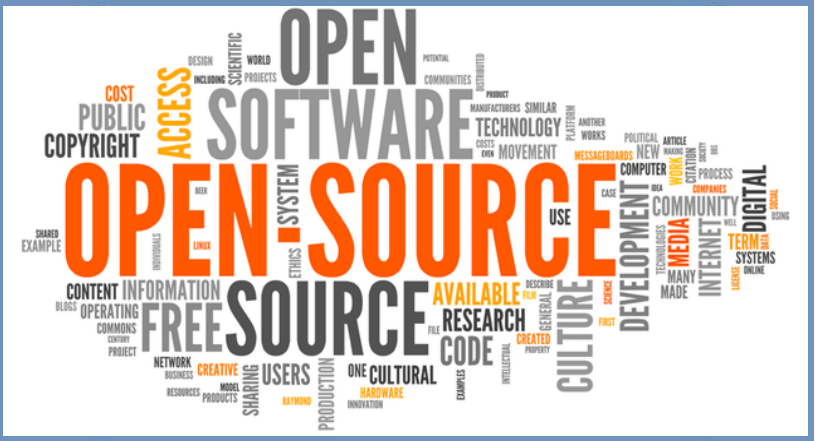
Open Source Software

Van Tham Nguyen

Thuyloi Univerity
175 Tay Son, Dong Da, Ha Noi

PHP-OOP



1. PHP OPP Introduction
2. Classes and Objects
3. Constructor and Destructor
4. Access Modifiers
5. Inheritance
6. Abstract class
7. Interfaces

1 PHP OPP Introduction

PHP 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.

PHP What is OOP?

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

PHP What is OOP?

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

PHP - What are Classes and Objects?

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

Look at the following illustration to see the difference between class and objects:

2 Classes and Objects

OOP Case

Let's assume we have a class named Fruit. A Fruit can have properties like name, color, weight, etc. We can define variables like $name, $color, and $weight to hold the values of these properties.

When the individual objects (apple, banana, etc.) are created, they inherit all the properties and behaviors from the class, but each object will have different values for the properties.

Define a Class

```php
<?php
class <Class Name> {
  // code goes here...
}
?>
```

```php
<?php
class Fruit {
  // Properties
  public $name;
  public $color;

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

Define Objects

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 are created using the new keyword.

```
<?php
$<Object Name>=new <Class Name>();
?>
```

Define Objects

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 are created using the new keyword.

```
<?php
$<Object Name>=new <Class 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();
?>
```

The $this Keyword

The $this keyword refers to the current object, and is only available inside methods.

1. Inside the class (by adding a set_name() method and use `$this`):

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

echo $apple->name;
?>
```

Apple

The $this Keyword

The `$this` keyword refers to the current object, and is only available inside methods.

2. Outside the class (by directly changing the property value):

```php
<?php
class Fruit {
  public $name;
}
$apple = new Fruit();
$apple->name = "Apple";

echo $apple->name;
?>
```

Apple

instanceof

You can use the `instanceof` keyword to check if an object belongs to a specific class:

```php
<?php
class Fruit {
  // Properties
  public $name;
  public $color;

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

$apple = new Fruit();
var_dump($apple instanceof Fruit);
?>
```

3 Constructor and Destructor

The __construct Function

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** (__)!

The __construct Function

```php
<?php
class Fruit {
  public $name;
  public $color;

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

$apple = new Fruit("Apple", "red");
echo $apple->get_name();
echo "<br>";
echo $apple->get_color();
?>
```

The __destruct Function

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 __destruct Function

```php
<?php
class Fruit {
  // Properties
  var $name;
  var $color;

  // Methods
  function __construct($name, $color) {
    $this->name = $name;
    $this->color = $color;
  }
  function __destruct() {
    echo "The fruit is {$this->name} and the color is {$this->color}.";
  }
}

$apple = new Fruit("Apple", "red");
?>
```

The fruit is Apple and the color is red.

4 Access Modifiers

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

Access Modifiers

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

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

Access Modifiers

```php
<?php
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
?>
```

5 Inheritance

What is 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.

Inheritance and the Protected Access Modifier

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?

```php
<?php
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}.";
  }
}
```

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

Inheritance and the Protected Access Modifier


```php
// 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
?>
```

The final Keyword

The `final` keyword can be used to prevent class inheritance or to prevent method overriding.

```php
<?php
class Fruit {
  final public function intro() {
  }
}

class Strawberry extends Fruit {
  // will result in error
  public function intro() {
  }
}
?>
```

6 Abstract class

Syntax

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

Abstract Classes and Methods

When inheriting from an abstract class, the child class method must be defined with the same name, and the same or a less restricted access modifier.

So, if the abstract method is defined as protected, the child class method must be defined as either protected or public, but not private.

Also, the type and number of required arguments must be the same. However, the child classes may have optional arguments in addition.

Abstract Classes and Methods

So, when a child class is inherited from an `abstract` class, we have the following rules:

The child class method must be defined with the same name and it redeclares the parent `abstract` method

The child class method must be defined with the same or a less restricted access modifier

The number of required arguments must be the same. However, the child class may have optional arguments in addition

Abstract Classes and Methods

```php
<?php
abstract class ParentClass {
  // Abstract method with an argument
  abstract protected function prefixName($name);
}
```

```php
class ChildClass extends ParentClass {
  public function prefixName($name) {
    if ($name == "John Doe") {
      $prefix = "Mr.";
    } elseif ($name == "Jane Doe") {
      $prefix = "Mrs.";
    } else {
      $prefix = "";
    }
    return "{$prefix} {$name}";
  }
}
```

Abstract Classes and Methods

```php
$class = new ChildClass;
echo $class->prefixName("John Doe");
echo "<br>";
echo $class->prefixName("Jane Doe");
```

Mr. John Doe

Mrs. Jane Doe

7 Interfaces

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:

```php
<?php
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.

```php
// 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 ";
  }
}
```

```php
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();
}
```

Static Methods

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

Static methods are declared with the `static` keyword:

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

Static Methods

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

```
ClassName::staticMethod();
```

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

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

More on Static Methods

A class can have both static and non-static methods. A static method can be accessed from a method in the same class using the self keyword and double colon (::):

```php
<?php
class greeting {
  public static function welcome() {
    echo "Hello World!";
  }
  public function __construct() {
    self::welcome();
  }
}

new greeting();
?>
```

Static Properties

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

Static properties are declared with the static keyword:

```
<?php
class ClassName {
  public static $staticProp = “Hello";
}
?>
```

Static Properties

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

```
ClassName::$staticProp;
```

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

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

More on Static Properties

A class can have both static and non-static properties. A static property can be accessed from a method in the same class using the self keyword and double colon (::):

```php
<?php
class pi {
  public static $value=3.14159;
  public function staticValue() {
    return self::$value;
  }
}

// Get static property
$pi = new pi();
echo $pi->staticValue();
?>
```

Any

Question