OOP Tutorial

PHP is a server-side scripting language, mainly used for web development but also used as a general-purpose programming language. Object-Oriented Programming (PHP OOP),  is a type of programming language principle added to php5, that helps in building complex, reusable web applications.

In this blog, we will be explaining some of the Object-Oriented Programming concepts in PHP with some examples.

The  Object Oriented concepts in PHP are:

* Class
* Objects
* Inheritance
* Interface
* Abstraction
* Magic Methods

**Class  & Object:**

* Class is a programmer-defined data type, which includes local methods and local variables.
* Class is a collection of objects. Object has properties and behaviour.
* First we have to define a php class, where classname should be same as filename.
* You can use include/include\_once or require/require\_once

require\_once('class.php');

Class vs Object

Class is like your house blueprint. Before your house is constructed, there is a house blueprint. It is not an actual house, but a plan how this house will look like, how many rooms it will have and so on. Then the house will be constructed by following the blueprint exactly. In this analogy, the house blueprint is a class and your actual house is an object. We can have unlimited objects of a class, just like we can build unlimited exact houses by following the same house blueprint.

A few key points to keep in mind:

* Class is generic, whereas Object is specific
* Class defines properties/functions of an Object
* Object is an instance of a Class
* You can instantiate an object, but not a Class

## PHP Class

Class is consist of properties and methods. Below is a PHP class. In this simple class. **$postCode** is a property and **ringBell()** is a method. They are all prefixed with a visibility keyword (public).

Class House {

public $postCode = “560121”;

public function ringBell() {

echo “Ding Dang Dong”;

}

}

To instantiate an object of a class, use the keyword new as below:

$house = new House();

## Example:

|  |
| --- |
| <?php |
|  | class Books{ |
|  | public function name(){ |
|  | echo “Drupal book”; |
|  | } |
|  | public function price(){ |
|  | echo “900 Rs/-”; |
|  | } |
|  | } |
|  | To create php object we have to use a  new operator. Here php object is the object of the Books Class. |
|  | $obj = new Books(); |
|  | $obj->name(); |
|  | $obj->price(); |
|  | ?> |

## Example:

|  |
| --- |
| <?php |
|  | class Mobile { |
|  | /\* Member variables \*/ |
|  | var $price; |
|  | var $title; |
|  | /\* Member functions \*/ |
|  | function setPrice($par){ |
|  | $this->price = $par; |
|  | } |
|  | function getPrice(){ |
|  | echo $this->price ."<br/>"; |
|  | } |
|  | function setName($par){ |
|  | $this->title = $par; |
|  | } |
|  | function getName(){ |
|  | echo $this->title ." <br/>"; |
|  | } |
|  | } |
|  | $Samsung = new Mobile(); |
|  | $Xiaomi = new Mobile(); |
|  | $Iphone = new Mobile(); |
|  | $Samsung->setName( "SamsungS8 ); |
|  | $Iphone->setName( "Iphone7s" ); |
|  | $Xiaomi->setName( "MI4" ); |
|  | $Samsung->setPrice( 90000 ); |
|  | $Iphone->setPrice( 65000 ); |
|  | $Xiaomi->setPrice( 15000 ); |
|  | Now you call another member functions to get the values set by in above example |
|  | $Samsung->getName(); |
|  | $Iphone->getName(); |
|  | $Xiaomi->getName(); |
|  | $Samsung->getPrice(); |
|  | $Iphone->getPrice(); |
|  | $Xiaomi->getPrice(); |
|  | ?> |

## Visibility

Each method and property has its visibility. There are three types of visibility in PHP. They are declared by keywords *public*, *protected* and *private*. Each one of them controls how a method/property can be accessed by outsiders.

**Public**: It allows anyone from outside access its method/property. This is the default visibility in PHP class when no keywords are prefixed to a method/property.

**Protected**: It only allows itself or children classes to access its method/property.

**Private**: It does not allow anyone except itself to access its method/property.

## Inheritance

It lets subclass inherits characteristics of the parent class. Parent class decides what and how its properties/methods to be inherited by declared visibility.

**Inheritance**

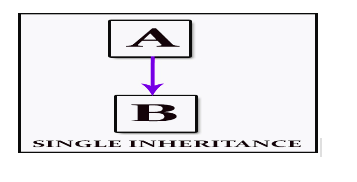
When the properties and the methods of the parent class are accessed by the child class, we call the concept has inheritance. The child class can inherit the parent method and give own method implementation, this property is called overridden method. When the same method of the parent class is inherited we call as inherited method. Now let us see types of inheritance supported in Object Oriented Programming and corresponding Php inheritance examples.

**Types Of Inheritance**

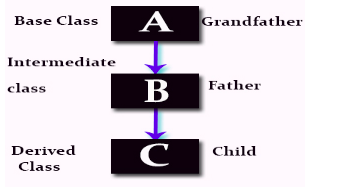
1. Single Level Inheritance
2. Multilevel Inheritance

**Single Level Inheritance:**  In Single Level Inheritance the Parent class methods will be extended by the child class. All the methods can be inherited.

**Single Level Inheritance**



**MultiLevel Inheritance :**  In MultiLevel Inheritance, the parent class method will be inherited by child class and again subclass will inherit the child class method.



class Shape {

public function name() {

echo "I am a shape";

}

}

class Circle extends Shape {

}

$circle = new Circle();

$circle->name(); // I am a shape

The key word here is **extends**. When **Circle** class extends from **Shape**class, it inherits all of the public and protected methods as well as properties from **Shape** class.

## Example:

|  |
| --- |
| <?php |
|  | class A { |
|  | public function myage() { |
|  | return ' age is 80'; |
|  | } |
|  | } |
|  | class B extends A { |
|  | public function mysonage() { |
|  | return ' age is 50'; |
|  | } |
|  | } |
|  | class C extends B { |
|  | public function mygrandsonage() { |
|  | return 'age is 20'; |
|  | } |
|  | public function myHistory() { |
|  | echo "Class A " .$this->myage(); |
|  | echo "Class B ".$this-> mysonage(); |
|  | echo "Class C " . $this->mygrandsonage(); |
|  | } |
|  | } |
|  | $obj = new C(); |
|  | $obj->myHistory(); |
|  | ?> |

## Polymorphism

The provision of a single interface to entities of different types. Basically it means PHP is able to process objects differently depending on their data type or class. This powerful feature allows you to write interchangeable objects that sharing the same interface.

interface Shape {

public function name();

}

class Circle implements Shape {

public function name() {

echo "I am a circle";

}

}

class Triangle implements Shape {

public function name() {

echo "I am a triangle";

}

}

function test(Shape $shape) {

$shape->name();

}

test(new Circle()); // I am a circle

test(new Triangle()); // I am a triangle

Above example, **test(Shape $shape)** function declares(type hints) its only parameter to be **Shape** type. This function is not aware of Circle and Triangle classes. When either class is passed to this function as a parameter, it processes respectively.

## Encapsulation

Encapsulation is used to hide the values or state of a structured data object inside a class, preventing unauthorized parties' direct access to them. It is a concept that motivates us to think through a method/class responsibility and hide its internal implementation/details accordingly. This will make it easy to modify the internal code in a long run without affecting other part of the system. Visibility is the mechanism for encapsulation.

class Person {

private $name;

public function setName($name) {

$this->name = $name;

}

public function getName($name) {

return $this->name;

}

}

$robin = new Person();

$robin->setName('Robin');

$robin->getName();

In this simple class above. Field **$name** is encapsulated (private). Users of the class is not aware how **$name** is stored in **Person** class. Right now the **$name** is stored in memory. We can modify internal code of **Person** class to store it to a flat file or event a database. Users of the class will not need to change any code, in fact they do not even know how **$name** is stored, because that is encapsulated and hided from them.

## Abstraction

Abstraction is the concept of moving the focus from the details and concrete implementation of things, to the types of things (i.e. classes), the operations available (i.e. methods), etc, thus making the programming simpler, more general, and more abstract. It is like a generalization instead of a specification.

class TV {

private $isOn = false;

public function turnOn() {

$this->isOn = true;

}

public function turnOff() {

$this->isOn = false;

}

}

$tv = new TV();

$tv->turnOn();

$tv->turnOff();

Code above defined an **TV** class. We can not do much with it except turning it on and off. The class **TV** is an abstraction of a real **TV** in a very simple use case.

## Interface vs Abstract class

### Interface

Interface declares what methods a class must have without having to implement them. Any class that implements the interface will have to implement details of those declared methods. Interface is not a class, so you can not instantiate an interface. It is useful when you need to enforce some classes to do something.

interface Vehicle {

public function startEngine();

}

class Car implements Vehicle {

public function startEngine() {

echo "Engine Started";

}

}

**Vehicle** is an interface with a declared method **startEngine()**. **Car**implements **Vechicle**, so it has to implement what **startEngine()** method does.

### Abstract class

Abstract class is able to enforce subclasses to implement methods similar to interface. When a method is declared as **abstract** in an abstract class, its derived classes must implement that method.

However it is very different from interface. You can have normal properties and methods as a normal class, because it is in fact a class, so it can be instantiated as a normal class.

abstract class Vehicle {

abstract public function startEngine();

public function stopEngine() {

echo "Engine stoped";

}

}

class Car extends Vehicle {

public function startEngine() {

echo "Engine Started";

}

}

**Vehicle** is an abstract class. **Car** extends **Vechicle**, so it has to implement what **startEngine()** method does, because this method is declared as abstract. However **Car** does not have to anything with method **stopEngine()**, it is inherited as a normal class does.