Introduction to Object-Oriented Programming (OOP)

Object-Oriented Programming (OOP) is a programming paradigm that uses "objects" to design applications and computer programs. These objects represent real-world entities and can contain data (attributes) and methods (functions).

Key Concepts of OOP

1. **Class**: A blueprint for creating objects. It defines a set of attributes and methods that the created objects will have.
2. **Object**: An instance of a class. It is created using the class blueprint and can have its own unique data.
3. **Attributes**: Variables that belong to an object. They describe the object's state.
4. **Methods**: Functions that belong to an object. They define the behavior of the object.
5. **Inheritance**: A way to form new classes using classes that have already been defined. It helps to reuse and enhance existing code.
6. **Encapsulation**: The bundling of data and methods that operate on the data within one unit, e.g., a class. It restricts direct access to some of the object's components.
7. **Polymorphism**: The ability to present the same interface for different data types. It allows methods to be used interchangeably.

OOP Use Case

- **Creating Reusable Components**: Design reusable classes for common functionalities like logging, data access, or UI elements.
- **Modeling Real-World Entities**: Represent real-world entities like customers, products, and orders in an e-commerce application with classes.
- **Encapsulation**: Encapsulate data and related methods within classes to hide implementation details and expose a clear interface.
- **Inheritance for Reusability and Extensibility**: Create a base class with common functionality and extend it in derived classes for specialized behavior.
- **Design Patterns Implementation**: Implement common design patterns (Singleton, Factory, Observer) to solve recurring design problems.
- **APIs and Web Services**: Develop APIs and web services with classes to manage requests, responses, and data handling.
- **Database Interaction**: Use classes to represent database tables and manage CRUD operations through Object-Relational Mapping (ORM) frameworks.
- **Data Structures**: Implement custom data structures (linked lists, trees, graphs) using classes and objects.
- **Testing and Mocking**: Use classes to create mock objects for unit testing, isolating the code being tested from its dependencies.
- **Security and Access Control**: Implement security features like authentication and authorization using classes to manage user roles and permissions.
- **File and Resource Management**: Manage file operations, network connections, and other resources with classes that ensure proper resource handling and cleanup.
- **Business Logic Implementation**: Encapsulate complex business rules and logic within classes to maintain clarity and separation from other parts of the application.

Class & Object

- Class allow developers to encapsulate related data and functions into a single entity.
- Making it easier to manage and extend code
- An object is an instance of a class. You create an object of a class using the new keyword.

```
class Car {
    public $color = "red";
    public function drive() {
        echo "Car is driving!";
    }
}

$myCar = new Car();
echo $myCar->color;
$myCar->drive();
```

Accessing Class Properties Inside Class

- Access class properties inside a class using the `$this` keyword.
- The `$this` keyword refers to the current instance of the class, allowing you to access properties and methods from within the class.

```
class Car {
    public $color = "red";
    public function drive() {
        echo "The " . $this->color . " car is driving!";
    }
}

$myCar = new Car();
echo $myCar->color;
$myCar->drive();
```

Constructor

- Method that gets executed whenever an object is instantiated from a class
- The constructor method has a magic name: `__construct`

```
class Car {
    public function __construct() {
        $num1=10;
        $num2=20;
        echo $num1+$num2;
    }
}
$myCar = new Car();
```

Constructor Parameters

- Pass parameters to the constructor just like you would with any other function or method.
- Constructor can assign value to class properties

```
class Car {
    public function __construct($num1,$num2) {
        echo $num1+$num2;
    }
}
$myCar = new Car(2,3);
```

Set Class variables value using constructor parameters

```
class Car {
    public $num1;
    public $num2;
    public function __construct($num1,$num2) {
        $this->num1 = $num1;
        $this->num2 = $num2;
    }
    function AddTwoNum(){
        echo  $this->num1+$this->num2;
    }
}
$myCar = new Car(2,3);
$myCar->AddTwoNum();
```

Static Properties

- Static properties are tied to the class, not an instance of the class.
- They can be accessed without creating an instance of the class.

```
class MyClass {
    public static $staticProperty = "Static Property";
}

echo MyClass::$staticProperty;  // Outputs: Static Property
```

Static Methods

- Just like static properties, static methods are accessed without creating an instance of the class
- They are often used as utility functions that do not rely on any instance-specific data

```
class MyClass {
    public static function staticMethod() {
        echo "Static Method";
    }
}

MyClass::staticMethod();
```

Accessing Static Properties Inside Class Methods

- Within class methods, static properties and methods are accessed using the self keyword followed by the scope resolution operator

```
class MyClass {
    public static $value = "Static Value";
    public static function showValue() {
        echo self::$value;
    }
}
MyClass::showValue();
```

Inheritance

- Inheritance sets up a "like parent, like child" relationship between classes.
- Instead of rewriting code, the child class can reuse or change what it gets from the parent.
- One class (the child) can use everything from another class (the parent).

```
class Father {
    public function print100() {
       for($i=0;$i<=100;$i++){
        echo "$i <br/>";
       }
    }
}

class Son extends Father {

}

$SonObject = new Son();
$SonObject->print100();
```

Parent Keyword

- You can call the parent class's method using the parent keyword.

```
class Father {
    public function print100() {
       for($i=0;$i<=100;$i++){
        echo "$i <br/>";
       }
    }
}

class Son extends Father {
    public function CallFromFather() {
        parent::print100();
    }
}

$SonObject = new Son();
$SonObject->CallFromFather();
```

Overriding Methods

- Subclasses can override inherited methods from the superclass.

```
class Father {
    public function print100() {
       for($i=0;$i<=100;$i++){
        echo "$i <br/>";
       }
    }
}


class Son extends Father {
    public function print100() {
        for($i=0;$i<=80;$i++){
         echo "$i <br/>";
        }
     }
}

$SonObject = new Son();
$SonObject->print100();
```

Final Keyword

- If you declare a class as final, it means it cannot be extended (inherited).
- If you declare a method as final, it means it cannot be overridden by a subclass.

```
final class Father {
    final  public function print100() {
       for($i=0;$i<=100;$i++){
        echo "$i <br/>";
       }
    }
}

class Son extends Father {
    public function print100() {
        for($i=0;$i<=80;$i++){
         echo "$i <br/>";
        }
     }
}
```

Abstract Classes

- Abstract classes cannot be instantiated on their own but can be subclassed

```
abstract class Father {
    public function print100() {
       for($i=0;$i<=100;$i++){
        echo "$i <br/>";
       }
    }
}

class Son extends Father {

}

$SonObject = new Son();
$SonObject->print100();
```

Constructors and Inheritance

- If a child class has its own constructor, the parent class's constructor will not be automatically called.
- Use `parent::__construct()` if you want to explicitly call the base class's constructor.

```
class Father {
    public function __construct() {
        echo "Father constructor";
    }
}


class Son extends Father {
    public function __construct() {
        parent::__construct();
        echo " and Son constructor";
    }
}

$newObj = new Son();
```

Inheritance and Static Properties

- Static properties belong to the class rather than an instance of the class.
- They can be accessed without creating an instance of the class using `ClassName::$propertyName`.
- **Static Properties and Methods**: Declared using the `static` keyword and are shared among all instances of the class.
- **Inheritance of Static Properties**: A child class can access static properties and methods from the parent class using `parent::`.
- **Accessing Static Properties**: You do not need to instantiate an object to access static properties or methods; they are accessed using the `::` operator.

```
class ParentClass {
    public static $familyName = "Smith";

    public static function getFamilyName() {
        return self::$familyName;
    }
}

class ChildClass extends ParentClass {
    public static function getFamilyDetails() {
        // Accessing static property from parent class
        return "Family Name: " . parent::$familyName;
    }
}
```

```
// Accessing static property without instantiation
echo ParentClass::$familyName;

// Accessing static method without instantiation
echo ParentClass::getFamilyName();

// Accessing static method in the child class
echo ChildClass::getFamilyDetails();
```

Access modifiers

Access modifiers control the visibility of class properties and methods

- **public** – accessible everywhere
- **protected** – accessible within the class and its subclasses (inheritance)
- **private** – accessible only within the class itself

```
class Fruit {
    public $color; // Can be accessed anywhere
    protected $taste; // Can be accessed within this class and derived classes
    private $origin;  // Can be accessed only within this class


    public function setTaste($taste) {
        $this->taste = $taste;
    }

    public function setOrigin($origin) {
        $this->origin = $origin;
    }

    public function describe() {
        echo "This fruit is " . $this->color . " and tastes " . $this->taste . " from " . $this-
>origin . ".\n";
    }
}



class Apple extends Fruit {
    public function revealTaste() {
        return $this->taste;  // Allowed because $taste is protected
    }

     public function revealOrigin() {

       return $this->origin;

     }

}


$apple = new Apple();
$apple->color = "red";
$apple->setTaste("sweet");
$apple->setOrigin("Washington");
$apple->describe();  // This fruit is red and tastes sweet from Washington.
echo $apple->revealTaste();  // Outputs: sweet
echo $apple->revealOrigin();
```

Interfaces:

- Interfaces only declare method signatures; they do not contain method bodies.
- A class can implement multiple interfaces.
- Methods defined in an interface must be **public**.
- Interfaces cannot contain properties, only method declarations.

Syntax of an Interface

```
interface MyInterface {
    public function method1();
    public function method2($param);
}
```

Implementing an Interface

When a class implements an interface, it must define all the methods declared in the interface.

```
interface MyInterface {
    public function method1();
    public function method2($param);
}

class MyClass implements MyInterface {
    public function method1() {
        echo "Method1 implemented.";
    }

    public function method2($param) {
        echo "Method2 implemented with param: $param";
    }
}

// Usage
$obj = new MyClass();
$obj->method1(); // Output: Method1 implemented.
$obj->method2("Test"); // Output: Method2 implemented with param: Test
```

Multiple Interfaces

A class can implement multiple interfaces, allowing for more flexibility in design.

```
interface Interface1 {
    public function method1();
}

interface Interface2 {
    public function method2();
}
```

```
class MyClass implements Interface1, Interface2 {
    public function method1() {
        echo "Method1 from Interface1.";
    }

    public function method2() {
        echo "Method2 from Interface2.";
    }
}

// Usage
$obj = new MyClass();
$obj->method1(); // Output: Method1 from Interface1.
$obj->method2(); // Output: Method2 from Interface2.
```

Method Overloading in PHP

- Method overloading in PHP is achieved using the `__call()` or `__callStatic()` magic methods.
- These methods allow you to handle calls to methods that are not explicitly defined in the class.

```
class MethodOverloadingExample {
    public function __call($name, $arguments) {
        // Handle undefined method calls
        echo "Calling instance method '$name' with arguments: " . implode(", ", $arguments) .
"\n";
    }

    public static function __callStatic($name, $arguments) {
        // Handle undefined static method calls
        echo "Calling static method '$name' with arguments: " . implode(", ", $arguments) . "\n";
    }
}

$example = new MethodOverloadingExample();
$example->undefinedMethod("arg1", "arg2"); // Output: Calling instance method 'undefinedMethod'
with arguments: arg1, arg2

MethodOverloadingExample::undefinedStaticMethod("arg1", "arg2"); // Output: Calling static method
'undefinedStaticMethod' with arguments: arg1, arg2
```

---

3. Encapsulation

- **Encapsulation** is the concept of bundling data (properties) and methods that operate on that data into a single unit (class).
- It helps restrict direct access to some of an object's components to protect the integrity of the object.
- Use `private`, `protected`, and `public` access modifiers to control the visibility of properties and methods.

```
class EncapsulationExample {
    private $data = "Confidential";

    public function getData() {
        return $this->data;
    }
}
```

```
$example = new EncapsulationExample();
echo $example->getData(); // Output: Confidential
```

4. Polymorphism

- **Polymorphism** allows different classes to be treated as instances of the same class through inheritance.
- This is usually achieved through method overriding, where a child class defines a specific implementation of a method that is already defined in the parent class.

```
class Animal {
    public function sound() {
        echo "Some generic animal sound\n";
    }
}

class Dog extends Animal {
    public function sound() {
        echo "Bark\n";
    }
}

$animal = new Animal();
$animal->sound(); // Output: Some generic animal sound

$dog = new Dog();
$dog->sound(); // Output: Bark
```

#php