PHP Object-Oriented Programming Exercises

Introduction

These exercises are designed to solidify your understanding of PHP OOP concepts. Complete them in order, as they progressively build upon previous concepts.

Exercise 1: Basic Class and Object Creation

Objective: Understand classes, objects, properties, and methods.

Task:

Create a `Book` class with the following:

- Properties: `title`, `author`, `price`, `isbn`
- Methods:
  - `getDetails()` - returns a formatted string with all book information
  - `applyDiscount($percentage)` - reduces the price by the given percentage
  - `getPrice()` - returns the current price

Create 3 book objects and display their details. Apply a 10% discount to one book and show the updated price.

Exercise 2: Constructor and Destructor

Objective: Learn how to initialize objects and clean up resources.

Task:

Create a `DatabaseConnection` class with:

- Properties: `host`, `username`, `database`, `connected` (boolean)
- Constructor that accepts host, username, and database parameters
- A `connect()` method that sets `connected` to true and prints a connection message
- A destructor that prints a disconnection message

Create multiple database connection objects and observe the constructor and destructor behavior.

Exercise 3: Encapsulation (Getters and Setters)

Objective: Implement proper encapsulation using access modifiers.

Task:

Create a `BankAccount` class with:

- Private properties: `accountNumber`, `balance`, `accountHolder`
- Public methods:
  - `__construct($accountNumber, $accountHolder, $initialBalance)`
  - `deposit($amount)` - add money (validate amount is positive)
  - `withdraw($amount)` - remove money (check sufficient balance)
  - `getBalance()` - return current balance
  - `getAccountInfo()` - return account details (but not the full balance for security)

Create accounts and perform various transactions. Try to access private properties directly and observe what happens.

Exercise 4: Static Properties and Methods

Objective: Understand when and how to use static members.

Task:

Create a `User` class with:

- Private static property: `$userCount` (tracks total users)
- Private properties: `id`, `name`, `email`
- Public static method: `getUserCount()` - returns total number of users created
- Public static method: `create($name, $email)` - factory method to create users
- Constructor that auto-increments `$userCount` and assigns unique IDs

Create several users using the static factory method and display the total count.

Exercise 5: Inheritance

Objective: Learn to extend classes and reuse code.

Task:

Create a base class `Vehicle` with:

- Protected properties: `brand`, `model`, `year`
- Public methods: `__construct()`, `getInfo()`, `start()`

Create two child classes:

1. `Car` extends `Vehicle`
   - Additional property: `numberOfDoors`
   - Override `start()` to include car-specific message
   - Add method: `openTrunk()`
2. `Motorcycle` extends `Vehicle`
   - Additional property: `hasSidecar`
   - Override `start()` to include motorcycle-specific message
   - Add method: `wheelie()`

Create instances of both and demonstrate inheritance.

Exercise 6: Abstract Classes

Objective: Create blueprints for related classes.

Task:

Create an abstract class `Shape` with:

- Abstract methods: `calculateArea()`, `calculatePerimeter()`
- Concrete method: `getShapeType()` that returns the class name

Create three concrete classes:

1. `Circle` - properties: `radius`

2. `Rectangle` - properties: `length`, `width`
3. `Triangle` - properties: `side1`, `side2`, `side3`

Each class should implement the abstract methods. Create objects and calculate areas and perimeters.

Exercise 7: Interfaces

Objective: Learn to define contracts for classes.

Task:

Create interfaces:

1. `Payable` with method: `calculatePayment()`
2. `Identifiable` with methods: `getId()`, `setId($id)`

Create classes that implement these interfaces:

1. `Employee` implements both `Payable` and `Identifiable`
   - Properties: `id`, `name`, `hourlyRate`, `hoursWorked`
   - Implement `calculatePayment()` based on hourly rate
2. `Freelancer` implements both interfaces
   - Properties: `id`, `name`, `projectRate`, `projectsCompleted`
   - Implement `calculatePayment()` based on project rate

Create a function `processPayment(Payable $worker)` that accepts any payable object and displays the payment.

Exercise 8: Traits

Objective: Learn to share methods across unrelated classes.

Task:

Create traits:

1. `Timestampable` with properties: `createdAt`, `updatedAt` and methods to set/get these

2. `SoftDeletable` with property: `deletedAt` and methods: `softDelete()`, `restore()`, `isDeleted()`

Create classes that use these traits:

1. `Post` (uses both traits)
   - Properties: `title`, `content`, `author`
2. `Comment` (uses both traits)
   - Properties: `postId`, `content`, `author`

Demonstrate creating, soft deleting, and restoring objects.

Exercise 9: Method Overloading (Magic Methods)

Objective: Use magic methods for dynamic behavior.

Task:

Create a `Product` class that uses magic methods:

- `__get($property)` - handle reading inaccessible properties
- `__set($property, $value)` - handle writing to inaccessible properties
- `__isset($property)` - handle isset() calls
- `__toString()` - return a string representation
- `__call($method, $arguments)` - handle undefined method calls

Store properties in a private array and use magic methods to access them dynamically.

Exercise 10: Namespaces

Objective: Organize code and avoid naming conflicts.

Task:

Create the following namespace structure:

```
App\
  Models\
    User.php
    Product.php
  Controllers\
    UserController.php
    ProductController.php
  Services\
    EmailService.php
    PaymentService.php
```

Each class should be in its proper namespace. Create an `index.php` file that:

- Uses these classes with proper namespace imports
- Demonstrates both `use` statements and fully qualified names
- Shows aliasing to resolve naming conflicts

Exercise 11: Polymorphism

Objective: Write flexible code that works with different object types.

Task:

Create an interface `NotificationInterface` with method: `send($message)`

Create multiple classes implementing this interface:

1. `EmailNotification`
2. `SMSNotification`
3. `PushNotification`

Create a `NotificationManager` class with:

- Method: `notify($notification, $message)` that accepts any NotificationInterface
- Method: `notifyAll($notifiers, $message)` that sends via multiple channels

Demonstrate polymorphism by sending notifications through different channels using the same interface.

Exercise 12: Dependency Injection

Objective: Learn loose coupling and testable code.

Task:

Create a `Logger` interface with method: `log($message)`

Create two implementations:

1. `FileLogger` - writes to a file
2. `DatabaseLogger` - simulates writing to database

Create a `UserService` class that:

- Accepts a Logger through constructor (dependency injection)
- Has methods: `createUser()`, `updateUser()`, `deleteUser()`
- Uses the injected logger to log actions

Demonstrate using the UserService with different logger implementations.

Exercise 13: Design Pattern - Singleton

Objective: Implement the Singleton pattern.

Task:

Create a `Configuration` class that:

- Implements the Singleton pattern
- Has private constructor
- Has static method: `getInstance()`
- Stores configuration as an array
- Has methods: `get($key)`, `set($key, $value)`, `loadFromFile($filename)`

Demonstrate that only one instance exists by trying to create multiple instances.

Exercise 14: Design Pattern - Factory

Objective: Implement the Factory pattern.

Task:

Create a `PaymentFactory` class that creates different payment processors:

- `CreditCardPayment`
- `PayPalPayment`
- `BankTransferPayment`

All payment classes should implement a `PaymentInterface` with methods:

- `processPayment($amount)`
- `refund($transactionId)`

The factory should have a static method: `createPayment($type)` that returns the appropriate payment processor.

Exercise 15: Real-World Application - E-Commerce System

Objective: Combine all OOP concepts into a practical application.

Task:

Build a mini e-commerce system with the following:

1. **Product Management:**
   - Abstract `Product` class
   - Concrete classes: `PhysicalProduct`, `DigitalProduct`
   - Interface: `Purchasable`
2. **Shopping Cart:**
   - `Cart` class with add/remove items
   - Calculate total with tax
   - Apply discount codes
3. **User System:**
   - Abstract `User` class
   - `Customer` and `Admin` classes

    - Interface: `Authenticatable`

4. **Order Processing:**
    - `Order` class
    - `OrderItem` class
    - Multiple order statuses (use constants or enums)
5. **Payment Processing:**
    - Use the Factory pattern for payment methods
    - Interface for all payment processors

Requirements:

- Use proper encapsulation
- Implement at least 3 interfaces
- Use inheritance appropriately
- Include at least one trait
- Use namespaces
- Implement dependency injection
- Add proper validation
- Include error handling

Bonus Exercise: Build a Simple MVC Framework

Create a basic MVC structure with:

- `Router` class (handles URL routing)
- `Controller` base class
- `Model` base class (with basic CRUD operations)
- `View` class (for rendering templates)
- Implement autoloading using `spl_autoload_register()`

Build a simple blog application using this framework with posts and comments.

Evaluation Criteria

For each exercise, your code should:

1. Follow PSR-12 coding standards
2. Use meaningful variable and method names
3. Include proper error handling
4. Have clear comments explaining complex logic
5. Demonstrate understanding of the concept
6. Be tested with sample data/scenarios

Resources

- PHP Manual: https://www.php.net/manual/en/language.oop5.php
- PSR-12: https://www.php-fig.org/psr/psr-12/
- Design Patterns: Research Gang of Four patterns

Good luck with your exercises!