']['tmp\_name'], $filePath)) {

throw new Exception("Failed to move uploaded file.");

}

echo "File uploaded successfully: " . $filePath . "\n";

// Read file content

$content = file\_get\_contents($filePath);

if ($content === false) {

throw new Exception("Failed to read file content.");

}

echo "File Content:\n" . htmlspecialchars($content) . "\n";

} catch (Exception $e) {

echo "Error: " . $e->getMessage() . "\n";

}

}

?>

<form action="" method="post" enctype="multipart/form-data">

Select file to upload:

<input type="file" name="file">

<input type="submit" value="Upload File">

</form>

**Handling Emails**

**17. Practical Exercise: Develop a function to send a welcome email to a user upon registration, ensuring the email format is validated first.**

// Function to send welcome email

function sendWelcomeEmail(string $email, string $name): void {

try {

$validatedEmail = validateEmail($email);

$subject = "Welcome to Our Platform, $name!";

$message = "Hello $name,\n\nThank you for registering with us. We are excited to have you on board!\n\nBest Regards,\nThe Team";

$headers = "From: no-reply@example.com\r\n" .

"Reply-To: support@example.com\r\n" .

"X-Mailer: PHP/" . phpversion();

if (!mail($validatedEmail, $subject, $message, $headers)) {

throw new Exception("Failed to send email.");

}

echo "Welcome email sent successfully to $validatedEmail.\n";

} catch (Exception $e) {

echo "Error: " . $e->getMessage() . "\n";

}

}

**MVC Architecture**

**18. Practical Exercise: Extend the simple MVC application to include a model for managing user profiles, a view for displaying user details, and a controller for handling user actions.**

<?php

// Model: UserModel.php

class UserModel {

private $users = [

1 => ["name" => "John Doe", "email" => "john@example.com", "age" => 25],

2 => ["name" => "Jane Smith", "email" => "jane@example.com", "age" => 28]

];

public function getUserById($id) {

return $this->users[$id] ?? null;

}

}

// View: UserView.php

class UserView {

public function renderUserProfile($user) {

if ($user) {

echo "<h2>User Profile</h2>";

echo "<p><strong>Name:</strong> {$user['name']}</p>";

echo "<p><strong>Email:</strong> {$user['email']}</p>";

echo "<p><strong>Age:</strong> {$user['age']}</p>";

} else {

echo "<p>User not found.</p>";

}

}

}

// Controller: UserController.php

class UserController {

private $model;

private $view;

public function \_\_construct() {

$this->model = new UserModel();

$this->view = new UserView();

}

public function showUserProfile($userId) {

$user = $this->model->getUserById($userId);

$this->view->renderUserProfile($user);

}

}

// Entry Point: index.php

$userId = $\_GET['id'] ?? 1; // Default to user ID 1

$controller = new UserController();

$controller->showUserProfile($userId);

?>

**Practical Example: Implementation of all the OOPs Concepts**

**19. Practical Exercise: Develop a project that simulates a library system with classes for User, Book, and Transaction, applying all OOP principles.**

**<?php**

**// Model: User.php**

**class User {**

**private $name;**

**private $email;**

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

**$this->name = $name;**

**$this->email = $email;**

**}**

**public function getName() {**

**return $this->name;**

**}**

**public function getEmail() {**

**return $this->email;**

**}**

**}**

**// Model: Book.php**

**class Book {**

**private $title;**

**private $author;**

**public function \_\_construct($title, $author) {**

**$this->title = $title;**

**$this->author = $author;**

**}**

**public function getTitle() {**

**return $this->title;**

**}**

**public function getAuthor() {**

**return $this->author;**

**}**

**}**

**// Model: Transaction.php**

**class Transaction {**

**private $user;**

**private $book;**

**private $date;**

**public function \_\_construct(User $user, Book $book) {**

**$this->user = $user;**

**$this->book = $book;**

**$this->date = date("Y-m-d H:i:s");**

**}**

**public function getTransactionDetails() {**

**return "{$this->user->getName()} borrowed '{$this->book->getTitle()}' by {$this->book->getAuthor()} on {$this->date}.";**

**}**

**}**

**// Example Usage**

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

**$book = new Book("The Great Gatsby", "F. Scott Fitzgerald");**

**$transaction = new Transaction($user, $book);**

**echo $transaction->getTransactionDetails();**

**?>**

**Connection with MySQL Database**

**20. Practical Exercise: Write a class Database that handles database connections and queries. Use this class in another script to fetch user data from a users table.**

<?php

// Database Connection: Database.php

class Database {

private $host = "localhost";

private $db\_name = "library";

private $username = "root";

private $password = "";

private $conn;

public function \_\_construct() {

try {

$this->conn = new PDO("mysql:host={$this->host};dbname={$this->db\_name}", $this->username, $this->password);

$this->conn->setAttribute(PDO::ATTR\_ERRMODE, PDO::ERRMODE\_EXCEPTION);

} catch (PDOException $e) {

die("Connection failed: " . $e->getMessage());

}

}

public function query($sql, $params = []) {

$stmt = $this->conn->prepare($sql);

$stmt->execute($params);

return $stmt;

}

public function \_\_destruct() {

$this->conn = null;

}

}

// Model: User.php

class User {

private $name;

private $email;

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

$this->name = $name;

$this->email = $email;

}

public function getName() {

return $this->name;

}

public function getEmail() {

return $this->email;

}

public static function getAllUsers(Database $db) {

$stmt = $db->query("SELECT name, email FROM users");

return $stmt->fetchAll(PDO::FETCH\_ASSOC);

}

}

// Model: Book.php

class Book {

private $title;

private $author;

public function \_\_construct($title, $author) {

$this->title = $title;

$this->author = $author;

}

public function getTitle() {

return $this->title;

}

public function getAuthor() {

return $this->author;

}

}

// Model: Transaction.php

class Transaction {

private $user;

private $book;

private $date;

public function \_\_construct(User $user, Book $book) {

$this->user = $user;

$this->book = $book;

$this->date = date("Y-m-d H:i:s");

}

public function getTransactionDetails() {

return "{$this->user->getName()} borrowed '{$this->book->getTitle()}' by {$this->book->getAuthor()} on {$this->date}.";

}

}

// Example Usage

$db = new Database();

$users = User::getAllUsers($db);

foreach ($users as $userData) {

echo "User: {$userData['name']}, Email: {$userData['email']}<br>";

}

?>

**SQL Injection**

**21. Practical Exercise: Create a vulnerable PHP script that demonstrates SQL injection. Then, rewrite it using prepared statements to prevent SQL injection attacks.**

<?php

// Database Connection: Database.php

class Database {

private $host = "localhost";

private $db\_name = "library";

private $username = "root";

private $password = "";

private $conn;

public function \_\_construct() {

try {

$this->conn = new PDO(

"mysql:host={$this->host};dbname={$this->db\_name}",

$this->username,

$this->password

);

$this->conn->setAttribute(PDO::ATTR\_ERRMODE, PDO::ERRMODE\_EXCEPTION);

} catch (PDOException $e) {

die("Connection failed: " . $e->getMessage());

}

}

public function query($sql, $params = []) {

$stmt = $this->conn->prepare($sql);

$stmt->execute($params);

return $stmt;

}

public function \_\_destruct() {

$this->conn = null;

}

}

// Vulnerable Script: SQL Injection

if (isset($\_GET['email'])) {

$db = new Database();

$email = $\_GET['email'];

$sql = "SELECT \* FROM users WHERE email = '$email'"; // Vulnerable to SQL Injection

$stmt = $db->query($sql);

$user = $stmt->fetch(PDO::FETCH\_ASSOC);

if ($user) {

echo "User found: " . $user['name'];

} else {

echo "User not found.";

}

}

// Secure Script: Prevent SQL Injection with Prepared Statements

if (isset($\_GET['secure\_email'])) {

$db = new Database();

$email = $\_GET['secure\_email'];

$sql = "SELECT \* FROM users WHERE email = ?";

$stmt = $db->query($sql, [$email]);

$user = $stmt->fetch(PDO::FETCH\_ASSOC);

if ($user) {

echo "User found: " . $user['name'];

} else {

echo "User not found.";

}

}

?>

**Practical: Exception Handling with Try-Catch for Database Connection and Queries**

**22. Practical Exercise: Implement a complete registration process with a database connection that uses try-catch blocks to handle exceptions for all operations.**

<?php

// Database Connection: Database.php

class Database {

private $host = "localhost";

private $db\_name = "library";

private $username = "root";

private $password = "";

private $conn;

public function \_\_construct() {

try {

$this->conn = new PDO("mysql:host={$this->host};dbname={$this->db\_name}", $this->username, $this->password);

$this->conn->setAttribute(PDO::ATTR\_ERRMODE, PDO::ERRMODE\_EXCEPTION);

} catch (PDOException $e) {

die("Connection failed: " . $e->getMessage());

}

}

public function query($sql, $params = []) {

try {

$stmt = $this->conn->prepare($sql);

$stmt->execute($params);

return $stmt;

} catch (PDOException $e) {

die("Query failed: " . $e->getMessage());

}

}

public function \_\_destruct() {

$this->conn = null;

}

}

// Registration Process: Register.php

if ($\_SERVER['REQUEST\_METHOD'] == 'POST') {

$db = new Database();

$name = $\_POST['name'];

$email = $\_POST['email'];

$password = password\_hash($\_POST['password'], PASSWORD\_DEFAULT);

try {

$sql = "INSERT INTO users (name, email, password) VALUES (?, ?, ?)";

$db->query($sql, [$name, $email, $password]);

echo "Registration successful!";

} catch (Exception $e) {

echo "Error: " . $e->getMessage();

}

}

?>

<!DOCTYPE html>

<html>

<head>

<title>Register</title>

</head>

<body>

<form method="post" action="">

<label>Name:</label>

<input type="text" name="name" required><br>

<label>Email:</label>

<input type="email" name="email" required><br>

<label>Password:</label>

<input type="password" name="password" required><br>

<button type="submit">Register</button>

</form>

</body>

</html>

**Server-Side Validation while Registration using Regular Expressions**

**23. Practical Exercise: Write a PHP script that validates user inputs (username, password, email) using regular expressions, providing feedback on any validation errors.**

<?php

// Validate User Input

function validateInput($name, $email, $password) {

$errors = [];

if (!preg\_match("/^[a-zA-Z0-9\_]{3,20}$/", $name)) {

$errors[] = "Username must be 3-20 characters long and contain only letters, numbers, and underscores.";

}

if (!filter\_var($email, FILTER\_VALIDATE\_EMAIL)) {

$errors[] = "Invalid email format.";

}

if (!preg\_match("/^(?=.\*[A-Za-z])(?=.\*\d)[A-Za-z\d]{6,}$/", $password)) {

$errors[] = "Password must be at least 6 characters long and include both letters and numbers.";

}

return $errors;

}

?>

**Send Mail While Registration**

**24. Practical Exercise: Extend the registration process to send a confirmation email to the user after successful registration and validate the email format.**

<?php

// Send Confirmation Email

function sendConfirmationEmail($email, $name) {

$subject = "Welcome to Our Platform!";

$message = "Hello $name,\n\nThank you for registering. Your account has been created successfully!\n\nBest regards,\nThe Team";

$headers = "From: noreply@yourdomain.com";

if (!mail($email, $subject, $message, $headers)) {

throw new Exception("Failed to send confirmation email.");

}

}

// Registration Process: Register.php

if ($\_SERVER['REQUEST\_METHOD'] == 'POST') {

$db = new Database();

$name = $\_POST['name'];

$email = $\_POST['email'];

$password = $\_POST['password'];

$errors = validateInput($name, $email, $password);

if (empty($errors)) {

try {

$hashedPassword = password\_hash($password, PASSWORD\_DEFAULT);

$sql = "INSERT INTO users (name, email, password) VALUES (?, ?, ?)";

$db->query($sql, [$name, $email, $hashedPassword]);

// Send Confirmation Email

sendConfirmationEmail($email, $name);

echo "Registration successful! A confirmation email has been sent.";

} catch (Exception $e) {

echo "Error: " . $e->getMessage();

}

} else {

foreach ($errors as $error) {

echo "<p style='color: red;'>$error</p>";

}

}

}

?>

**Session and Cookies**

**25. Practical Exercise: Implement a login system that uses sessions to keep track of user authentication and demonstrates cookie usage for "Remember Me" functionality.**

<?php

session\_start();

// Database Connection: Database.php

class Database {

private $host = "localhost";

private $db\_name = "library";

private $username = "root";

private $password = "";

private $conn;

public function \_\_construct() {

try {

$this->conn = new PDO("mysql:host={$this->host};dbname={$this->db\_name}", $this->username, $this->password);

$this->conn->setAttribute(PDO::ATTR\_ERRMODE, PDO::ERRMODE\_EXCEPTION);

} catch (PDOException $e) {

die("Connection failed: " . $e->getMessage());

}

}

public function query($sql, $params = []) {

try {

$stmt = $this->conn->prepare($sql);

$stmt->execute($params);

return $stmt;

} catch (PDOException $e) {

die("Query failed: " . $e->getMessage());

}

}

public function \_\_destruct() {

$this->conn = null;

}

}

// User Authentication

function authenticateUser($email, $password) {

$db = new Database();

$sql = "SELECT \* FROM users WHERE email = ?";

$stmt = $db->query($sql, [$email]);

$user = $stmt->fetch(PDO::FETCH\_ASSOC);

if ($user && password\_verify($password, $user['password'])) {

$\_SESSION['user'] = $user['name'];

return true;

}

return false;

}

// Remember Me functionality

if (isset($\_POST['remember\_me'])) {

setcookie("user\_email", $\_POST['email'], time() + (86400 \* 30), "/");

}

// Login Process: login.php

if ($\_SERVER['REQUEST\_METHOD'] == 'POST' && isset($\_POST['login'])) {

$email = $\_POST['email'];

$password = $\_POST['password'];

if (authenticateUser($email, $password)) {

echo "Login successful! Welcome, " . $\_SESSION['user'] . "!";

} else {

echo "Invalid email or password.";

}

}

// Logout Process: logout.php

if (isset($\_GET['logout'])) {

session\_destroy();

setcookie("user\_email", "", time() - 3600, "/");

header("Location: login.php");

}

?>

<!DOCTYPE html>

<html>

<head>

<title>Login</title>

</head>

<body>

<form method="post" action="">

<label>Email:</label>

<input type="email" name="email" value="<?php echo isset($\_COOKIE['user\_email']) ? $\_COOKIE['user\_email'] : ''; ?>" required><br>

<label>Password:</label>

<input type="password" name="password" required><br>

<label>

<input type="checkbox" name="remember\_me"> Remember Me

</label><br>

<button type="submit" name="login">Login</button>

</form>

<a href="?logout=true">Logout</a>

</body>

</html>

**File Upload**

**26. Practical Exercise: Create a file upload feature that allows users to upload images. Ensure that the uploaded images are checked for file type and size for security.**

<?php

session\_start();

// Database Connection: Database.php

class Database {

private $host = "localhost";

private $db\_name = "library";

private $username = "root";

private $password = "";

private $conn;

public function \_\_construct() {

try {

$this->conn = new PDO("mysql:host={$this->host};dbname={$this->db\_name}", $this->username, $this->password);

$this->conn->setAttribute(PDO::ATTR\_ERRMODE, PDO::ERRMODE\_EXCEPTION);

} catch (PDOException $e) {

die("Connection failed: " . $e->getMessage());

}

}

public function query($sql, $params = []) {

try {

$stmt = $this->conn->prepare($sql);

$stmt->execute($params);

return $stmt;

} catch (PDOException $e) {

die("Query failed: " . $e->getMessage());

}

}

public function \_\_destruct() {

$this->conn = null;

}

}

// User Authentication

function authenticateUser($email, $password) {

$db = new Database();

$sql = "SELECT \* FROM users WHERE email = ?";

$stmt = $db->query($sql, [$email]);

$user = $stmt->fetch(PDO::FETCH\_ASSOC);

if ($user && password\_verify($password, $user['password'])) {

$\_SESSION['user'] = $user['name'];

return true;

}

return false;

}

// Remember Me functionality

if (isset($\_POST['remember\_me'])) {

setcookie("user\_email", $\_POST['email'], time() + (86400 \* 30), "/");

}

// Login Process: login.php

if ($\_SERVER['REQUEST\_METHOD'] == 'POST' && isset($\_POST['login'])) {

$email = $\_POST['email'];

$password = $\_POST['password'];

if (authenticateUser($email, $password)) {

echo "Login successful! Welcome, " . $\_SESSION['user'] . "!";

} else {

echo "Invalid email or password.";

}

}

// Logout Process: logout.php

if (isset($\_GET['logout'])) {

session\_destroy();

setcookie("user\_email", "", time() - 3600, "/");

header("Location: login.php");

}

// File Upload Feature

if ($\_SERVER['REQUEST\_METHOD'] == 'POST' && isset($\_FILES['image'])) {

$uploadDir = "uploads/";

$uploadFile = $uploadDir . basename($\_FILES['image']['name']);

$imageFileType = strtolower(pathinfo($uploadFile, PATHINFO\_EXTENSION));

$allowedTypes = ["jpg", "jpeg", "png", "gif"];

if (!in\_array($imageFileType, $allowedTypes)) {

echo "Only JPG, JPEG, PNG & GIF files are allowed.";

} elseif ($\_FILES['image']['size'] > 2000000) {

echo "File is too large. Max 2MB allowed.";

} else {

if (move\_uploaded\_file($\_FILES['image']['tmp\_name'], $uploadFile)) {

echo "File uploaded successfully.";

} else {

echo "File upload failed.";

}

}

}

?>

<!DOCTYPE html>

<html>

<head>

<title>Login & File Upload</title>

</head>

<body>

<form method="post" action="" enctype="multipart/form-data">

<label>Email:</label>

<input type="email" name="email" value="<?php echo isset($\_COOKIE['user\_email']) ? $\_COOKIE['user\_email'] : ''; ?>" required><br>

<label>Password:</label>

<input type="password" name="password" required><br>

<label>

<input type="checkbox" name="remember\_me"> Remember Me

</label><br>

<button type="submit" name="login">Login</button>

</form>

<a href="?logout=true">Logout</a>

<h2>Upload an Image</h2>

<form method="post" action="" enctype="multipart/form-data">

<input type="file" name="image" required><br>

<button type="submit">Upload</button>

</form>

</body>

</html>

**PHP with MVC Architecture**

**27. Practical Exercise: Build a small blog application using the MVC architecture, where users can create, read, update, and delete posts.**

<?php

session\_start();

// Database Connection: Database.php

class Database {

private $host = "localhost";

private $db\_name = "blog";

private $username = "root";

private $password = "";

private $conn;

public function \_\_construct() {

try {

$this->conn = new PDO("mysql:host={$this->host};dbname={$this->db\_name}", $this->username, $this->password);

$this->conn->setAttribute(PDO::ATTR\_ERRMODE, PDO::ERRMODE\_EXCEPTION);

} catch (PDOException $e) {

die("Connection failed: " . $e->getMessage());

}

}

public function query($sql, $params = []) {

try {

$stmt = $this->conn->prepare($sql);

$stmt->execute($params);

return $stmt;

} catch (PDOException $e) {

die("Query failed: " . $e->getMessage());

}

}

public function \_\_destruct() {

$this->conn = null;

}

}

// Blog Post Model

class Post {

private $db;

public function \_\_construct() {

$this->db = new Database();

}

public function createPost($title, $content) {

$sql = "INSERT INTO posts (title, content) VALUES (?, ?)";

return $this->db->query($sql, [$title, $content]);

}

public function getPosts() {

$sql = "SELECT \* FROM posts ORDER BY created\_at DESC";

return $this->db->query($sql)->fetchAll(PDO::FETCH\_ASSOC);

}

public function getPost($id) {

$sql = "SELECT \* FROM posts WHERE id = ?";

return $this->db->query($sql, [$id])->fetch(PDO::FETCH\_ASSOC);

}

public function updatePost($id, $title, $content) {

$sql = "UPDATE posts SET title = ?, content = ? WHERE id = ?";

return $this->db->query($sql, [$title, $content, $id]);

}

public function deletePost($id) {

$sql = "DELETE FROM posts WHERE id = ?";

return $this->db->query($sql, [$id]);

}

}

// Blog Controller

class BlogController {

private $postModel;

public function \_\_construct() {

$this->postModel = new Post();

}

public function createPost() {

if ($\_SERVER['REQUEST\_METHOD'] === 'POST') {

$title = $\_POST['title'];

$content = $\_POST['content'];

$this->postModel->createPost($title, $content);

header("Location: index.php");

}

}

public function deletePost() {

if (isset($\_GET['delete'])) {

$id = $\_GET['delete'];

$this->postModel->deletePost($id);

header("Location: index.php");

}

}

}

$controller = new BlogController();

$controller->deletePost();

$posts = (new Post())->getPosts();

?>

<!DOCTYPE html>

<html>

<head>

<title>Simple Blog</title>

</head>

<body>

<h1>Simple Blog</h1>

<form method="post" action="">

<label>Title:</label>

<input type="text" name="title" required><br>

<label>Content:</label>

<textarea name="content" required></textarea><br>

<button type="submit" name="create">Create Post</button>

</form>

<hr>

<h2>Posts</h2>

<?php foreach ($posts as $post): ?>

<h3><?php echo htmlspecialchars($post['title']); ?></h3>

<p><?php echo nl2br(htmlspecialchars($post['content'])); ?></p>

<a href="?delete=<?php echo $post['id']; ?>">Delete</a>

<hr>

<?php endforeach; ?>

</body>

</html>