Web Development with PHP
Practical - 8
OOP with PHP - 2

Encapsulation in PHP

Encapsulation is a fundamental concept that allows the bundling of data and methods within a class, providing a protective barrier around the workings of an object. It enables the control of access to the properties and methods of an object, ensuring that they can only be accessed or modified through predefined interfaces.

Encapsulation promotes data hiding, preventing direct manipulation of object properties from outside the class. By encapsulating data and methods, we can create robust and maintainable code, enhancing code reusability, security, and abstraction.

Encapsulation is a key principle in object-oriented programming (OOP) that promotes the concept of bundling data and methods within a class, providing an encapsulated unit of functionality. Encapsulation allows the hiding of internal implementation details and the protection of data from direct access by external entities. Encapsulation helps in achieving data abstraction, as it allows the definition of public interfaces through which objects can interact with each other. It establishes boundaries and controls the visibility and accessibility of properties and methods within a class.

One of the primary benefits of encapsulation is data protection. By declaring properties as private or protected, they cannot be accessed directly from outside the class. This prevents unauthorized modification or access to the internal state of an object, ensuring data integrity and maintaining the desired behavior of the class.

Encapsulation also facilitates code organization and maintainability. By encapsulating related data and methods within a class, it becomes easier to manage and understand the codebase. Changes to the internal implementation of a class can be made without affecting other parts of the program that interact with the class through its public interface. Additionally, encapsulation enhances code reusability by promoting the concept of black-box reuse.

```php
<?php
class Car {
    private $brand;
    private $model;
    private $year;

    public function __construct($brand, $model, $year) {
        $this->brand = $brand;
        $this->model = $model;
        $this->year = $year;
    }

    public function getBrand() {
        return $this->brand;
```

```
    }

    public function getModel() {
        return $this->model;
    }

    public function getYear() {
        return $this->year;
    }

    public function setYear($year) {
        $this->year = $year;
    }
}

// Create a Car object
$car = new Car("Toyota", "Camry", 2020);

// Access and display the brand, model, and year
echo "Brand: " . $car->getBrand() . "\n";
echo "Model: " . $car->getModel() . "\n";
echo "Year: " . $car->getYear() . "\n";

// Update the year using the setYear() method
$car->setYear(2022);

// Display the updated year
echo "Updated Year: " . $car->getYear() . "\n";
?>
```

Here, we have a Car class that encapsulates the brand, model, and year of a car. The properties $brand, $model, and $year are declared as private, making them accessible only within the class itself. To access these private properties, we provide public getter methods (getBrand(), getModel(), getYear()) that return their respective values. We also provide a public setter method (setYear()) to update the year property.

The encapsulation in this example ensures that the internal state of the Car object is protected, and the properties can only be accessed or modified through the defined public interface of getter and setter methods. Run the above code in your editor for a better and clear explanation.

Advantages of Encapsulation

- Data Hiding:

  Encapsulation hides the internal details of a class, including its properties and implementation logic, from the outside world. This protects the integrity of the data and prevents direct manipulation by external entities. Only the defined public methods can access and modify the internal state, ensuring controlled and secure data manipulation.

- Code Organization and Maintainability:

  Encapsulation promotes a modular approach to code development. By encapsulating related properties and methods within a class, you create a self-contained unit that is easier to understand, modify, and maintain. Encapsulated classes can be individually worked on and tested without impacting the rest of the codebase, leading to better code organization and improved maintainability.

- Code Reusability:

  Encapsulated classes can be reused in different parts of your application or even in other projects. By providing a public interface to interact with the class, you create a black box that can be used without needing to know its internal implementation. This promotes code reusability, reduces code duplication, and speeds up development.

- Abstraction:

  Encapsulation helps in achieving abstraction by hiding complex implementation details and providing a simplified interface. It allows you to focus on the essential aspects of a class and ignore unnecessary complexities. This abstraction enhances code readability and comprehension, making it easier to work with and understand the code.

- Security and Validation:

  With encapsulation, you can enforce data validation and security checks within the class itself. By controlling access to properties through getter and setter methods, you can validate and sanitize input values, ensuring that only valid data is stored or retrieved. This helps in preventing data corruption, enforcing business rules, and enhancing the security of your application.

- Flexibility and Extensibility:

  Encapsulation provides a flexible foundation for extending and modifying your code. By encapsulating behavior and data within classes, you can easily add new features, modify existing functionality, or extend the behavior of a class without affecting other parts of the codebase. This promotes code flexibility, scalability, and adaptability to changing requirements.

Disadvantages of Encapsulation

Encapsulation is a fundamental principle in object-oriented programming (OOP) that promotes the concept of data hiding and encapsulating data and methods within classes. While encapsulation offers numerous benefits, it is important to be aware of some potential disadvantages:

- Increased Complexity:

  Implementing encapsulation can introduce additional complexity to your codebase. By encapsulating data and behavior within classes, you need to design and manage the class structure effectively. This can lead to more intricate code, making it harder to understand and maintain, especially for developers who are new to the codebase.

- Overhead in Performance:

  Encapsulation often involves accessing data through getter and setter methods rather than directly manipulating the data. This can result in a slight performance overhead due to the additional method calls. While the impact is usually negligible, in performance-critical scenarios, direct access to data might be more efficient.

- Dependency on Class Structure:

  Encapsulation tightly binds data and behavior within classes. This can introduce dependencies between classes, where changes in one class can affect other classes that rely on its encapsulated elements. This tight coupling can make it harder to modify or extend the codebase without affecting other parts of the application.

- Limited Accessibility:

  By encapsulating data within classes, you limit its accessibility from external code. While this is often a desired effect to enforce data integrity and encapsulation, it can sometimes make it more difficult to access or modify the data when necessary. Properly defining and managing access levels (public, protected, private) becomes crucial to strike the right balance between encapsulation and data usability.

- Potential for Code Duplication:

  Encapsulation can lead to code duplication if similar behavior or data needs to be encapsulated in multiple classes. This can arise when classes have overlapping responsibilities or share common functionality. Ensuring proper code organization and reusability is important to mitigate the risk of code duplication.

- Testing Challenges:

  Encapsulated code can present challenges in unit testing. Since encapsulated elements are not directly accessible outside the class, it may require using getter and setter methods or

employing reflection techniques to access and test the encapsulated data. This additional complexity can make testing more difficult and increase the effort required to write effective tests.

Despite these potential disadvantages, encapsulation remains a crucial principle in OOP and is widely used for creating modular, maintainable, and scalable code. It is important to carefully design and implement encapsulation, considering the specific needs and constraints of your project, to ensure a good balance between encapsulation and other design principles such as code simplicity and extensibility.

Types of Inheritance

Inheritance is a core concept in object-oriented programming (OOP) that allows classes to inherit properties and methods from other classes. In PHP, there are several types of inheritance that developers can use to create class hierarchies and achieve code reusability.

Single Inheritance:

Single inheritance refers to a scenario where a class inherits from a single parent class. In PHP, single inheritance is the most commonly used type of inheritance. It allows a class to inherit the properties and methods of one parent class, also known as the superclass or base class.

```
class ChildClass extends ParentClass {
   // Class definition
}
```

Example

```
class Animal {
   public function sound() {
      echo "The animal makes a sound.";
   }
}

class Dog extends Animal {
   // Additional class definition for Dog
}
```

Multilevel Inheritance:

Multilevel inheritance involves a class inheriting from a parent class, and that parent class itself inheriting from another parent class. This forms a hierarchy or chain of classes. In PHP, multilevel inheritance allows classes to inherit properties and methods from multiple levels up the class hierarchy.

```
class ChildClass extends ParentClass {
   // Class definition
}
```

Example:

```
class Animal {
   public function sound() {
      echo "The animal makes a sound.";
   }
}

class Mammal extends Animal {
   // Additional class definition for Mammal
}

class Dog extends Mammal {
   // Additional class definition for Dog
}
```

Hierarchical Inheritance:

Hierarchical inheritance in php only occurs when multiple classes inherit from a single parent class. In this type of inheritance, one parent class serves as the base class, and multiple child classes derive from it.

```
class ChildClass1 extends ParentClass {
   // Class definition
}

class ChildClass2 extends ParentClass {
   // Class definition
}
```

Example

```
class Shape {
   public function area() {
      // Code to calculate the area
   }
}

class Circle extends Shape {
   // Additional class definition for Circle
}

class Rectangle extends Shape {
   // Additional class definition for Rectangle
}
```

Interfaces

Interface is a collection of abstract methods that define a set of behaviors that a class should implement. An interface provides a blueprint for classes to follow, allowing for greater code flexibility and reusability.

```
interface Shape {
  public function getArea();
  public function getPerimeter();
}

class Circle implements Shape {
  private $radius;

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

  public function getArea() {
    return pi() * pow($this->radius, 2);
  }

  public function getPerimeter() {
    return 2 * pi() * $this->radius;
  }
}

class Rectangle implements Shape {
  private $width;
  private $height;

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

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

  public function getPerimeter() {
    return 2 * ($this->width + $this->height);
  }
}

// Create objects of the Circle and Rectangle classes
```

```
$circle = new Circle(5);
$rectangle = new Rectangle(3, 4);

// Call the getArea() and getPerimeter() methods on the
objects
echo "Circle area: " . $circle->getArea() . "\n";
echo "Circle perimeter: " . $circle->getPerimeter() . "\n";
echo "Rectangle area: " . $rectangle->getArea() . "\n";
echo "Rectangle perimeter: " . $rectangle->getPerimeter() .
"\n";
```

In this example, we define an interface called "Shape" with two abstract methods: getArea() and getPerimeter(). We then define two classes, Circle and "Rectangle", that implement the "Shape" interface and provide their own implementations of the getArea() and getPerimeter() methods.

Abstract Classes

An abstract class is a class that cannot be instantiated directly, but instead must be extended by a subclass. An abstract class serves as a template for other classes to inherit from, providing common functionality that can be reused across multiple classes. To define an abstract class in PHP, you use the "abstract" keyword in the class declaration.

```
abstract class Shape {
  protected $color;

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

  abstract public function getArea();
}

class Square extends Shape {
  protected $length;

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

  public function getArea() {
    return pow($this->length, 2);
```

```
  }
}

// Create an object of the Square class
$square = new Square("red", 5);

// Call the getArea() method on the object
echo $square->getArea(); // Output: 25
```

In this example, we define an abstract class called "Shape" with a protected property "color" and an abstract method getArea(). The getArea() method is not implemented in the abstract class, but is instead defined in a subclass.

Static Keyword

The "static" keyword is used to declare class properties and methods that can be accessed without creating an object of the class. Instead, they are accessed directly from the class itself.

```
class MyClass {
  public static $myProperty = "Hello, world!";

  public static function myMethod() {
    echo "This is a static method!";
  }
}

// Access the static property
echo MyClass::$myProperty; // Output: Hello, world!

// Call the static method
MyClass::myMethod(); // Output: This is a static method!
```

In this example, we define a class called "MyClass" with a static property called "$myProperty" and a static method called "myMethod()". We can access the static property by using the class name followed by the scope resolution operator "::".

Static properties and methods are useful in scenarios where you need to access shared data or functionality across multiple instances of the class. They are also commonly used in utility

classes that don't require any state and simply provide a set of functions or constants.

Final Keyword

The final keyword is used to restrict the behavior of a class, method, or property. When a class, method, or property is marked as final, it cannot be extended or overridden by any child class. The final keyword can be applied to a class declaration, a method declaration, or a property declaration.

Final class declaration:

```
final class MyClass {
  // Class implementation
}
```

Here the MyClass class is marked as final, which means that no other class can extend it.

Final method declaration:

```
class MyClass {
  final public function myMethod() {
    // Method implementation
  }
}
```

Here, the myMethod() method is marked as final, which means that no child class can override it.

Final property declaration:

```
class MyClass {
  final public $myProperty = 'foo';
}
```

Here, the $myProperty property is marked as final, which means that no child class can redeclare or modify its value.

The final keyword can be useful in situations where you want to ensure that a certain class, method, or property is not modified or extended in unintended ways. However, it should be used sparingly, as it can limit the flexibility and extensibility of your code.

self::, parent:: and static::

self::, parent:: and static:: allow you to access static members and const (public and static) WITHIN the class definition.

- self:: refer to the current class.
- parent:: refer to the superclass and
- static:: refer to the current class but with late binding

self vs. $this

Use $this to refer to the current object. Use self to refer to the current class. In other words, within a class definition, use $this->member to reference non-static members, use self::$member for static members.

Polymorphism

Polymorphism refers to the ability of objects to take on different forms and exhibit different behaviors while sharing a common interface. It allows different classes to implement the same method name with different functionality. This concept enhances code reusability and flexibility by allowing objects of different types to be treated interchangeably. Polymorphism enables the use of inheritance and interfaces, facilitating the creation of more modular and extensible code. It simplifies the process of adding new classes and functionality to existing codebases without breaking the existing code.

Polymorphism is achieved through method overriding and method overloading. Method overriding allows a child class to provide its own implementation of a method that is already defined in its parent class. This means that objects of different classes can respond differently to the same method call, based on their specific implementations.

Polymorphism is particularly useful when working with class hierarchies and interfaces. It allows you to write code that operates on a superclass or interface, and it can seamlessly work with any derived class that adheres to that superclass or implements that interface. This promotes code reuse and simplifies the management of related classes, as you can treat them uniformly through their common superclass or interface.

Example:

```
// Parent class
class Animal {
    public function makeSound() {
        // Default implementation for making sound
        echo "The animal makes a sound.\n";
    }
}

// Child classes inheriting from Animal
class Dog extends Animal {
    public function makeSound() {
        // Implementation specific to dogs
        echo "The dog barks.\n";
    }
}

class Cat extends Animal {
    public function makeSound() {
        // Implementation specific to cats
        echo "The cat meows.\n";
    }
}

// Interface
interface CanFly {
```

```
    public function fly();
}

// Class implementing the CanFly interface
class Bird extends Animal implements CanFly {
    public function makeSound() {
        // Implementation specific to birds
        echo "The bird chirps.\n";
    }

    public function fly() {
         // Implementation of the fly method from the CanFly
interface
        echo "The bird is flying.\n";
    }
}

// Usage
$animal = new Animal();
$animal->makeSound();  // Output: The animal makes a sound.

$dog = new Dog();
$dog->makeSound();  // Output: The dog barks.

$cat = new Cat();
$cat->makeSound();  // Output: The cat meows.

$bird = new Bird();
$bird->makeSound();  // Output: The bird chirps.
$bird->fly();  // Output: The bird is flying.
```

Example: Polymorphism with Interfaces

```
interface Shape {
    public function calculateArea();
}

class Circle implements Shape {
    private $radius;

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

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

class Rectangle implements Shape {
    private $length;
    private $width;

    public function __construct($length, $width) {
        $this->length = $length;
        $this->width = $width;
    }

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

$circle = new Circle(5);
echo $circle->calculateArea();   // Output: 78.539816339745

$rectangle = new Rectangle(4, 6);
echo $rectangle->calculateArea();   // Output: 24
```

Example: Polymorphism with Abstract Classes

```
abstract class Vehicle {
    abstract public function start();
    abstract public function stop();
}

class Car extends Vehicle {
    public function start() {
        echo "Car started.\n";
    }

    public function stop() {
        echo "Car stopped.\n";
    }
}

class Bike extends Vehicle {
    public function start() {
        echo "Bike started.\n";
    }
```

```
    public function stop() {
        echo "Bike stopped.\n";
    }
}

$car = new Car();
$car->start();    // Output: Car started.
$car->stop();     // Output: Car stopped.

$bike = new Bike();
$bike->start();   // Output: Bike started.
$bike->stop();    // Output: Bike stopped.
```

Note:

In PHP, to see the contents of the class, use var_dump(). The var_dump() function is used to display the structured information (type and value) about one or more variables.

Exercise:

1. Create a class Course having property coursename, no_of_year with a constructor, and display() method for the display course information. Create a class named Student which inherits a Course having stud_id, stud_name, array of marks(3 subject). Class contains the constructor and is calling the parent constructor. Class contains method caltotal() which returns the total of 3 subject marks. It contains display method which shows all the information about the students: Name,id,marks,total,course name,no_of_year
2. Create a class named shape having a method area() which calculates the area of the shape and returns the area.
    a. Create a class circle which inherits Shape and overrides the area() method and returns the area of the circle.
    b. Create a class square which inherits the Shape class and have an area() method to return the area of the square.
    c. Create a class Rectangle which inherits the Shape class and have an area() method which returns the area of the Rectangle.
    d. Create the object of class shape ,circle,square and rectangle class and display area of each. [override the area() method]
3. Create an abstract class Employee having a constructor for setting name, year of joining, date of birth and department of employee. Create an appropriate method to display the details of the Employee in well designed HTML format. Create an abstract method calculate_salary() in Employee class.
    a. Create a class Manager inherits the Employee class. It should include properties like basic salary, DA ,tax amount, HRA etc. property. Create the constructor for setting the values.
    Implement the calculate_salary() (basic+DA+HRA – tax amount).
    b. Create a class worker which inherits the Employee class. Create a constructor which sets the property wages per hour,worked hour.
    Implement calsal() method (wages per hour*worked hour) .
    c. Create appropriate methods in each class to display the well formatted details in HTML.
    d. Write a php program which create multiple object of Manager and Worker class, and display the name, designation and salary of each
4. Create a class Item with
    Property : Item name,Item no
    Method : display -> display item name and ino

    a. Create a class Category which inherits Item
    Property : category name[e.g cloth/electronics/kids toys.. etc] , subcategory[eg mobile, laptop, jeans,t shirt] , price
    Method: displayItem() to display category, subcategory and price and getprice() to return the price of item

    b. Create a class purchase which inherits Item
    Property : purchase id, total amount , quantity
    Method :
        - calculate_order_amout() : calculate and return the total amount

(qty*price)

- display_purchase() : display the all details of Item and category.

c. Write a php script which creates the necessary constructor and creates an object of purchase class. Calculate the total order amount. It should also display the details purchase