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

Область видимости »
« Автоматическая загрузка классов

- Руководство по PHP
- Справочник языка
- Классы и объекты

Change language: Russian

Конструкторы и деструкторы

Конструктор

__construct(mixed ...$values = ""): void

PHP разрешает разработчикам объявлять для классов методы-конструкторы. Метод-конструктор будет вызван на каждом вновь созданном экземпляре класса. Поэтому объявление метода-конструктора удобно для инициализации того, что может потребоваться объекту в начале работы.

> **Замечание**: Конструкторы, которые определены в классах-родителях, не вызываются автоматически, если дочерний класс определяет свой конструктор. Чтобы запустить конструктор родительского класса, необходимо вызвать его внутри конструктора дочернего класса — **parent::__construct()**. Если в дочернем классе не определён конструктор, то он будет унаследован от родительского класса как обычный метод (если родительский конструктор не был определён как приватный).

Пример #1 Конструкторы при наследовании

```
<?php
class BaseClass {
function __construct() {
print "Конструктор класса BaseClass\n";
}
}

class SubClass extends BaseClass {
function __construct() {
parent::__construct();
print "Конструктор класса SubClass\n";
}
}

class OtherSubClass extends BaseClass {
// наследует конструктор класса BaseClass
}

// Конструктор класса BaseClass
$obj = new BaseClass();

// Конструктор класса BaseClass
// Конструктор класса SubClass
$obj = new SubClass();

// Конструктор класса BaseClass
$obj = new OtherSubClass();
?>
```

В отличие от других методов, метод __construct() освобождается от обычных правил совместимости сигнатуры при наследовании.

Конструкторы — это обычные методы, которые вызываются при инстанциировании объектов, которые их содержат, или объектов дочерних классов без конструктора. Поэтому в них может быть определено произвольное количество аргументов, которые можно объявить обязательными, типизированными, со значением по умолчанию. Аргументы конструктора указываются в круглых скобках после имени класса.

Пример #2 Объявление аргументов в конструкторах

```
<?php
class Point {
protected int $x;
protected int $y;
```

```
public function __construct(int $x, int $y = 0) {
$this->x = $x;
$this->y = $y;
}
}

// Передаём оба параметра.
$p1 = new Point(4, 5);
// Передаём только обязательные параметры. Для переменной $y установлено значение по умолчанию 0.
$p2 = new Point(4);
// Вызываем с именованными параметрами (начиная с PHP 8.0):
$p3 = new Point(y: 5, x: 4);
?>
```

Если у класса нет конструктора, или его конструктор не содержит обязательных параметров, скобки после имени класса можно не писать.

Конструкторы в старом стиле

До PHP 8.0.0 классы в глобальном пространстве имён будут интерпретировать названный именем класса метод как конструктор старого стиля. Этот синтаксис устарел и будет вызывать ошибку уровня **`E_DEPRECATED`**, но всё равно эти методы будут вызываться в качестве конструктора. Если в классе присутствуют и метод __construct(), и метод с именем класса, то в качестве конструктора будет вызван метод __construct().

Начиная с PHP 8.0.0 для классов внутри пространства имён и для всех классов метод, названный по имени класса, будет проигнорирован.

В новом коде всегда используют метод __construct().

Определение свойств объекта в конструкторе

Начиная с PHP 8.0.0 через параметры конструктора можно устанавливать свойства объекта. Это распространённая практика — присваивать свойствам объекта значения только за счёт переданных в конструктор параметров. Определение свойств класса в конструкторе значительно сокращает количество шаблонного кода для такого случая. Пример выше можно будет переписать как показано ниже:

Пример #3 Использование определения свойств в конструкторе

```
<?php
class Point {
public function __construct(protected int $x, protected int $y = 0) {
}
}
```

Если декларация аргумента конструктора включает модификатор, PHP интерпретирует его одновременно и как аргумент конструктора, и как свойство объекта, и автоматически установит свойству значение, переданное в конструктор. При этом, если не требуется дополнительной логики, тело конструктора можно оставить пустым. Код конструктора выполнится после того, как значения аргументов будут присвоены параметрам функции или свойствам класса.

Не все передаваемые в конструктор аргументы обязаны быть продвинутыми (устанавливающими свойства объекта). Продвинутые и обычные аргументы можно указывать и сопоставлять в любом порядке. Продвинутые аргументы не влияют на код, исполняемый в конструкторе.

> **Замечание**:
>
> Указать модификатор области видимости (`public`, `protected` или `private`) — это наиболее вероятный способ применить продвинутую установку свойств, но любой другой модификатор (например, `readonly`) даст такой же эффект.

> **Замечание**:
>
> Нельзя указывать свойствам объекта тип callable. Это связано с неоднозначностью, которую они представляют для движка PHP. Поэтому и аргументам конструктора, которые устанавливают классу свойства, также нельзя указывать тип callable. Любые другие декларации типов допустимы.

Замечание:

Поскольку при разборе кода PHP трансформирует синтаксический сахар продвинутых свойств в обычное декларирование свойств конструктора с теми же модификаторами видимости и типом данных, которые указаны в продвинутом аргументе, а также присваивает значение продвинутого аргумента и параметру функции, и свойству класса, все без исключения ограничения на определение аргументов конструктора будут применены как к параметрам функции, так и к свойствам класса.

Замечание:

Атрибуты, установленные для аргумента в продвинутом конструкторе, будут реплицированы как на аргумент, так и на свойство класса. Значение по умолчанию для аргумента в продвинутом конструкторе распространяется только на аргумент, а не свойство.

New в инициализации класса

С PHP 8.1.0 объекты можно присваивать в качестве значений параметров по умолчанию, статических переменных и глобальных констант, а также в аргументах атрибутов. Объекты также можно передавать в функцию define().

Замечание:

При этом динамические или нестроковые имена классов или анонимных классов не разрешены. Использовать распаковку аргументов не разрешено. Неподдерживаемые выражения в качестве аргументов не разрешены.

Пример #4 Пример использования new в инициализации класса

```
<?php
// Всё допустимо:
static $x = new Foo;
const C = new Foo;

function test($param = new Foo) {}

#[AnAttribute(new Foo)]
class Test {
public function __construct(
public $prop = new Foo,
) {}
}
// Всё недопустимо (ошибка во времени компиляции):
function test(
$a = new (CLASS_NAME_CONSTANT)(), // динамическое имя класса
$b = new class {}, // анонимный класс
$c = new A(...[]), // распаковка аргументов
$d = new B($abc), // неподдерживаемое постоянное выражение
) {}
?>
```

Статические методы создания объекта

PHP поддерживает только один конструктор для класса. Однако бывает так, что нужно создавать разные объекты для разных входных данных. Рекомендуемый способ — использовать статические методы как обёртки над конструктором.

Пример #5 Использование статических методов для создания объектов

```
<?php
class Product {

private ?int $id;
private ?string $name;

private function __construct(?int $id = null, ?string $name = null) {
$this->id = $id;
```

```
$this->name = $name;
}

public static function fromBasicData(int $id, string $name): static {
$new = new static($id, $name);
return $new;
}

public static function fromJson(string $json): static {
$data = json_decode($json);
return new static($data['id'], $data['name']);
}

public static function fromXml(string $xml): static {
// Пользовательская логика.
$data = convert_xml_to_array($xml);
$new = new static();
$new->id = $data['id'];
$new->name = $data['name'];
return $new;
}
}

$p1 = Product::fromBasicData(5, 'Widget');
$p2 = Product::fromJson($some_json_string);
$p3 = Product::fromXml($some_xml_string);
```

Конструктор можно сделать закрытым или защищённым для предотвращения его прямого вызова. Тогда объект класса можно будет создать только вызовом статического метода. Поскольку это методы того же класса, у них есть доступ ко всем его скрытым методам, даже если они относятся к разным экземплярам класса. Закрытый конструктор опционален, и может быть объявлен или не иметь смысла в разных ситуациях.

В примере выше три публичных статических метода показывают разные способы создания экземпляра объекта.

- Метод `fromBasicData()` принимает явные параметры, создаёт экземпляр класса через конструктор и возвращает объект.
- Метод `fromJson()` принимает JSON-строку, производит над ней преобразования, извлекает необходимые для создания объекта данные и, так же как и предыдущий метод, вызывает конструктор и возвращает созданный объект.
- Метод `fromXml()` принимает XML-строку, извлекает нужные данные и, так как в конструкторе нет обязательных параметров, вызывает его без них. После этого, так как ему доступны скрытые свойства, он присваивает им значения напрямую. После чего возвращает готовый объект.

Во всех трёх случаях ключевое слово `static` транслируется в имя класса, в котором этот код вызывается. В нашем случае — в класс `Product`.

Деструкторы

__destruct(): void

PHP наследует концепцию деструктора, аналогичную другим объектно-ориентированным языкам, например, C++. Деструктор будет вызван при освобождении всех ссылок на объект или при завершении скрипта (порядок выполнения деструкторов не гарантируется).

Пример #6 Пример использования деструктора

```
<?php
class MyDestructableClass
{
function __construct() {
print "Конструктор\n";
}

function __destruct() {
```

```
print "Уничтожается " . __CLASS__ . "\n";
}
}

$obj = new MyDestructableClass();
```

Как и конструкторы, деструкторы, объявленные в родительском классе, не будут вызываться автоматически. Чтобы запустить деструктор родительского класса, необходимо вызвать **parent::__destruct()** в теле деструктора дочернего класса. Аналогично конструкторам, дочерний класс, в котором не определен деструктор, наследует его из родительского класса.

Деструктор будет вызван даже если скрипт был остановлен функцией exit(). Вызов функции exit() в деструкторе предотвратит запуск всех остальных процедур завершения работы.

> **Замечание**:
>
> Деструкторы, вызываемые при завершении скрипта, вызываются после отправки HTTP-заголовков. На этапе завершения работы скрипта рабочая директория SAPI (например, в Apache) может измениться.

> **Замечание**:
>
> Попытка выбросить исключение из деструктора (вызванного во время завершения работы скрипта) вызовет фатальную ошибку.

User Contributed Notes 15 notes

150
david dot scourfield at llynfi dot co dot uk ¶
12 years ago

Be aware of potential memory leaks caused by circular references within objects. The PHP manual states "[t]he destructor method will be called as soon as all references to a particular object are removed" and this is precisely true: if two objects reference each other (or even if one object has a field that points to itself as in $this->foo = $this) then this reference will prevent the destructor being called even when there are no other references to the object at all. The programmer can no longer access the objects, but they still stay in memory.

Consider the following example:

```
<?php

header("Content-type: text/plain");

class Foo {

/**
* An indentifier
* @var string
*/
private $name;
/**
* A reference to another Foo object
* @var Foo
*/
private $link;

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

public function setLink(Foo $link){
$this->link = $link;
```

```
}

public function __destruct() {
echo 'Destroying: ', $this->name, PHP_EOL;
}
}

// create two Foo objects:
$foo = new Foo('Foo 1');
$bar = new Foo('Foo 2');

// make them point to each other
$foo->setLink($bar);
$bar->setLink($foo);

// destroy the global references to them
$foo = null;
$bar = null;

// we now have no way to access Foo 1 or Foo 2, so they OUGHT to be __destruct()ed
// but they are not, so we get a memory leak as they are still in memory.
//
// Uncomment the next line to see the difference when explicitly calling the GC:
// gc_collect_cycles();
//
// see also: http://www.php.net/manual/en/features.gc.php
//

// create two more Foo objects, but DO NOT set their internal Foo references
// so nothing except the vars $foo and $bar point to them:
$foo = new Foo('Foo 3');
$bar = new Foo('Foo 4');

// destroy the global references to them
$foo = null;
$bar = null;

// we now have no way to access Foo 3 or Foo 4 and as there are no more references
// to them anywhere, their __destruct() methods are automatically called here,
// BEFORE the next line is executed:

echo 'End of script', PHP_EOL;

?>
```

This will output:

```
Destroying: Foo 3
Destroying: Foo 4
End of script
Destroying: Foo 1
Destroying: Foo 2
```

But if we uncomment the gc_collect_cycles(); function call in the middle of the script, we get:

```
Destroying: Foo 2
Destroying: Foo 1
Destroying: Foo 3
Destroying: Foo 4
End of script
```

As may be desired.

NOTE: calling gc_collect_cycles() does have a speed overhead, so only use it if you feel you need to.

up
down
28

domger at freenet dot de ¶

6 years ago

The __destruct magic method must be public.

```
public function __destruct()
{
;
}
```

The method will automatically be called externally to the instance. Declaring __destruct as protected or private will result in a warning and the magic method will not be called.

Note: In PHP 5.3.10 i saw strange side effects while some Destructors were declared as protected.

up
down
24

spleen ¶

15 years ago

It's always the easy things that get you -

Being new to OOP, it took me quite a while to figure out that there are TWO underscores in front of the word __construct.

It is __construct
Not _construct

Extremely obvious once you figure it out, but it can be sooo frustrating until you do.

I spent quite a bit of needless time debugging working code.

I even thought about it a few times, thinking it looked a little long in the examples, but at the time that just seemed silly(always thinking "oh somebody would have made that clear if it weren't just a regular underscore...")

All the manuals I looked at, all the tuturials I read, all the examples I browsed through - not once did anybody mention this!

(please don't tell me it's explained somewhere on this page and I just missed it, you'll only add to my pain.)

I hope this helps somebody else!

up
down
7

iwwp at outlook dot com ¶

4 years ago

To better understand the __destrust method:

```
class A {
protected $id;

public function __construct($id)
{
$this->id = $id;
echo "construct {$this->id}\n";
}

public function __destruct()
{
```

```
echo "destruct {$this->id}\n";
}
}

$a = new A(1);
echo "-------------\n";
$aa = new A(2);
echo "=============\n";

The output content:

construct 1
-------------
construct 2
=============
destruct 2
destruct 1
```

up
down
10
prieler at abm dot at ¶
16 years ago

i have written a quick example about the order of destructors and shutdown functions in php 5.2.1:

```
<?php
class destruction {
var $name;

function destruction($name) {
$this->name = $name;
register_shutdown_function(array(&$this, "shutdown"));
}

function shutdown() {
echo 'shutdown: '.$this->name."\n";
}

function __destruct() {
echo 'destruct: '.$this->name."\n";
}
}

$a = new destruction('a: global 1');

function test() {
$b = new destruction('b: func 1');
$c = new destruction('c: func 2');
}
test();

$d = new destruction('d: global 2');

?>

this will output:
shutdown: a: global 1
shutdown: b: func 1
shutdown: c: func 2
shutdown: d: global 2
destruct: b: func 1
destruct: c: func 2
destruct: d: global 2
```

```
destruct: a: global 1

conclusions:
destructors are always called on script end.
destructors are called in order of their "context": first functions, then global objects
objects in function context are deleted in order as they are set (older objects first).
objects in global context are deleted in reverse order (older objects last)

shutdown functions are called before the destructors.
shutdown functions are called in there "register" order. ;)

regards, J
```

up
down
2

mmulej at gmail dot com ¶

1 year ago

<Double post> I can't edit my previous note to elaborate on modifiers. Please excuse me.

If both parent and child classes have a method with the same name defined, and it is called in parent's constructor, using `parent::__construct()` will call the method in the child.

```
<?php

class A {
public function __construct() {
$this->method();
}
public function method() {
echo 'A' . PHP_EOL;
}
}
class B extends A {
public function __construct() {
parent::__construct();
}
}
class C extends A {
public function __construct() {
parent::__construct();
}
public function method() {
echo 'C' . PHP_EOL;
}
}
$b = new B; // A
$c = new C; // C

?>
```

In this example both A::method and C::method are public.

You may change A::method to protected, and C::method to protected or public and it will still work the same.

If however you set A::method as private, it doesn't matter whether C::method is private, protected or public. Both $b and $c will echo 'A'.

up
down
7

Per Persson ¶

11 years ago

As of PHP 5.3.10 destructors are not run on shutdown caused by fatal errors.

For example:

```
<?php
class Logger
{
protected $rows = array();

public function __destruct()
{
$this->save();
}

public function log($row)
{
$this->rows[] = $row;
}

public function save()
{
echo '<ul>';
foreach ($this->rows as $row)
{
echo '<li>', $row, '</li>';
}
echo '</ul>';
}
}

$logger = new Logger;
$logger->log('Before');

$nonset->foo();

$logger->log('After');
?>
```

Without the $nonset->foo(); line, Before and After will both be printed, but with the line neither will be printed.

One can however register the destructor or another method as a shutdown function:

```
<?php
class Logger
{
protected $rows = array();

public function __construct()
{
register_shutdown_function(array($this, '__destruct'));
}

public function __destruct()
{
$this->save();
}

public function log($row)
{
$this->rows[] = $row;
}

public function save()
{
echo '<ul>';
```

```
foreach ($this->rows as $row)
{
echo '<li>', $row, '</li>';
}
echo '</ul>';
}
}

$logger = new Logger;
$logger->log('Before');

$nonset->foo();

$logger->log('After');
?>
```

Now Before will be printed, but not After, so you can see that a shutdown occurred after Before.

up
down
4

Yousef Ismaeil cliprz[At]gmail[Dot]com ¶

10 years ago

```
<?php

/**
* a funny example Mobile class
*
* @author Yousef Ismaeil Cliprz[At]gmail[Dot]com
*/

class Mobile {

/**
* Some device properties
*
* @var string
* @access public
*/
public $deviceName,$deviceVersion,$deviceColor;

/**
* Set some values for Mobile::properties
*
* @param string device name
* @param string device version
* @param string device color
*/
public function __construct ($name,$version,$color) {
$this->deviceName = $name;
$this->deviceVersion = $version;
$this->deviceColor = $color;
echo "The ".__CLASS__." class is stratup.<br /><br />";
}

/**
* Some Output
*
* @access public
*/
public function printOut () {
echo 'I have a '.$this->deviceName
.' version '.$this->deviceVersion
.' my device color is : '.$this->deviceColor;
```

```
}

/**
* Umm only for example we will remove Mobile::$deviceName Hum not unset only to check how __destruct working
*
* @access public
*/
public function __destruct () {
$this->deviceName = 'Removed';
echo '<br /><br />Dumpping Mobile::deviceName to make sure its removed, Olay :';
var_dump($this->deviceName);
echo "<br />The ".__CLASS__." class is shutdown.";
}

}

// Oh ya instance
$mob = new Mobile('iPhone','5','Black');

// print output
$mob->printOut();

?>

The Mobile class is stratup.

I have a iPhone version 5 my device color is : Black

Dumpping Mobile::deviceName to make sure its removed, Olay :
string 'Removed' (length=7)

The Mobile class is shutdown.
```

up
down
2

bolshun at mail dot ru ¶

15 years ago

Ensuring that instance of some class will be available in destructor of some other class is easy: just keep a reference to that instance in this other class.

up
down
0

Hayley Watson ¶

5 months ago

There are other advantages to using static factory methods to wrap object construction instead of bare constructor calls.

As well as allowing for different methods to use in different scenarios, with more relevant names both for the methods and the parameters and without the constructor having to handle different sets of arguments of different types:

* You can do all your input validation before attempting to construct the object.
* The object itself can bypass that input validation when constructing new instances of its own class, since you can ensure that it knows what it's doing.
* With input validation/preprocessing moved to the factory methods, the constructor itself can often be reduced to "set these properties to these arguments", meaning the constructor promotion syntax becomes more useful.
* Having been hidden away from users, the constructor's signature can be a bit uglier without becoming a pain for them. Heh.
* Static methods can be lifted and passed around as first class closures, to be called in the normal fashion wherever functions can be called, without the special "new" syntax.
* The factory method need not return a new instance of that exact class. It could return a pre-existing instance that would do the same job as the new one would (especially useful in the case of immutable "value type" objects by reducing duplication); or a simpler or more specific subclass to do the job with less overhead than a more generic instance of the original class. Returning a subclass means LSP still holds.

up
down
2
Jonathon Hibbard ¶
14 years ago
Please be aware of when using __destruct() in which you are unsetting variables...

Consider the following code:

```
<?php
class my_class {
public $error_reporting = false;

function __construct($error_reporting = false) {
$this->error_reporting = $error_reporting;
}

function __destruct() {
if($this->error_reporting === true) $this->show_report();
unset($this->error_reporting);
}
?>
```

The above will result in an error:
Notice: Undefined property: my_class::$error_reporting in my_class.php on line 10

It appears as though the variable will be unset BEFORE it actually can execute the if statement. Removing the unset will fix this. It's not needed anyways as PHP will release everything anyways, but just in case you run across this, you know why ;)
up
down
1
david at synatree dot com ¶
16 years ago
When a script is in the process of die()ing, you can't count on the order in which __destruct() will be called.

For a script I have been working on, I wanted to do transparent low-level encryption of any outgoing data. To accomplish this, I used a global singleton class configured like this:

```
class EncryptedComms
{
private $C;
private $objs = array();
private static $_me;

public static function destroyAfter(&$obj)
{
self::getInstance()->objs[] =& $obj;
/*
Hopefully by forcing a reference to another object to exist
inside this class, the referenced object will need to be destroyed
before garbage collection can occur on this object. This will force
this object's destruct method to be fired AFTER the destructors of
all the objects referenced here.
*/
}
public function __construct($key)
{
$this->C = new SimpleCrypt($key);
ob_start(array($this,'getBuffer'));
}
public static function &getInstance($key=NULL)
{
```

```
if(!self::$_me && $key)
self::$_me = new EncryptedComms($key);
else
return self::$_me;
}

public function __destruct()
{
ob_end_flush();
}

public function getBuffer($str)
{
return $this->C->encrypt($str);
}

}
```

In this example, I tried to register other objects to always be destroyed just before this object. Like this:

```
class A
{

public function __construct()
{
EncryptedComms::destroyAfter($this);
}
}
```

One would think that the references to the objects contained in the singleton would be destroyed first, but this is not the case. In fact, this won't work even if you reverse the paradigm and store a reference to EncryptedComms in every object you'd like to be destroyed before it.

In short, when a script die()s, there doesn't seem to be any way to predict the order in which the destructors will fire.

up
down
0

Reza Mahjourian ¶

17 years ago

Peter has suggested using static methods to compensate for unavailability of multiple constructors in PHP. This works fine for most purposes, but if you have a class hierarchy and want to delegate parts of initialization to the parent class, you can no longer use this scheme. It is because unlike constructors, in a static method you need to do the instantiation yourself. So if you call the parent static method, you will get an object of parent type which you can't continue to initialize with derived class fields.

Imagine you have an Employee class and a derived HourlyEmployee class and you want to be able to construct these objects out of some XML input too.

```
<?php
class Employee {
public function __construct($inName) {
$this->name = $inName;
}

public static function constructFromDom($inDom)
{
$name = $inDom->name;
return new Employee($name);
}

private $name;
}
```

```
class HourlyEmployee extends Employee {
public function __construct($inName, $inHourlyRate) {
parent::__construct($inName);
$this->hourlyRate = $inHourlyRate;
}

public static function constructFromDom($inDom)
{
// can't call parent::constructFromDom($inDom)
// need to do all the work here again
$name = $inDom->name; // increased coupling
$hourlyRate = $inDom->hourlyrate;
return new EmployeeHourly($name, $hourlyRate);
}

private $hourlyRate;
}
?>
```

The only solution is to merge the two constructors in one by adding an optional $inDom parameter to every constructor.

up
down
-3

ziggy at start dot dust ¶
1 year ago

Please note that constructor argument promotion is kind of half-baked (at least as of 8.1 and it does not look to be changed in 8.2) and you are not allowed to reuse promoted argument with other promoted arguments. For example having "old style" constructor:

```
<?php
public function __construct(protected string $val, protected Foo $foo = null) {
$this->val = $val;
$this->foo = $foo ?? new Foo($val);
}
?>
```

you will not be able to use argument promotion like this:

```
<?php
public function __construct(protected string $val, protected Foo $foo = new Foo($val)) {}
?>
```

nor

```
<?php
public function __construct(protected string $val, protected Foo $foo = new Foo($this->val)) {}
?>
```

as in both cases you will face "PHP Fatal error: Constant expression contains invalid operations".

up
down
-5

instatiendaweb at gmail dot com ¶
2 years ago

```
/**
* Haciendo una prueba con dos clases y dos destructores
* La prueba consta de acceder a la variable global del primer objeto en el segundo
* objeto el destructor 2
* Primera clase ==> $GLOBALS['obj']
* SEgunda clase ==> $GLOBALS['obj2']
* Se ejecuta construct y todo el codigo....
```

```
* Primer destruct borra el objeto y lo hace null
* Tratamos de acceder a $GLOBALS['obj'] en el segundo destruct pero
* ya no esta es un objeto null
* Warning: Undefined array key "obj" in...
*/

class MyDestructableClass{
public $parametro;

function __construct($parametro) {
echo("<div class=\"div\">"), "Construyendo ",__CLASS__ , ("</div>");
escribir::verifacionnota($this ,'Antes de guardar la variable ');
$this->parametro = $parametro;
escribir::verifacionnota($this ,'Despues de guardar la variable ');
}



function __destruct() {
escribir::linea(5); //Separador
echo("<div class=\"div\">"), "Destruyendo " , __CLASS__ , ("</div>");
escribir::verifacionnota($this ,'Antes de borrar la variable ');
unset($this->parametro);
escribir::verifacionnota($this ,'Despues de borrar la variable ');

// unset($GLOBALS[$this]);
}
}

$obj = new MyDestructableClass('parametroone');
escribir::verifacionnota($obj ,' Verificar la clase MyDestructableClass, no es necesario
borrar la clase porque se ejecuta al final del script ');
escribir::titulosep('Provando ejemplo aqui se puede acceder a la variable global');
escribir::verificacion($GLOBALS['obj']);

class destructora{
function __destruct(){
escribir::titulosep('Sin embargo esta variable muere aqui');
escribir::verificacion($GLOBALS['obj']);
}
}

$obj2 = new destructora();
```


- 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