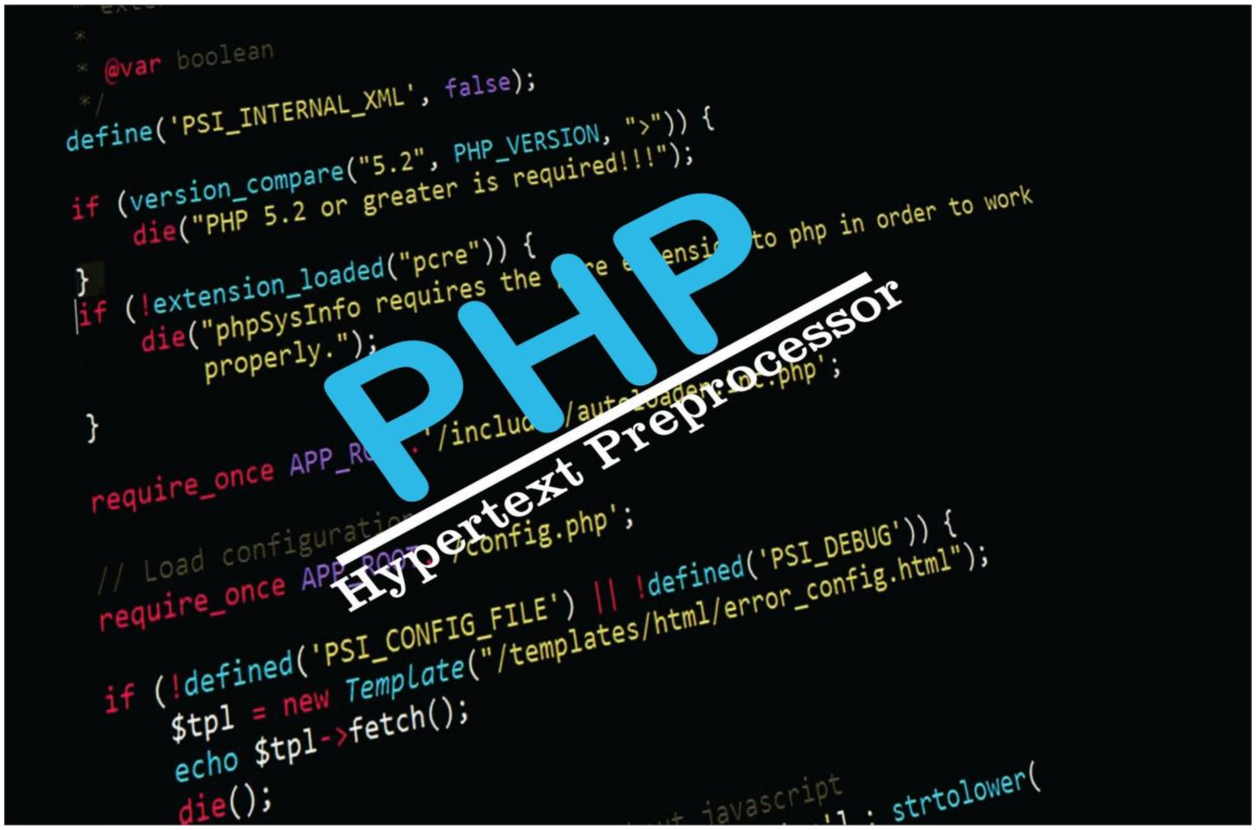

PHP TUTORIAL NOTES
by: SIR MUHAMMAD FARHAN

Contact No: +92316-2859445

Email: mohammadfarhan44500@gmail.com

GitHub: https://github.com/muhammadfarhandevelper

OOP PHP TUTORIALS

What is OOP?

OOP stands for Object-Oriented Programming.

Procedural programming is about writing procedures or functions that perform operations on the data, while object-oriented programming is about creating objects that contain both data and functions.

Object-oriented programming has several advantages over procedural programming:

- OOP is faster and easier to execute
- OOP provides a clear structure for the programs
- OOP helps to keep the PHP code DRY "Don't Repeat Yourself", and makes the code easier to maintain, modify and debug
- OOP makes it possible to create full reusable applications with less code and shorter development time

Tip: The "Don't Repeat Yourself" (DRY) principle is about reducing the repetition of code. You should extract out the codes that are common for the application, and place them at a single place and reuse them instead of repeating it.

Object Oriented Concepts:

Before we go in detail, lets define important terms related to Object Oriented Programming.

- **Class** − This is a programmer-defined data type, which includes local functions as well as local data. You can think of a class as a template for making many instances of the same kind (or class) of object.

- **Object** − An individual instance of the data structure defined by a class. You define a class once and then make many objects that belong to it. Objects are also known as instance.

- **Member Variable** − These are the variables defined inside a class. This data will be invisible to the outside of the class and can be accessed via member functions. These variables are called attribute of the object once an object is created.

- **Member function** − These are the function defined inside a class and are used to access object data.

- **Inheritance** − When a class is defined by inheriting existing function of a parent class then it is called inheritance. Here child class will inherit all or few member functions and variables of a parent class.

- **Parent class** − A class that is inherited from by another class. This is also called a base class or super class.

- **Child Class** − A class that inherits from another class. This is also called a subclass or derived class.

- **Polymorphism** − This is an object oriented concept where same function can be used for different purposes. For example function name will remain same but it take different number of arguments and can do different task.

- **Overloading** − a type of polymorphism in which some or all of operators have different implementations depending on the types of their arguments. Similarly functions can also be overloaded with different implementation.

- **Data Abstraction** − Any representation of data in which the implementation details are hidden (abstracted).

- **Encapsulation** − refers to a concept where we encapsulate all the data and member functions together to form an object.

- **Constructor** − refers to a special type of function which will be called automatically whenever there is an object formation from a class.

- **Destructor** − refers to a special type of function which will be called automatically whenever an object is deleted or goes out of scope.

What are Classes and Objects?

Classes and objects are the two main aspects of object-oriented programming.

Class Example:

```
class Fruits {
  public $name;
  public $color;

  function set_name($name) {
    $this->name = $name;
  }
  function get_name() {
    return $this->name;
  }
}
```

Define Object:

Classes are nothing without objects! We can create multiple objects from a class. Each object has all the properties and methods defined in the class, but they will have different property values.

Objects of a class is created using the new keyword.

Example:

```
class Fruit {

  public $name;
  public $color;

  function set_name($name) {
    $this->name = $name;
  }
  function get_name() {
    return $this->name;
  }
}

$apple = new Fruit();
$banana = new Fruit();
$apple->set_name('Apple');
$banana->set_name('Banana');

echo $apple->get_name();
echo "<br>";
echo $banana->get_name();
```

Example:

```
class Fruit {
   public $name;
   function set_name($name) {
     $this->name = $name;
   }
 }
 $apple = new Fruit();
 $apple->set_name("Apple");

 echo $apple->name;
```

Constructor:

A constructor allows you to initialize an object's properties upon creation of the object.

If you create a __construct() function, PHP will automatically call this function when you create an object from a class.

Notice that the construct function starts with two underscores (__)!

Example:

```
class Fruit {
   public $name;
   public $color;

   function __construct($name) {
     $this->name = $name;
   }
   function get_name() {
     return $this->name;
   }
 }

 $apple = new Fruit("Apple");
 echo $apple->get_name();
```

Destructor:

A destructor is called when the object is destructed or the script is stopped or exited.

If you create a __destruct() function, PHP will automatically call this function at the end of the script.

Notice that the destruct function starts with two underscores (__)!

The example below has a __construct() function that is automatically called when you create an object from a class, and a __destruct() function that is automatically called at the end of the script:

Example:

```
class Fruit {
  public $name;
  public $color;

  function __construct($name) {
    $this->name = $name;
  }
  function __destruct() {
    echo "The fruit is {$this->name}.";
  }
}

$apple = new Fruit("Apple");
```

Access Modifiers:

Properties and methods can have access modifiers which control where they can be accessed.

There are three access modifiers:

- public - the property or method can be accessed from everywhere. This is default
- protected - the property or method can be accessed within the class and by classes derived from that class
- private - the property or method can ONLY be accessed within the class

In the following example we have added three different access modifiers to three properties (name, color, and weight). Here, if you try to set the name property it will work fine (because the name property is public, and can be accessed from everywhere). However, if you try to set the color or weight property it will result in a fatal error (because the color and weight property are protected and private):

Example:

```
class Fruit {
  public $name;
  protected $color;
  private $weight;
}

$mango = new Fruit();
$mango->name = 'Mango'; // OK
$mango->color = 'Yellow'; // ERROR
$mango->weight = '300'; // ERROR
```

In the next example we have added access modifiers to two functions. Here, if you try to call the set_color() or the set_weight() function it will result in a fatal error (because the two functions are considered protected and private), even if all the properties are public:

Example:

```
class Fruit {
  public $name;
  public $color;
  public $weight;

  function set_name($n) {  // a public function (default)
    $this->name = $n;
  }
  protected function set_color($n) { // a protected function
    $this->color = $n;
  }
  private function set_weight($n) { // a private function
    $this->weight = $n;
  }
}

$mango = new Fruit();
$mango->set_name('Mango'); // OK
```

```
$mango->set_color('Yellow'); // ERROR
$mango->set_weight('300'); // ERROR
```

Inheritance:

Inheritance in OOP = When a class derives from another class.

The child class will inherit all the public and protected properties and methods from the parent class. In addition, it can have its own properties and methods.

An inherited class is defined by using the extends keyword.

Example:

```
class Fruit {
  public $name;
  public $color;
  public function __construct($name, $color) {
    $this->name = $name;
    $this->color = $color;
  }
  public function intro() {
    echo "The fruit is {$this->name} and the color is {$this->color}.";
  }
}

// Strawberry is inherited from Fruit
class Strawberry extends Fruit {
  public function message() {
    echo "Am I a fruit or a berry? ";
  }
}
$strawberry = new Strawberry("Strawberry", "red");
$strawberry->message();
$strawberry->intro();
```

Inheritance and the Protected Access Modifiers:

In the previous chapter we learned that protected properties or methods can be accessed within the class and by classes derived from that class. What does that mean?

Example:

```
class Fruit {
  public $name;
  public $color;
  public function __construct($name, $color) {
    $this->name = $name;
    $this->color = $color;
  }
  protected function intro() {
    echo "The fruit is {$this->name} and the color is {$this->color}.";
  }
}

class Strawberry extends Fruit {
  public function message() {
    echo "Am I a fruit or a berry? ";
  }
}

// Try to call all three methods from outside class
$strawberry = new Strawberry("Strawberry", "red");  // OK. __construct() is public
$strawberry->message(); // OK. message() is public
$strawberry->intro(); // ERROR. intro() is protected
```

In the example above we see that if we try to call a protected method (intro()) from outside the class, we will receive an error. public methods will work fine!

Example:

```
class Fruit {
  public $name;
  public $color;
  public function __construct($name, $color) {
    $this->name = $name;
    $this->color = $color;
  }
  protected function intro() {
    echo "The fruit is {$this->name} and the color is {$this->color}.";
  }
}

class Strawberry extends Fruit {
  public function message() {
    echo "Am I a fruit or a berry? ";
    // Call protected method from within derived class - OK
    $this -> intro();
  }
}

$strawberry = new Strawberry("Strawberry", "red"); // OK. __construct() is public
$strawberry->message();
```

Overriding Inherited Methods:

Inherited methods can be overridden by redefining the methods (use the same name) in the child class.

Look at the example below. The __construct() and intro() methods in the child class (Strawberry) will override the __construct() and intro() methods in the parent class (Fruit):

Example:

```
class Fruit {
  public $name;
  public $color;
  public function __construct($name, $color) {
    $this->name = $name;
    $this->color = $color;
  }
  public function intro() {
    echo "The fruit is {$this->name} and the color is {$this->color}.";
  }
}

class Strawberry extends Fruit {
  public $weight;
  public function __construct($name, $color, $weight) {
    $this->name = $name;
    $this->color = $color;
    $this->weight = $weight;
  }
  public function intro() {
    echo "The fruit is {$this->name}, the color is {$this->color}, and the weight is {$this->weight} gram.";
  }
}

$strawberry = new Strawberry("Strawberry", "red", 50);
```

```
$strawberry->intro();
```

Abstract Class:

Abstract classes and methods are when the parent class has a named method, but need its child class(es) to fill out the tasks.

An abstract class is a class that contains at least one abstract method. An abstract method is a method that is declared, but not implemented in the code.

Example:

```
abstract class ParentClass {
  abstract public function someMethod1();
  abstract public function someMethod2($name, $color);
  abstract public function someMethod3() : string;
}
```

Another Example:

```
// Parent class
abstract class Car {
  public $name;
  public function __construct($name) {
    $this->name = $name;
  }
  abstract public function intro() : string;
}

// Child classes
class Audi extends Car {
  public function intro() : string {
    return "Choose German quality! I'm an $this->name!";
  }
}

class Volvo extends Car {
  public function intro() : string {
    return "Proud to be Swedish! I'm a $this->name!";
  }
}

class Citroen extends Car {
  public function intro() : string {
    return "French extravagance! I'm a $this->name!";
  }
}

// Create objects from the child classes
$audi = new audi("Audi");
echo $audi->intro();
echo "<br>";

$volvo = new volvo("Volvo");
echo $volvo->intro();
echo "<br>";

$citroen = new citroen("Citroen");
echo $citroen->intro();
```

What are Interfaces:

Interfaces allow you to specify what methods a class should implement.

Interfaces make it easy to use a variety of different classes in the same way. When one or more classes use the same interface, it is referred to as "polymorphism".

Interfaces are declared with the interface keyword:

Example:

```
interface InterfaceName {
  public function someMethod1();
  public function someMethod2($name, $color);
  public function someMethod3() : string;
}
```

Interfaces vs Abstract Classes:

Interface are similar to abstract classes. The difference between interfaces and abstract classes are:

- Interfaces cannot have properties, while abstract classes can
- All interface methods must be public, while abstract class methods is public or protected
- All methods in an interface are abstract, so they cannot be implemented in code and the abstract keyword is not necessary
- Classes can implement an interface while inheriting from another class at the same time

Using Interfaces:

To implement an interface, a class must use the implements keyword.

A class that implements an interface must implement **all** of the interface's methods.

Example:

```
interface Animal {
  public function makeSound();
 }

 class Cat implements Animal {
  public function makeSound() {
    echo "Meow";
  }
 }

 $animal = new Cat();
 $animal->makeSound();
```

From the example above, let's say that we would like to write software which manages a group of animals. There are actions that all of the animals can do, but each animal does it in its own way.

Using interfaces, we can write some code which can work for all of the animals even if each animal behaves differently:

Example:

```
// Interface definition
interface Animal {
  public function makeSound();
 }

 // Class definitions
 class Cat implements Animal {
  public function makeSound() {
    echo " Meow ";
  }
 }

 class Dog implements Animal {
  public function makeSound() {
    echo " Bark ";
  }
 }

 class Mouse implements Animal {
  public function makeSound() {
    echo " Squeak ";
  }
 }

 // Create a list of animals
 $cat = new Cat();
 $dog = new Dog();
 $mouse = new Mouse();
 $animals = array($cat, $dog, $mouse);

 // Tell the animals to make a sound
 foreach($animals as $animal) {
  $animal->makeSound();
 }
```

Traits:

PHP only supports single inheritance: a child class can inherit only from one single parent.

So, what if a class needs to inherit multiple behaviors? OOP traits solve this problem.

Traits are used to declare methods that can be used in multiple classes. Traits can have methods and abstract methods that can be used in multiple classes, and the methods can have any access modifier (public, private, or protected).

Traits are declared with the trait keyword:

Example:

```
trait TraitName {
  // some code...
 }

class MyClass {
  use TraitName;
 }
```

Another Example:

```
trait message1 {
  public function msg1() {
     echo "OOP is fun! ";
    }
  }

  class Welcome {
    use message1;
  }

  $obj = new Welcome();
  $obj->msg1();
```

Static Methods:

Static methods can be called directly - without creating an instance of the class first.

Static methods are declared with the static keyword:

Example:

```
class ClassName {
  public static function staticMethod() {
    echo "Hello World!";
  }
}
```

To access a static method use the class name, double colon (::), and the method name:

Example:

```
class greeting {
  public static function welcome() {
    echo "Hello World!";
  }
}

// Call static method
greeting::welcome();
```

Static Properties:

Static properties can be called directly - without creating an instance of a class.

Static properties are declared with the static keyword:

```
class pi {
  public static $value = 3.14159;
}

// Get static property
echo pi::$value;
```

Namespaces:

Namespaces are qualifiers that solve two different problems:

1. They allow for better organization by grouping classes that work together to perform a task
2. They allow the same name to be used for more than one class

For example, you may have a set of classes which describe an HTML table, such as Table, Row and Cell while also having another set of classes to describe furniture, such as Table, Chair and Bed. Namespaces can be used to organize the classes into two different groups while also preventing the two classes Table and Table from being mixed up.

Declare Namespace:

Namespaces are declared at the beginning of a file using the namespace keyword:

Example:

```
namespace Html;
class Table {
  public $title = "";
  public $numRows = 0;
  public function message() {
    echo "<p>Table '{$this->title}' has {$this->numRows} rows.</p>";
  }
}
$table = new Table();
$table->title = "My table";
$table->numRows = 5;
```

What is Iterable?

An iterable is any value which can be looped through with a foreach() loop.

The iterable pseudo-type was introduced in PHP 7.1, and it can be used as a data type for function arguments and function return values.

Example:

```
function printIterable(iterable $myIterable) {
   foreach($myIterable as $item) {
     echo $item;
   }
 }
 $arr = ["a", "b", "c"];
 printIterable($arr);
```