Search

Keyboard Shortcuts

?
: This help

j
: Next menu item

k
: Previous menu item

g p
: Previous man page

g n
: Next man page

G
: Scroll to bottom

g g
: Scroll to top

g h
: Goto homepage

g s
: Goto search
(current page)

/
: Focus search box

Интерфейсы объектов »
« Ключевое слово static

- Руководство по PHP
- Справочник языка
- Классы и объекты

Change language: Russian

Абстрактные классы

PHP поддерживает определение абстрактных классов и методов. На основе абстрактного класса нельзя создавать объекты, и любой класс, содержащий хотя бы один абстрактный метод, должен быть определён как абстрактный. Методы, объявленные абстрактными, несут, по существу, лишь описательный смысл и не могут включать реализацию.

При наследовании от абстрактного класса, все методы, помеченные абстрактными в родительском классе, должны быть определены в дочернем классе и следовать обычным правилам наследования и совместимости сигнатуры.

Пример #1 Пример абстрактного класса

```php
<?php
abstract class AbstractClass
{
// Данные методы должны быть определены в дочернем классе
abstract protected function getValue();
abstract protected function prefixValue($prefix);

// Общий метод
public function printOut() {
print $this->getValue() . "\n";
}
}

class ConcreteClass1 extends AbstractClass
{
protected function getValue() {
return "ConcreteClass1";
}

public function prefixValue($prefix) {
return "{$prefix}ConcreteClass1";
}
}

class ConcreteClass2 extends AbstractClass
{
public function getValue() {
return "ConcreteClass2";
}

public function prefixValue($prefix) {
return "{$prefix}ConcreteClass2";
}
}

$class1 = new ConcreteClass1;
$class1->printOut();
echo $class1->prefixValue('FOO_') ."\n";

$class2 = new ConcreteClass2;
$class2->printOut();
echo $class2->prefixValue('FOO_') ."\n";
?>
```

Результат выполнения приведённого примера:

```
ConcreteClass1
FOO_ConcreteClass1
ConcreteClass2
FOO_ConcreteClass2
```

Пример #2 Пример абстрактного класса

```
<?php
abstract class AbstractClass
{
// Наш абстрактный метод требует только определить необходимые аргументы
abstract protected function prefixName($name);

}

class ConcreteClass extends AbstractClass
{

// Наш дочерний класс может определить необязательные аргументы, не указанные в объявлении родительского метода
public function prefixName($name, $separator = ".") {
if ($name == "Pacman") {
$prefix = "Mr";
} elseif ($name == "Pacwoman") {
$prefix = "Mrs";
} else {
$prefix = "";
}
return "{$prefix}{$separator} {$name}";
}
}

$class = new ConcreteClass;
echo $class->prefixName("Pacman"), "\n";
echo $class->prefixName("Pacwoman"), "\n";
?>
```

Результат выполнения приведённого примера:

```
Mr. Pacman
Mrs. Pacwoman
```


User Contributed Notes 26 notes

725
ironiridis at gmail dot com ¶
15 years ago

Just one more time, in the simplest terms possible:

An Interface is like a protocol. It doesn't designate the behavior of the object; it designates how your code tells that object to act. An interface would be like the English Language: defining an interface defines how your code communicates with any object implementing that interface.

An interface is always an agreement or a promise. When a class says "I implement interface Y", it is saying "I promise to have the same public methods that any object with interface Y has".

On the other hand, an Abstract Class is like a partially built class. It is much like a document with blanks to fill in. It might be using English, but that isn't as important as the fact that some of the document is already written.

An abstract class is the foundation for another object. When a class says "I extend abstract class Y", it is saying "I use some methods or properties already defined in this other class named Y".

So, consider the following PHP:

```
<?php
class X implements Y { } // this is saying that "X" agrees to speak language "Y" with your code.
```

```
class X extends Y { } // this is saying that "X" is going to complete the partial class "Y".
?>
```

You would have your class implement a particular interface if you were distributing a class to be used by other people. The interface is an agreement to have a specific set of public methods for your class.

You would have your class extend an abstract class if you (or someone else) wrote a class that already had some methods written that you want to use in your new class.

These concepts, while easy to confuse, are specifically different and distinct. For all intents and purposes, if you're the only user of any of your classes, you don't need to implement interfaces.

up
down
288
mbajoras at gmail dot com ¶
14 years ago

Here's an example that helped me with understanding abstract classes. It's just a very simple way of explaining it (in my opinion). Lets say we have the following code:

```
<?php
class Fruit {
private $color;

public function eat() {
//chew
}

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

class Apple extends Fruit {
public function eat() {
//chew until core
}
}

class Orange extends Fruit {
public function eat() {
//peel
//chew
}
}
?>
```

Now I give you an apple and you eat it.

```
<?php
$apple = new Apple();
$apple->eat();
?>
```

What does it taste like? It tastes like an apple. Now I give you a fruit.

```
<?php
$fruit = new Fruit();
$fruit->eat();
?>
```

What does that taste like??? Well, it doesn't make much sense, so you shouldn't be able to do that. This is accomplished

by making the Fruit class abstract as well as the eat method inside of it.

```
<?php
abstract class Fruit {
private $color;

abstract public function eat();

public function setColor($c) {
$this->color = $c;
}
}
?>
```

Now just think about a Database class where MySQL and PostgreSQL extend it. Also, a note. An abstract class is just like an interface, but you can define methods in an abstract class whereas in an interface they are all abstract.

up
down
9
swashata4u at gmail dot com ¶
5 years ago

Here is another thing about abstract class and interface.

Sometimes, we define an interface for a `Factory` and ease out some common methods of the `Factory` through an `abstract` class.

In this case, the abstract class implements the interface, but does not need to implement all methods of the interface.

The simple reason is, any class implementing an interface, needs to either implement all methods, or declare itself abstract.

Because of this, the following code is perfectly ok.

```
<?php
interface Element {
/**
* Constructor function. Must pass existing config, or leave as
* is for new element, where the default will be used instead.
*
* @param array $config Element configuration.
*/
public function __construct( $config = [] );

/**
* Get the definition of the Element.
*
* @return array An array with 'title', 'description' and 'type'
*/
public static function get_definition();

/**
* Get Element config variable.
*
* @return array Associative array of Element Config.
*/
public function get_config();

/**
* Set Element config variable.
*
* @param array $config New configuration variable.
*
```

```
* @return void
*/
public function set_config( $config );
}

abstract class Base implements Element {

/**
* Element configuration variable
*
* @var array
*/
protected $config = [];

/**
* Get Element config variable.
*
* @return array Associative array of Element Config.
*/
public function get_config() {
return $this->config;
}

/**
* Create an eForm Element instance
*
* @param array $config Element config.
*/
public function __construct( $config = [] ) {
$this->set_config( $config );
}
}

class MyElement extends Base {

public static function get_definition() {
return [
'type' => 'MyElement',
];
}

public function set_config( $config ) {
// Do something here
$this->config = $config;
}
}

$element = new MyElement( [
'foo' => 'bar',
] );

print_r( $element->get_config() );
?>
```

You can see the tests being executed here and PHP 5.4 upward, the output is consistent. https://3v4l.org/8NqqW

up
down
64

a dot tsiaparas at watergate dot gr ¶

12 years ago

Abstraction and interfaces are two very different tools. The are as close as hammers and drills. Abstract classes may have implemented methods, whereas interfaces have no implementation in themselves.

Abstract classes that declare all their methods as abstract are not interfaces with different names. One can implement multiple interfaces, but not extend multiple classes (or abstract classes).

The use of abstraction vs interfaces is problem specific and the choice is made during the design of software, not its implementation. In the same project you may as well offer an interface and a base (probably abstract) class as a reference that implements the interface. Why would you do that?

Let us assume that we want to build a system that calls different services, which in turn have actions. Normally, we could offer a method called execute that accepts the name of the action as a parameter and executes the action.

We want to make sure that classes can actually define their own ways of executing actions. So we create an interface IService that has the execute method. Well, in most of your cases, you will be copying and pasting the exact same code for execute.

We can create a reference implemention for a class named Service and implement the execute method. So, no more copying and pasting for your other classes! But what if you want to extend MySLLi?? You can implement the interface (copy-paste probably), and there you are, again with a service. Abstraction can be included in the class for initialisation code, which cannot be predefined for every class that you will write.

Hope this is not too mind-boggling and helps someone. Cheers,
Alexios Tsiaparas
up
down
27
jai at shaped dot ca ¶
6 years ago
This example will hopefully help you see how abstract works, how interfaces work, and how they can work together. This example will also work/compile on PHP7, the others were typed live in the form and may work but the last one was made/tested for real:

```
<?php

const ¶ = PHP_EOL;

// Define things a product *has* to be able to do (has to implement)
interface productInterface {
public function doSell();
public function doBuy();
}

// Define our default abstraction
abstract class defaultProductAbstraction implements productInterface {
private $_bought = false;
private $_sold = false;
abstract public function doMore();
public function doSell() {
/* the default implementation */
$this->_sold = true;
echo "defaultProductAbstraction doSell: {$this->_sold}".¶;
}
public function doBuy() {
$this->_bought = true;
echo "defaultProductAbstraction doBuy: {$this->_bought}".¶;
}
}

class defaultProductImplementation extends defaultProductAbstraction {
public function doMore() {
echo "defaultProductImplementation doMore()".¶;
}
}
```

```
class myProductImplementation extends defaultProductAbstraction {
public function doMore() {
echo "myProductImplementation doMore() does more!".¶;
}
public function doBuy() {
echo "myProductImplementation's doBuy() and also my parent's dubai()".¶;
parent::doBuy();
}
}

class myProduct extends defaultProductImplementation {
private $_bought=true;
public function __construct() {
var_dump($this->_bought);
}
public function doBuy () {
/* non-default doBuy implementation */
$this->_bought = true;
echo "myProduct overrides the defaultProductImplementation's doBuy() here {$this->_bought}".¶;
}
}

class myOtherProduct extends myProductImplementation {
public function doBuy() {
echo "myOtherProduct overrides myProductImplementations doBuy() here but still calls parent too".¶;
parent::doBuy();
}
}

echo "new myProduct()".¶;
$product = new myProduct();

$product->doBuy();
$product->doSell();
$product->doMore();

echo ¶."new defaultProductImplementation()".¶;

$newProduct = new defaultProductImplementation();
$newProduct->doBuy();
$newProduct->doSell();
$newProduct->doMore();

echo ¶."new myProductImplementation".¶;
$lastProduct = new myProductImplementation();
$lastProduct->doBuy();
$lastProduct->doSell();
$lastProduct->doMore();

echo ¶."new myOtherProduct".¶;
$anotherNewProduct = new myOtherProduct();
$anotherNewProduct->doBuy();
$anotherNewProduct->doSell();
$anotherNewProduct->doMore();
?>

Will result in:
<?php
/*
new myProduct()
bool(true)
```

```
myProduct overrides the defaultProductImplementation's doBuy() here 1
defaultProductAbstraction doSell: 1
defaultProductImplementation doMore()

new defaultProductImplementation()
defaultProductAbstraction doBuy: 1
defaultProductAbstraction doSell: 1
defaultProductImplementation doMore()

new myProductImplementation
myProductImplementation's doBuy() and also my parent's dubai()
defaultProductAbstraction doBuy: 1
defaultProductAbstraction doSell: 1
myProductImplementation doMore() does more!

new myOtherProduct
myOtherProduct overrides myProductImplementations doBuy() here but still calls parent too
myProductImplementation's doBuy() and also my parent's dubai()
defaultProductAbstraction doBuy: 1
defaultProductAbstraction doSell: 1
myProductImplementation doMore() does more!

*/
?>
```

up
down
16

shaman_master at list dot ru ¶

5 years ago

```
Also you may set return/arguments type declaring for abstract methods (PHP>=7.0)
<?php
declare(strict_types=1);

abstract class Adapter
{
protected $name;
abstract public function getName(): string;
abstract public function setName(string $value);
}

class AdapterFoo extends Adapter
{
public function getName(): string
{
return $this->name;
}
// return type declaring not defined in abstract class, set here
public function setName(string $value): self
{
$this->name = $value;
return $this;
}
}
?>
```

up
down
68

joelhy ¶

12 years ago

```
The documentation says: "It is not allowed to create an instance of a class that has been defined as abstract.". It only
means you cannot initialize an object from an abstract class. Invoking static method of abstract class is still feasible.
For example:
```

```
<?php
abstract class Foo
{
static function bar()
{
echo "test\n";
}
}

Foo::bar();
?>
```

up
down
2

shewa12kpi at gmail dot com ¶
2 years ago

```
<?php
//Here is a good example of abstract class. Here BaseEmployee is not actual employee its just asbtract class that reduce our code and enforce child class to implement abstract method

abstract class BaseEmployee {

/**
* employee common attributes could in asbtract class
*/
public $firstname,
$lastname;

function __construct($fn, $ln){
$this->firstname = $fn;
$this->lastname = $ln;
}

public function getFullName() {

return "$this->firstname $this->lastname";
}

/**
* asbtract method that a child class must have to define
*/
abstract protected static function task();
}

class WebDeveloper extends BaseEmployee {

static function task()
{
return ' Develop web application';
}
}

class HR extends BaseEmployee {

static function task()
{
return ' Manage human resources';
}

}
/**
* now instantiate and get data
```

```
*/
$webDeveloper = new WebDeveloper('shaikh','ahmed');
echo $webDeveloper->getFullName();
echo $webDeveloper->task();
```

up
down
16

sneakyimp at hotmail dot com ¶

16 years ago

Ok...the docs are a bit vague when it comes to an abstract class extending another abstract class. An abstract class that extends another abstract class doesn't need to define the abstract methods from the parent class. In other words, this causes an error:

```
<?php
abstract class class1 {
abstract public function someFunc();
}
abstract class class2 extends class1 {
abstract public function someFunc();
}
?>
```

```
Error: Fatal error: Can't inherit abstract function class1::someFunc() (previously declared abstract in class2) in
/home/sneakyimp/public/chump.php on line 7
```

However this does not:

```
<?php
abstract class class1 {
abstract public function someFunc();
}
abstract class class2 extends class1 {
}
?>
```

An abstract class that extends an abstract class can pass the buck to its child classes when it comes to implementing the abstract methods of its parent abstract class.

up
down
7

jai at shaped dot ca ¶

6 years ago

An interface specifies what methods a class must implement, so that anything using that class that expects it to adhere to that interface will work.

eg: I expect any $database to have ->doQuery(), so any class I assign to the database interface should implement the databaseInterface interface which forces implementation of a doQuery method.

```
<?php
interface dbInterface {
public function doQuery();
}

class myDB implements dbInterface {
public function doQuery() {
/* implementation details here */
}
}

$myDBObj = new myDB()->doQuery();
?>
```

An abstract class is similar except that some methods can be predefined. Ones listed as abstract will have to be defined as if the abstract class were an interface.

eg. I expect my $person to be able to ->doWalk(), most people walk fine with two feet, but some people have to hop along :(

```
<?php
interface PersonInterface() {
/* every person should walk, or attempt to */
public function doWalk($place);
/* every person should be able to age */
public function doAge();
}

abstract class AveragePerson implements PersonInterface() {
private $_age = 0;
public function doAge() {
$this->_age = $this->_age+1;
}
public function doWalk($place) {
echo "I am going to walk to $place".PHP_EOL;
}
/* every person talks differently! */
abstract function talk($say);
}

class Joe extends AveragePerson {
public function talk($say) {
echo "In an Austrailian accent, Joe says: $say".PHP_EOL;
}
}

class Bob extends AveragePerson {
public function talk($say) {
echo "In a Canadian accent, Bob says: $say".PHP_EOL;
}
public function doWalk($place) {
echo "Bob only has one leg and has to hop to $place".PHP_EOL;
}
}

$people[] = new Bob();
$people[] = new Joe();

foreach ($people as $person) {
$person->doWalk('over there');
$person->talk('PHP rules');
}
?>
```

up
down
16

bishop ¶
13 years ago

Incidentally, abstract classes do not need to be base classes:

```
<?php
class Foo {
public function sneeze() { echo 'achoooo'; }
}

abstract class Bar extends Foo {
```

```
public abstract function hiccup();
}

class Baz extends Bar {
public function hiccup() { echo 'hiccup!'; }
}

$baz = new Baz();
$baz->sneeze();
$baz->hiccup();
?>
```

up
down
3

Eugeny at Kostanay dot KZ ¶

7 years ago

A snippet of code to help you understand a bit more about properties inside abstract classes:

```
<?php
abstract class anotherAbsClass
{
// Define and set a static property
static $stProp = 'qwerty'; // We can still use it directly by the static way
// Define and set a protected property
protected $prProp = 'walrus';
// It is useless to set any other level of visibility for non-static variables of an abstract class.
// We cannot access to a private property even inside a declared method of an abstract class because we cannot call that
method in the object context.
// Implementation of a common method
protected function callMe() {
echo 'On call: ' . $this->prProp . PHP_EOL;
}
// Declaration of some abstract methods
abstract protected function abc($arg1, $arg2);
abstract public function getJunk($arg1, $arg2, $arg3, $junkCollector = true);
// Note: we cannot omit an optional value without getting error if it has already been declared by an abstract class
}

class someChildClass extends anotherAbsClass
{
function __construct() {
echo $this->callMe() . PHP_EOL; // now we get the protected property $prProp inhereted from within the abstract class
}
// There must be implementation of the declared functions abc and getJunk below
protected function abc($val1, $val) {
// do something
}
function getJunk($val1, $val2, $val3, $b = false) { // optional value is neccessary, because it has been declared above
// do something
}
}

echo anotherAbsClass::$stProp; // qwerty
$objTest = new someChildClass; // On call: walrus
?>
```

up
down
3

Malcolm ¶

7 years ago

I've found an inconsistency with: Example #2 Abstract class example

If you remove the default value of $separator

```
<?php
public function prefixName($name, $separator) {
// ...
}
?>
```

Then php will show this fatal message:
Fatal error: Declaration of ConcreteClass::prefixName() must be compatible with AbstractClass::prefixName($name) in /index.php on line 23

Stange enough it gives an incorrect declaration of "ConcreteClass::prefixName()"... It is missing both arguments. Because of that I'm assuming that this is just a bug that maybe already has been taking care of in newer versions. (Or is just specific to my version) I'm mainly noting this because it was driving me absolutely insane in some test code that I was writing derived from Example #2 (without a default value for an extra argument). Perhaps this saves some frustrations to other people.

--
Please note that i'm running this on php5.5.
OS: ubuntu-16.04-server-amd64.iso
Repo: ppa:ondrej/php

```
# php5.5 --version
PHP 5.5.36-2+donate.sury.org~xenial+1 (cli)
Copyright (c) 1997-2015 The PHP Group
Zend Engine v2.5.0, Copyright (c) 1998-2015 Zend Technologies with Zend OPcache v7.0.6-dev, Copyright (c) 1999-2015, by Zend Technologies
```

up
down
6
joebert ¶
16 years ago

I don't agree with jfkallens' last comparison between Abstract Classes & Object Interfaces completely.

In an Abstract Class, you can define how some methods work, where as in an Object Interface you can not.

An Object Interface is essentually nothing but a list of function names that a class must define if the class implements that interface.

An Abstract Class is essentually a prototype which hints towards what extending classes should be doing.
An Abstract Class can also be thought of as a Base Class that provides some basic functionality, & also defines a built-in Object Interface that all extending classes will implement.

So, an Object Interface is really a built-in part of an Abstract Class.

up
down
1
arma99eDAN at yahoo dot com ¶
8 years ago

You can use an abstract class like this too:

```
abstract class A{
public function show(){
echo 'A';
}
}
class B extends A{
public function hello(){
echo 'B';
parent::show();
}
}
```

```
$obj = new B;
$obj->hello(); // BA
# See that the abstract class does not have at least one abstract method
# Even in this case, I'm still able to extend it, or call its non-abstract member
```

up
down
4

pete at surfaceeffect dot com ¶

14 years ago

One fairly important difference between php's abstract functions and, say, Java, is that php does not specify the return type in any way - or indeed whether there has to be one.

```
<?php public abstract function square($number); ?>
```

could be implemented by...

```
<?php
public function square($number) {
return $number*$number;
}
?>
```

or

```
<?php
public function square($number) {
print ($number*$number);
}
?>
```

So you need to take care that incompatibilities don't arise due to not returning the right kind of value and this is not enforced in any way.

up
down
-1

aloydev2586 at gmail dot com ¶

9 years ago

Just in case you are confused about function arguments:

```
/*************Example 1*******************/

abstract class my_class {
abstract public function my_function($number);
}

class subclass extends my_class {
public function my_function($new_number, $string = ' is an integer!!!')
{
echo $new_number . $string;
}
}
$var = new subclass();
$var->my_function(1024); //this will output: 1024 is an integer!!!

/*************Example 2*******************/
abstract class my_class {
abstract public function my_function($number);
}

class subclass extends my_class {
//now $string = ' is a float!!!'
```

```
public function my_function($new_number, $string = ' is a float!!!')
{
echo $new_number . $string;
}

}
$var = new subclass();
//added ' is an integer'
$var->my_function(1024, ' is an integer!!!'); //this will output: 1024 is an integer!!!, rewrote $string.

/*************Example 3*******************/

abstract class my_class {
abstract public function my_function($number);
}

class subclass extends my_class {
//now $string isn't initialized
public function my_function($new_number, $string )
{
echo $new_number . $string;
}

}

$var = new subclass();
$var->my_function(1024, ' is an integer!!!'); /*this will trigger a fatal error of incompatibility between
subclass::my_function() and my_class::my_function($number)*/

/*************Example 4*******************/

abstract class my_class {
abstract public function my_function($number);
}

class subclass extends my_class {
public function my_function($new_number, $string )
{
echo $new_number . $string;
}

}
$var = new subclass();
//no second argument, no matter
$var->my_function(1024);//fatal error too. Optional arguments have to be initialized in the extending class function.
```

up
down
-1

designbyjeeba at gmail dot com ¶

13 years ago

Please be aware of the visibility of the parent fields. If the fields are private, then you are not going to see those fields in their childrens. Its basic OOP, but can be problematic sometimes.

up
down
-1

nathan dot vorbei dot tech at gmail dot com ¶

14 years ago

"additionally, these methods must be defined with the same (or a less restricted) visibility."

The words were not restricted in abstract class but also normal classes,
the method in child Class which overwrites the parent Class can also change the the visibility of the method to same or less restricted.

for example:

```
<?php
class ClassOne {
protected static $staticone = 'nathan';
protected function changestaticone() {
return self::$staticone = 'john';
}
}

class ClassTwo extends ClassOne {
public function changestaticone() {
return self::$staticone = 'Alexey';
}
}
$classtwo = new ClassTwo();
echo $classtwo->changestaticone();
```

up
down
-3
Cheese Doodle ¶
14 years ago

There isn't really that much of a great hurdle in understanding these things, there really isn't.

If you're defining a new class that is abstract, it means that you can make some non-abstract functions that you can use to define the general underlying behavior of that class along side abstract ones.

In interfaces, you can't do that since functions defined therewithin cannot have a body.

Abstract functions you use for classes that must define more specific behavior when "extending" your class.

So for a crude example - define by your non-abstract functions how that particular object (which may be part of a larger class hierarchy) would store and process it's data in SQL, XML, etc.

Then define abstract functions which allow someone implementing that class to specifically manipulate the data that is to be stored. Then require a format which this data must be returned in, and then in your non-abstract functions call those functions on destruction, in normal runtime, and so on.

Again, non-abstract functions, or even another class could implement the finer points of ensuring that data is in the correct format, and so on, ad infinitum.

It isn't too much of a reach to say that if you used a normal class instead of an abstract class, then there isn't much intrinsic requirement between the two classes.

Assuming that you wanted the functions to use each-others functions and you'd need to use them specifically by name, you'd have to write some code which checked to see -- lamely using function_exists() and other lamery -- if that class has the function you require for interoperability, when you could avoid all possible confusion and headaches by simply using the right tool for the job.

And reading a decent OOP book.

up
down
-4
eeescalona ¶
15 years ago

here is a real world example of abstract using:

a (abstract) person class
a student and an employee final class, which extends person class.

simple theory is that both student and employee is an extension of the person class. the difference lies on which table the data is written on, and what other pre processing (ie mandatory field checking, type checking, etc.) needed before writing each of the classes.

codes:

```
<?php

abstract class person {

abstract protected function write_info();

public $LastName;
public $FirstName;
public $BirthDate;

public function get_Age($today=NULL){
//age computation function
}
}

final class employee extends person{
public $EmployeeNumber;
public $DateHired;

public function write_info(){
echo "Writing ". $this->LastName . "'s info to emloyee dbase table";
//ADD unique mandatory checking unique to EMPLOYEE ONLY
//actual sql codes here
}
}

final class student extends person{
public $StudentNumber;
public $CourseName;

public function write_info(){
echo "Writing ". $this->LastName . "'s info to student dbase table";
//ADD unique mandatory checking unique to STUDENT ONLY
//actual sql codes here
}
}

///----------
$personA = new employee;
$personB = new student;

$personA->FirstName="Joe";
$personA->LastName="Sbody";

$personB->FirstName="Ben";
$personB->LastName="Dover";

$personA->write_info();
?>

OUTPUT:Writing Sbody's info to emloyee dbase table
```

up
down
-4
rmoisto at gmail dot com ¶
8 years ago

The self keyword in an abstract class will refer to the abstract class itself, not the extending class no matter what.

For instance the following code looks really pretty, yet results in a Fatal error (Cannot instantiate abstract class

```
Basic).

<?php
abstract class Basic {
public static function doWork() {
return (new self())->work();
}

abstract protected function work();
}

class Keeks extends Basic {
protected function work() {
return 'Keeks';
}
}

echo Keeks::doWork();
?>
```

up
down
-5

nikola at petkanski dot com ¶

10 years ago

```
Invoking static method of abstract class should be removed.

What interfaces are?
- A mean to ensure all implementation have the same methods implemented.

What is an abstract class?
- It is a interface that can also include some concrete methods.

Is it right for the developer to be able to invoke a static method of an interface?
- I think not.

The GoF teach us to rely on abstract classes and interfaces to hide differences between subclasses from clients.
- Interface defines an object’s use (protocol)
- Implementation defines particular policy

I don't think one should be able to call some abstract logic that is defined inside an abstract class, without even
inheriting the class itself.
```

up
down
-6

sam at righthandedmonkey dot com ¶

9 years ago

```
Please note order or positioning of the classes in your code can affect the interpreter and can cause a Fatal error: Class
'YourClass' not found if there are multiple levels of abstraction out of order. For example:
<?php
abstract class horse extends animal {
public function get_breed() { return "Jersey"; }
}

class cart extends horse {
public function get_breed() { return "Wood"; }
}

abstract class animal {
public abstract function get_breed();
}

$cart = new cart();
```

```
print($cart->get_breed());
?>

this outputs:
Wood

However, if you put the cart before the abstract horse (literally):

<?php
class cart extends horse {
public function get_breed() { return "Wood"; }
}

abstract class horse extends animal {
public function get_breed() { return "Jersey"; }
}

abstract class animal {
public abstract function get_breed();
}

$cart = new cart();
print($cart->get_breed());
?>

this throws an error:
Fatal error: Class 'horse' not found

So, when using multiple levels of abstraction, be careful of the positioning of the classes within the source code - and don't put the cart before the abstract horse.
```

up
down
-14
oliver at ananit dot de ¶
12 years ago

```
Abstract classes may have an final constructor, and sometime it makes sense to implement a class with a final constructor.

<?php
abstract class AbstractModel
{
//our models must use the default constuctor
public final function __construct(){}
public function inject($array){
foreach(array_keys(get_class_vars(get_called_class())) as $property){
$this->$property = $array[$property];
}
}
}

class ProductModel extends AbstractModel
{
public $name;
public $price;
protected $id;

public function getId(){return $this->id;}
}

class Factory{
private $dataSource;
public function __consruct($dataSource){
$this->dataSource = $dataSource;
```

```
}

public function get($class, $table, $filter, $orderby, $limit){
$result = array();
foreach($datasource->fetchAssoc($table, $filter, $orderby, $limit) as $rawData){
$obj = new $class();//this can only work if ALL models have a default constructor
$obj->inject($rawData);
$result[] = $obj;
}
return $result;
}
}
?>

Note: This is a very simple example, and I am aware that there are other (better) ways to do this.
Oliver Anan
```

up
down
-18

balbuf ¶
8 years ago

The abstract keyword cannot be used to dictate properties or class constants that a derivative class must set/define. Instead, those required properties or constants can be included in the abstract class with the expectation that they will be overridden in derivative classes, which at least ensures that the desired property/constant is set/defined.

＋add a note