Search

Getting Started
- Introduction
- A simple tutorial

Language Reference
- Basic syntax
- Types
- Variables
- Constants
- Expressions
- Operators
- Control Structures
- Functions
- Classes and Objects
- Namespaces
- Enumerations
- Errors
- Exceptions
- Fibers
- Generators
- Attributes
- References Explained
- Predefined Variables
- Predefined Exceptions
- Predefined Interfaces and Classes
- Predefined Attributes
- Context options and parameters
- Supported Protocols and Wrappers

Security
- Introduction
- General considerations
- Installed as CGI binary
- Installed as an Apache module
- Session Security
- Filesystem Security
- Database Security
- Error Reporting
- User Submitted Data
- Hiding PHP
- Keeping Current

Features
- HTTP authentication with PHP
- Cookies
- Sessions
- Dealing with XForms
- Handling file uploads
- Using remote files
- Connection handling
- Persistent Database Connections
- Command line usage

Garbage Collection
DTrace Dynamic Tracing

Function Reference
Affecting PHP's Behaviour
Audio Formats Manipulation
Authentication Services
Command Line Specific Extensions
Compression and Archive Extensions
Cryptography Extensions
Database Extensions
Date and Time Related Extensions
File System Related Extensions
Human Language and Character Encoding Support
Image Processing and Generation
Mail Related Extensions
Mathematical Extensions
Non-Text MIME Output
Process Control Extensions
Other Basic Extensions
Other Services
Search Engine Extensions
Server Specific Extensions
Session Extensions
Text Processing
Variable and Type Related Extensions
Web Services
Windows Only Extensions
XML Manipulation
GUI Extensions

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

Свойства »
« Введение

- Руководство по PHP
- Справочник языка
- Классы и объекты

Change language: Russian

Основы

class

Каждое определение класса начинается с ключевого слова `class`, затем следует имя класса, и далее пара фигурных скобок, которые заключают в себе определение свойств и методов этого класса.

Именем класса может быть любое слово, при условии, что оно не входит в список зарезервированных слов PHP, начинается с буквы или символа подчёркивания и за которым следует любое количество букв, цифр или символов подчёркивания. Если задать эти правила в виде регулярного выражения, то получится следующее выражение: `^[a-zA-Z_\x80-\xff][a-zA-Z0-9_\x80-\xff]*$`.

Класс может содержать собственные константы, переменные (называемые свойствами) и функции (называемые методами).

Пример #1 Простое определение класса

```
<?php
class SimpleClass
{
// объявление свойства
public $var = 'значение по умолчанию';

// объявление метода
public function displayVar() {
echo $this->var;
}
}
?>
```

Псевдопеременная *$this* доступна в том случае, если метод был вызван в контексте объекта. *$this* - значение вызывающего объекта.

Внимание

Вызов нестатического метода статически вызывает ошибку Error. До PHP 8.0.0 это привело бы к уведомлению об устаревании, и *$this* не была бы определена.

Пример #2 Некоторые примеры псевдо-переменной *$this*

```
<?php
class A
{
function foo()
{
if (isset($this)) {
echo '$this определена (';
echo get_class($this);
echo ")\n";
} else {
echo "\$this не определена.\n";
}
}
}

class B
{
function bar()
{
A::foo();
}
}
```

```php
$a = new A();
$a->foo();

A::foo();

$b = new B();
$b->bar();

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

Результат выполнения приведённого примера в PHP 7:

```
$this определена (A)

Deprecated: Non-static method A::foo() should not be called statically in %s  on line 27
$this не определена.

Deprecated: Non-static method A::foo() should not be called statically in %s  on line 20
$this не определена.

Deprecated: Non-static method B::bar() should not be called statically in %s  on line 32

Deprecated: Non-static method A::foo() should not be called statically in %s  on line 20
$this не определена.
```

Результат выполнения приведённого примера в PHP 8:

```
$this определена (A)

Fatal error: Uncaught Error: Non-static method A::foo() cannot be called statically in %s :27
Stack trace:
#0 {main}
  thrown in %s  on line 27
```

Классы, доступные только для чтения

Начиная с PHP 8.2.0, класс может быть помечен модификатором readonly. Пометка класса как readonly добавит модификатор readonly к каждому объявленному свойству и предотвратит создание динамических свойств. Более того, невозможно добавить их поддержку с помощью атрибута AllowDynamicProperties. Попытка это сделать приведёт к ошибке компиляции.

```php
<?php
#[\AllowDynamicProperties]
readonly class Foo {
}

// Fatal error: Cannot apply #[AllowDynamicProperties] to readonly class Foo
?>
```

Поскольку ни нетипизированные, ни статические свойства не могут быть помечены модификатором `readonly`, классы, доступные только для чтения также не могут их объявлять:

```php
<?php
readonly class Foo
{
public $bar;
}

// Fatal error: Readonly property Foo::$bar must have type
?>
<?php
readonly class Foo
{
public static int $bar;
}
```

```php
// Fatal error: Readonly class Foo cannot declare static properties
?>
```

Класс readonly может быть расширен тогда и только тогда, когда дочерний класс также является классом readonly.

new

Для создания экземпляра класса используется директива new. Новый объект всегда будет создан, за исключением случаев, когда он содержит конструктор, в котором определён вызов исключения в случае возникновения ошибки. Рекомендуется определять классы до создания их экземпляров (в некоторых случаях это обязательно).

Если с директивой new используется строка (string), содержащая имя класса, то будет создан новый экземпляр этого класса. Если имя находится в пространстве имён, то оно должно быть задано полностью.

> **Замечание**:
>
> В случае отсутствия аргументов в конструктор класса, круглые скобки после названия класса можно опустить.

Пример #3 Создание экземпляра класса

```php
<?php
$instance = new SimpleClass();

// Это же можно сделать с помощью переменной:
$className = 'SimpleClass';
$instance = new $className(); // new SimpleClass()
?>
```

Начиная с PHP 8.0.0, поддерживается использование оператора new с произвольными выражениями. Это позволяет создавать более сложные экземпляры, если выражение представлено в виде строки (string). Выражения должны быть заключены в круглые скобки.

Пример #4 Создание экземпляра с использованием произвольного выражения

В данном примере мы показываем несколько вариантов допустимых произвольных выражений, которые представляют имя класса. Пример вызова функции, конкатенации строк и константы **::class**.

```php
<?php

class ClassA extends \stdClass {}
class ClassB extends \stdClass {}
class ClassC extends ClassB {}
class ClassD extends ClassA {}

function getSomeClass(): string
{
return 'ClassA';
}

var_dump(new (getSomeClass()));
var_dump(new ('Class' . 'B'));
var_dump(new ('Class' . 'C'));
var_dump(new (ClassD::class));
?>
```

Результат выполнения приведённого примера в PHP 8:

```
object(ClassA)#1 (0) {
}
object(ClassB)#1 (0) {
}
object(ClassC)#1 (0) {
}
object(ClassD)#1 (0) {
}
```

В контексте класса можно создать новый объект через `new self` и `new parent`.

Когда происходит присвоение уже существующего экземпляра класса новой переменной, то эта переменная будет указывать на этот же экземпляр класса. То же самое происходит и при передаче экземпляра класса в функцию. Копию уже созданного объекта можно создать через её клонирование.

Пример #5 Присваивание объекта

```php
<?php

$instance = new SimpleClass();

$assigned = $instance;
$reference =& $instance;

$instance->var = '$assigned будет иметь это значение';

$instance = null; // $instance и $reference становятся null

var_dump($instance);
var_dump($reference);
var_dump($assigned);
?>
```

Результат выполнения приведённого примера:

```
NULL
NULL
object(SimpleClass)#1 (1) {
   ["var"]=>
     string(30) "$assigned будет иметь это значение"
}
```

Создавать экземпляры объекта можно двумя способами:

Пример #6 Создание новых объектов

```php
<?php
class Test
{
static public function getNew()
{
return new static;
}
}

class Child extends Test
{}

$obj1 = new Test();
$obj2 = new $obj1;
var_dump($obj1 !== $obj2);

$obj3 = Test::getNew();
var_dump($obj3 instanceof Test);

$obj4 = Child::getNew();
var_dump($obj4 instanceof Child);
?>
```

Результат выполнения приведённого примера:

```
bool(true)
bool(true)
bool(true)
```

Обратиться к свойству или методу только что созданного объекта можно с помощью одного выражения:

Пример #7 Доступ к свойствам/методам только что созданного объекта

```
<?php
echo (new DateTime())->format('Y');
?>
```

Вывод приведённого примера будет похож на:

```
2016
```

> **Замечание**: До PHP 7.1 аргументы не имели значения, если не определена функция конструктора.

Свойства и методы

Свойства и методы класса живут в разделённых "пространствах имён", так что возможно иметь свойство и метод с одним и тем же именем. Ссылки как на свойства, так и на методы имеют одинаковую нотацию, и получается, что получите вы доступ к свойству или же вызовете метод - определяется контекстом использования.

Пример #8 Доступ к свойству vs. вызов метода

```
<?php
class Foo
{
public $bar = 'свойство';

public function bar() {
return 'метод';
}
}

$obj = new Foo();
echo $obj->bar, PHP_EOL, $obj->bar(), PHP_EOL;
```

Результат выполнения приведённого примера:

```
свойство
метод
```

Это означает, что вызвать анонимную функцию, присвоенную переменной, напрямую не получится. Вместо этого свойство должно быть назначено, например, переменной. Можно вызвать такое свойство напрямую, заключив его в скобки.

Пример #9 Вызов анонимной функции, содержащейся в свойстве

```
<?php
class Foo
{
public $bar;

public function __construct() {
$this->bar = function() {
return 42;
};
}
}

$obj = new Foo();

echo ($obj->bar)(), PHP_EOL;
```

Результат выполнения приведённого примера:

```
42
```

extends

Класс может наследовать константы, методы и свойства другого класса используя ключевое слово `extends` в его

объявлении. Невозможно наследовать несколько классов, один класс может наследовать только один базовый класс.

Наследуемые константы, методы и свойства могут быть переопределены (за исключением случаев, когда метод или константа класса объявлены как final) путём объявления их с теми же именами, как и в родительском классе. Существует возможность доступа к переопределённым методам или статическим свойствам путём обращения к ним через parent::

> **Замечание**: Начиная с PHP 8.1.0, константы можно объявлять окончательными (final).

Пример #10 Простое наследование классов

```
<?php
class ExtendClass extends SimpleClass
{
// Переопределение метода родителя
function displayVar()
{
echo "Расширенный класс\n";
parent::displayVar();
}
}

$extended = new ExtendClass();
$extended->displayVar();
?>
```

Результат выполнения приведённого примера:

```
Расширенный класс
значение по умолчанию
```

Правила совместимости сигнатуры

При переопределении метода его сигнатура должна быть совместима с родительским методом. В противном случае выдаётся фатальная ошибка или, до PHP 8.0.0, генерируется ошибка уровня **`E_WARNING`**. Сигнатура является совместимой, если она соответствует правилам контравариантности, делает обязательный параметр необязательным, добавляет только необязательные новые параметры и не ограничивает, а только ослабляет видимость. Это известно как принцип подстановки Барбары Лисков или сокращённо LSP. Правила совместимости не распространяются на конструктор и сигнатуру `private` методов, они не будут выдавать фатальную ошибку в случае несоответствия сигнатуры.

Пример #11 Совместимость дочерних методов

```
<?php

class Base
{
public function foo(int $a) {
echo "Допустимо\n";
}
}

class Extend1 extends Base
{
function foo(int $a = 5)
{
parent::foo($a);
}
}

class Extend2 extends Base
{
function foo(int $a, $b = 5)
{
parent::foo($a);
```

```
}
}

$extended1 = new Extend1();
$extended1->foo();
$extended2 = new Extend2();
$extended2->foo(1);
```

Результат выполнения приведённого примера:

```
Допустимо
Допустимо
```

Следующие примеры демонстрируют, что дочерний метод, который удаляет параметр или делает необязательный параметр обязательным, несовместим с родительским методом.

Пример #12 Фатальная ошибка, когда дочерний метод удаляет параметр

```
<?php

class Base
{
public function foo(int $a = 5) {
echo "Допустимо\n";
}
}

class Extend extends Base
{
function foo()
{
parent::foo(1);
}
}
```

Результат выполнения приведённого примера в PHP 8 аналогичен:

```
Fatal error: Declaration of Extend::foo() must be compatible with Base::foo(int $a = 5) in /in/evtlq on line 13
```

Пример #13 Фатальная ошибка, когда дочерний метод делает необязательный параметр обязательным.

```
<?php

class Base
{
public function foo(int $a = 5) {
echo "Допустимо\n";
}
}

class Extend extends Base
{
function foo(int $a)
{
parent::foo($a);
}
}
```

Результат выполнения приведённого примера в PHP 8 аналогичен:

```
Fatal error: Declaration of Extend::foo(int $a) must be compatible with Base::foo(int $a = 5) in /in/qJXVC on line 13
```

Внимание

Переименование параметра метода в дочернем классе не является несовместимостью сигнатуры. Однако это не рекомендуется, так как приведёт к Error во время выполнения, если используются именованные аргументы.

Пример #14 Ошибка при использовании именованных аргументов и параметров, переименованных в дочернем классе

```
<?php

class A {
public function test($foo, $bar) {}
}

class B extends A {
public function test($a, $b) {}
}

$obj = new B;

// Передача параметров согласно контракту A::test()
$obj->test(foo: "foo", bar: "bar"); // ОШИБКА!
```

Вывод приведённого примера будет похож на:

```
Fatal error: Uncaught Error: Unknown named parameter $foo in /in/XaaeN:14
Stack trace:
#0 {main}
  thrown in /in/XaaeN on line 14
```

::class

Ключевое слово `class` используется для разрешения имени класса. Чтобы получить полное имя класса `ClassName`, используйте `ClassName::class`. Обычно это довольно полезно при работе с классами, использующими пространства имён.

Пример #15 Разрешение имени класса

```
<?php
namespace NS {
class ClassName {
}

echo ClassName::class;
}
?>
```

Результат выполнения приведённого примера:

```
NS\ClassName
```

> **Замечание**:
>
> Разрешение имён класса с использованием `::class` происходит на этапе компиляции. Это означает, что на момент создания строки с именем класса автозагрузки класса не происходит. Как следствие, имена классов раскрываются, даже если класс не существует. Ошибка в этом случае не выдаётся.
>
> **Пример #16 Отсутствует разрешение имени класса**
>
> ```
> <?php
> print Does\Not\Exist::class;
> ?>
> ```
>
> Результат выполнения приведённого примера:
>
> ```
> Does\Not\Exist
> ```

Начиная с PHP 8.0.0, константа `::class` также может использоваться для объектов. Это разрешение происходит во время выполнения, а не во время компиляции. То же самое, что и при вызове get_class() для объекта.

Пример #17 Разрешение имени объекта

```
<?php
```

```
namespace NS {
class ClassName {
}
}
$c = new ClassName();
print $c::class;
?>
```

Результат выполнения приведённого примера:

```
NS\ClassName
```

Методы и свойства Nullsafe

Начиная с PHP 8.0.0, к свойствам и методам можно также обращаться с помощью оператора "nullsafe": `?->`. Оператор nullsafe работает так же, как доступ к свойству или методу, как указано выше, за исключением того, что если разыменование объекта выдаёт **`null`**, то будет возвращён **`null`**, а не выброшено исключение. Если разыменование является частью цепочки, остальная часть цепочки пропускается.

Аналогично заключению каждого обращения в is_null(), но более компактный.

Пример #18 Оператор Nullsafe

```
<?php

// Начиная с PHP 8.0.0, эта строка:
$result = $repository?->getUser(5)?->name;

// Эквивалентна следующему блоку кода:
if (is_null($repository)) {
$result = null;
} else {
$user = $repository->getUser(5);
if (is_null($user)) {
$result = null;
} else {
$result = $user->name;
}
}
?>
```

> **Замечание**:
>
> Оператор nullsafe лучше всего использовать, когда null считается допустимым и ожидаемым значением для возвращаемого свойства или метода. Для индикации ошибки предпочтительнее выбрасывать исключение.

User Contributed Notes 11 notes

635
aaron at thatone dot com ¶
16 years ago
I was confused at first about object assignment, because it's not quite the same as normal assignment or assignment by reference. But I think I've figured out what's going on.

First, think of variables in PHP as data slots. Each one is a name that points to a data slot that can hold a value that is one of the basic data types: a number, a string, a boolean, etc. When you create a reference, you are making a second name that points at the same data slot. When you assign one variable to another, you are copying the contents of one data slot to another data slot.

Now, the trick is that object instances are not like the basic data types. They cannot be held in the data slots directly. Instead, an object's "handle" goes in the data slot. This is an identifier that points at one particular instance of an

obect. So, the object handle, although not directly visible to the programmer, is one of the basic datatypes.

What makes this tricky is that when you take a variable which holds an object handle, and you assign it to another variable, that other variable gets a copy of the same object handle. This means that both variables can change the state of the same object instance. But they are not references, so if one of the variables is assigned a new value, it does not affect the other variable.

```
<?php
// Assignment of an object
Class Object{
public $foo="bar";
};

$objectVar = new Object();
$reference =& $objectVar;
$assignment = $objectVar

//
// $objectVar --->+---------+
// |(handle1)----+
// $reference --->+---------+ |
// |
// +---------+ |
// $assignment -->|(handle1)----+
// +---------+ |
// |
// v
// Object(1):foo="bar"
//
?>
```

$assignment has a different data slot from $objectVar, but its data slot holds a handle to the same object. This makes it behave in some ways like a reference. If you use the variable $objectVar to change the state of the Object instance, those changes also show up under $assignment, because it is pointing at that same Object instance.

```
<?php
$objectVar->foo = "qux";
print_r( $objectVar );
print_r( $reference );
print_r( $assignment );

//
// $objectVar --->+---------+
// |(handle1)----+
// $reference --->+---------+ |
// |
// +---------+ |
// $assignment -->|(handle1)----+
// +---------+ |
// |
// v
// Object(1):foo="qux"
//
?>
```

But it is not exactly the same as a reference. If you null out $objectVar, you replace the handle in its data slot with NULL. This means that $reference, which points at the same data slot, will also be NULL. But $assignment, which is a different data slot, will still hold its copy of the handle to the Object instance, so it will not be NULL.

```
<?php
$objectVar = null;
print_r($objectVar);
```

```
print_r($reference);
print_r($assignment);

//
// $objectVar --->+---------+
// | NULL |
// $reference --->+---------+
//
// +---------+
// $assignment -->|(handle1)----+
// +---------+ |
// |
// v
// Object(1):foo="qux"
?>
```

up
down
84

kStarbe at gmail point com ¶

6 years ago

You start using :: in second example although the static concept has not been explained. This is not easy to discover when you are starting from the basics.

up
down
129

Doug ¶

13 years ago

What is the difference between $this and self ?

Inside a class definition, $this refers to the current object, while self refers to the current class.

It is necessary to refer to a class element using self ,
and refer to an object element using $this .
Note also how an object variable must be preceded by a keyword in its definition.

The following example illustrates a few cases:

```
<?php
class Classy {

const STAT = 'S' ; // no dollar sign for constants (they are always static)
static $stat = 'Static' ;
public $publ = 'Public' ;
private $priv = 'Private' ;
protected $prot = 'Protected' ;

function __construct( ){ }

public function showMe( ){
print '<br> self::STAT: ' . self::STAT ; // refer to a (static) constant like this
print '<br> self::$stat: ' . self::$stat ; // static variable
print '<br>$this->stat: ' . $this->stat ; // legal, but not what you might think: empty result
print '<br>$this->publ: ' . $this->publ ; // refer to an object variable like this
print '<br>' ;
}
}
$me = new Classy( ) ;
$me->showMe( ) ;

/* Produces this output:
self::STAT: S
```

```
self::$stat: Static
$this->stat:
$this->publ: Public
*/
?>
```

up
down
23

Hayley Watson ¶

6 years ago

```
Class names are case-insensitive:
<?php
class Foo{}
class foo{} //Fatal error.
?>

Any casing can be used to refer to the class
<?php
class bAr{}
$t = new Bar();
$u = new bar();
echo ($t instanceof $u) ? "true" : "false"; // "true"
echo ($t instanceof BAR) ? "true" : "false"; // "true"
echo is_a($u, 'baR') ? "true" : "false"; // "true"
?>

But the case used when the class was defined is preserved as "canonical":
<?php
echo get_class($t); // "bAr"
?>

And, as always, "case-insensitivity" only applies to ASCII.
<?php
class пасха{}
class Пасха{} // valid
$p = new ПАСХА(); // Uncaught warning.
?>
```

up
down
65

wbcarts at juno dot com ¶

15 years ago

```
CLASSES and OBJECTS that represent the "Ideal World"

Wouldn't it be great to get the lawn mowed by saying $son->mowLawn()? Assuming the function mowLawn() is defined, and you have a son that doesn't throw errors, the lawn will be mowed.

In the following example; let objects of type Line3D measure their own length in 3-dimensional space. Why should I or PHP have to provide another method from outside this class to calculate length, when the class itself holds all the neccessary data and has the education to make the calculation for itself?

<?php

/*
* Point3D.php
*
* Represents one locaton or position in 3-dimensional space
* using an (x, y, z) coordinate system.
*/
class Point3D
{
public $x;
```

```
public $y;
public $z; // the x coordinate of this Point.

/*
* use the x and y variables inherited from Point.php.
*/
public function __construct($xCoord=0, $yCoord=0, $zCoord=0)
{
$this->x = $xCoord;
$this->y = $yCoord;
$this->z = $zCoord;
}

/*
* the (String) representation of this Point as "Point3D(x, y, z)".
*/
public function __toString()
{
return 'Point3D(x=' . $this->x . ', y=' . $this->y . ', z=' . $this->z . ')';
}
}

/*
* Line3D.php
*
* Represents one Line in 3-dimensional space using two Point3D objects.
*/
class Line3D
{
$start;
$end;

public function __construct($xCoord1=0, $yCoord1=0, $zCoord1=0, $xCoord2=1, $yCoord2=1, $zCoord2=1)
{
$this->start = new Point3D($xCoord1, $yCoord1, $zCoord1);
$this->end = new Point3D($xCoord2, $yCoord2, $zCoord2);
}

/*
* calculate the length of this Line in 3-dimensional space.
*/
public function getLength()
{
return sqrt(
pow($this->start->x - $this->end->x, 2) +
pow($this->start->y - $this->end->y, 2) +
pow($this->start->z - $this->end->z, 2)
);
}

/*
* The (String) representation of this Line as "Line3D[start, end, length]".
*/
public function __toString()
{
return 'Line3D[start=' . $this->start .
', end=' . $this->end .
', length=' . $this->getLength() . ']';
}
}

/*
```

```
* create and display objects of type Line3D.
*/
echo '<p>' . (new Line3D()) . "</p>\n";
echo '<p>' . (new Line3D(0, 0, 0, 100, 100, 0)) . "</p>\n";
echo '<p>' . (new Line3D(0, 0, 0, 100, 100, 100)) . "</p>\n";

?>

<-- The results look like this -->

Line3D[start=Point3D(x=0, y=0, z=0), end=Point3D(x=1, y=1, z=1), length=1.73205080757]

Line3D[start=Point3D(x=0, y=0, z=0), end=Point3D(x=100, y=100, z=0), length=141.421356237]

Line3D[start=Point3D(x=0, y=0, z=0), end=Point3D(x=100, y=100, z=100), length=173.205080757]

My absolute favorite thing about OOP is that "good" objects keep themselves in check. I mean really, it's the exact same thing in reality... like, if you hire a plumber to fix your kitchen sink, wouldn't you expect him to figure out the best plan of attack? Wouldn't he dislike the fact that you want to control the whole job? Wouldn't you expect him to not give you additional problems? And for god's sake, it is too much to ask that he cleans up before he leaves?

I say, design your classes well, so they can do their jobs uninterrupted... who like bad news? And, if your classes and objects are well defined, educated, and have all the necessary data to work on (like the examples above do), you won't have to micro-manage the whole program from outside of the class. In other words... create an object, and LET IT RIP!
```

up
down
28

moty66 at gmail dot com ¶

14 years ago

```
I hope that this will help to understand how to work with static variables inside a class

<?php

class a {

public static $foo = 'I am foo';
public $bar = 'I am bar';

public static function getFoo() { echo self::$foo; }
public static function setFoo() { self::$foo = 'I am a new foo'; }
public function getBar() { echo $this->bar; }
}

$ob = new a();
a::getFoo(); // output: I am foo
$ob->getFoo(); // output: I am foo
//a::getBar(); // fatal error: using $this not in object context
$ob->getBar(); // output: I am bar
// If you keep $bar non static this will work
// but if bar was static, then var_dump($this->bar) will output null

// unset($ob);
a::setFoo(); // The same effect as if you called $ob->setFoo(); because $foo is static
$ob = new a(); // This will have no effects on $foo
$ob->getFoo(); // output: I am a new foo

?>

Regards
Motaz Abuthiab
```

up
down

4

pawel dot zimnowodzki at gmail dot com ¶

1 year ago

Although there is no null-safe operator for not existed array keys I found workaround for it: ($array['not_existed_key'] ?? null)?->methodName()

up

down

37

Notes on stdClass ¶

14 years ago

stdClass is the default PHP object. stdClass has no properties, methods or parent. It does not support magic methods, and implements no interfaces.

When you cast a scalar or array as Object, you get an instance of stdClass. You can use stdClass whenever you need a generic object instance.

```
<?php
// ways of creating stdClass instances
$x = new stdClass;
$y = (object) null; // same as above
$z = (object) 'a'; // creates property 'scalar' = 'a'
$a = (object) array('property1' => 1, 'property2' => 'b');
?>
```

stdClass is NOT a base class! PHP classes do not automatically inherit from any class. All classes are standalone, unless they explicitly extend another class. PHP differs from many object-oriented languages in this respect.

```
<?php
// CTest does not derive from stdClass
class CTest {
public $property1;
}
$t = new CTest;
var_dump($t instanceof stdClass); // false
var_dump(is_subclass_of($t, 'stdClass')); // false
echo get_class($t) . "\n"; // 'CTest'
echo get_parent_class($t) . "\n"; // false (no parent)
?>
```

You cannot define a class named 'stdClass' in your code. That name is already used by the system. You can define a class named 'Object'.

You could define a class that extends stdClass, but you would get no benefit, as stdClass does nothing.

(tested on PHP 5.2.8)

up

down

1

johannes dot kingma at gmail dot com ¶

2 years ago

BEWARE!

Like Hayley Watson pointed out class names are not case sensitive.

```
<?php
class Foo{}
class foo{} // Fatal error: Cannot declare class foo, because the name is already in use
?>
```

As well as

```
<?php
class BAR{}
$bar = new Bar();
echo get_class($bar);
?>
```

Is perfectly fine and will return 'BAR'.

This has implications on autoloading classes though. The standard spl_autoload function will strtolower the class name to cope with case in-sensitiveness and thus the class BAR can only be found if the file name is bar.php (or another variety if an extension was registered with spl_autoload_extensions();) not BAR.php for a case sensitive file and operating system like linux. Windows file system is case sensitive but the OS is not and there for autoloading BAR.php will work.

up
down
4

Anonymous ¶
5 years ago

At first I was also confused by the assignment vs referencing but here's how I was finally able to get my head around it. This is another example which is somewhat similar to one of the comments but can be helpful to those who did not understand the first example. Imagine object instances as rooms where you can store and manipulate your properties and functions. The variable that contains the object simply holds 'a key' to this room and thus access to the object. When you assign this variable to another new variable, what you are doing is you're making a copy of the key and giving it to this new variable. That means these two variable now have access to the same 'room' (object) and can thus get in and manipulate the values. However, when you create a reference, what you doing is you're making the variables SHARE the same key. They both have access to the room. If one of the variable is given a new key, then the key that they are sharing is replaced and they now share a new different key. This does not affect the other variable with a copy of the old key...that variable still has access to the first room

up
down
14

Jeffrey ¶
15 years ago

A PHP Class can be used for several things, but at the most basic level, you'll use classes to "organize and deal with like-minded data". Here's what I mean by "organizing like-minded data". First, start with unorganized data.

```
<?php
$customer_name;
$item_name;
$item_price;
$customer_address;
$item_qty;
$item_total;
?>
```

Now to organize the data into PHP classes:

```
<?php
class Customer {
$name; // same as $customer_name
$address; // same as $customer_address
}

class Item {
$name; // same as $item_name
$price; // same as $item_price
$qty; // same as $item_qty
$total; // same as $item_total
}
?>
```

Now here's what I mean by "dealing" with the data. Note: The data is already organized, so that in itself makes writing new functions extremely easy.

```
<?php
class Customer {
public $name, $address; // the data for this class...
```

```
// function to deal with user-input / validation
// function to build string for output
// function to write -> database
// function to read <- database
// etc, etc
}

class Item {
public $name, $price, $qty, $total; // the data for this class...

// function to calculate total
// function to format numbers
// function to deal with user-input / validation
// function to build string for output
// function to write -> database
// function to read <- database
// etc, etc
}
?>
```

Imagination that each function you write only calls the bits of data in that class. Some functions may access all the data, while other functions may only access one piece of data. If each function revolves around the data inside, then you have created a good class.

- Классы и объекты
  - Введение
  - Основы
  - Свойства
  - Константы классов
  - Автоматическая загрузка классов
  - Конструкторы и деструкторы
  - Область видимости
  - Наследование
  - Оператор разрешения области видимости (::)
  - Ключевое слово static
  - Абстрактные классы
  - Интерфейсы объектов
  - Трейты
  - Анонимные классы
  - Перегрузка
  - Итераторы объектов
  - Магические методы
  - Ключевое слово final
  - Клонирование объектов
  - Сравнение объектов
  - Позднее статическое связывание
  - Объекты и ссылки
  - Сериализация объектов
  - Ковариантность и контравариантность
  - Журнал изменений ООП