




- [Downloads](#)
- [Documentation](#)
- [Get Involved](#)
- [Help](#)
- 

[Dutch PHP Conference 2024](#)

## [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

[Итераторы объектов »](#)  
[« Анонимные классы](#)

- [Руководство по PHP](#)
- [Справочник языка](#)
- [Классы и объекты](#)

Change language: Russian

## Перегрузка

Перегрузка в PHP означает возможность динамически «создавать» свойства и методы. Эти динамические сущности обрабатываются с помощью магических методов, которые можно создать в классе для различных видов действий.

Методы перегрузки вызываются при взаимодействии со свойствами или методами, которые не были объявлены или не [видны](#) в текущей области видимости. Далее в этом разделе будут использоваться термины «недоступные свойства» или «недоступные методы» для обозначения этой комбинации объявления и области видимости.

Все методы перегрузки должны быть объявлены как `public`.

### Замечание:

Ни один из аргументов этих магических методов не может быть передан [по ссылке](#).

### Замечание:

Интерпретация «перегрузки» в PHP отличается от большинства объектно-ориентированных языков. Традиционно перегрузка означает возможность иметь несколько одноимённых методов с разным количеством и типами аргументов.

## Перегрузка свойств

```
public __set(string $name, mixed $value): void
public __get(string $name): mixed
public __isset(string $name): bool
public __unset(string $name): void
```

Метод [\\_\\_set\(\)](#) будет выполнен при записи данных в недоступные (защищённые или приватные) или несуществующие свойства.

Метод [\\_\\_get\(\)](#) будет выполнен при чтении данных из недоступных (защищённых или приватных) или несуществующих свойств.

Метод [\\_\\_isset\(\)](#) будет выполнен при использовании [isset\(\)](#) или [empty\(\)](#) на недоступных (защищённых или приватных) или несуществующих свойствах.

Метод [\\_\\_unset\(\)](#) будет выполнен при вызове [unset\(\)](#) на недоступном (защищённом или приватном) или несуществующем свойстве.

Аргумент *\$name* представляет собой имя вызываемого свойства. Метод [\\_\\_set\(\)](#) содержит аргумент *\$value*, представляющий собой значение, которое будет записано в свойство с именем *\$name*.

Перегрузка свойств работает только в контексте объекта. Данные магические методы не будут вызваны в статическом контексте. Поэтому эти методы не должны объявляться [статическими](#). При объявлении любого магического метода как `static` будет выдано предупреждение.

### Замечание:

Возвращаемое значение [\\_\\_set\(\)](#) будет проигнорировано из-за способа обработки в PHP оператора присваивания. Аналогично, [\\_\\_get\(\)](#) никогда не вызывается при объединении присваиваний, например, подобным образом:

```
$a = $obj->b = 8;
```

### Замечание:

PHP не будет вызывать перегруженный метод изнутри того же перегруженного метода. Это означает, что, например, написание `return $this->foo` внутри [\\_\\_get\(\)](#) вернёт `null` и вызовет ошибку уровня `E_WARNING`, если не определено свойство `foo`, вместо того, чтобы вызвать метод [\\_\\_get\(\)](#) во второй раз. Однако методы перегрузки могут неявно вызывать другие методы перегрузки (например, метод [\\_\\_set\(\)](#) вызывает метод [\\_\\_get\(\)](#)).

**Пример #1** Перегрузка свойств с помощью методов [\\_\\_get\(\)](#), [\\_\\_set\(\)](#), [\\_\\_isset\(\)](#) и [\\_\\_unset\(\)](#)

```
class PropertyTest
{
    /** Место хранения перегружаемых данных. */
    private $data = array();

    /** Перегрузка не применяется к объявленным свойствам. */
    public $declared = 1;

    /** Здесь перегрузка будет использована только при доступе вне класса. */
    private $hidden = 2;

    public function __set($name, $value)
    {
        echo "Установка '$name' в '$value'\n";
        $this->data[$name] = $value;
    }

    public function __get($name)
    {
        echo "Получение '$name'\n";
        if (array_key_exists($name, $this->data)) {
            return $this->data[$name];
        }

        $trace = debug_backtrace();
        trigger_error(
            'Неопределённое свойство в __get(): ' . $name .
            ' в файле ' . $trace[0]['file'] .
            ' на строке ' . $trace[0]['line'],
            E_USER_NOTICE);
        return null;
    }

    public function __isset($name)
    {
        echo "Установлено ли '$name'? \n";
        return isset($this->data[$name]);
    }

    public function __unset($name)
    {
        echo "Уничтожение '$name'\n";
        unset($this->data[$name]);
    }

    /** Не магический метод, просто для примера. */
    public function getHidden()
    {
        return $this->hidden;
    }

    echo "<pre>\n";

    $obj = new PropertyTest;

    $obj->a = 1;
    echo $obj->a . "\n\n";

    var_dump(isset($obj->a));
```

```

unset($obj->a);
var_dump(isset($obj->a));
echo "\n";

echo $obj->declared . "\n\n";

echo "Давайте поэкспериментируем с закрытым свойством 'hidden':\n";
echo "Закрытые свойства видны внутри класса, поэтому __get() не используется...\n";
echo $obj->getHidden() . "\n";
echo "Закрытые свойства не видны вне класса, поэтому __get() используется...\n";
echo $obj->hidden . "\n";
?>

```

Результат выполнения приведённого примера:

```

Установка 'a' в '1'
Получение 'a'
1

```

```

Установлено ли 'a'?
bool(true)
Уничтожение 'a'
Установлено ли 'a'?
bool(false)

```

```

1

```

```

Давайте поэкспериментируем с закрытым свойством 'hidden':
Закрытые свойства видны внутри класса, поэтому __get() не используется...
2
Закрытые свойства не видны вне класса, поэтому __get() используется...
Получение 'hidden'

```

Notice: Неопределённое свойство в \_\_get(): hidden в <file> on line 70 in <file> on line 29

## Перегрузка методов

```

public __call(string $name, array $arguments): mixed
public static __callStatic(string $name, array $arguments): mixed

```

[\\_\\_call\(\)](#) запускается при вызове недоступных методов в контексте объекта.

[\\_\\_callStatic\(\)](#) запускается при вызове недоступных методов в статическом контексте.

Аргумент *\$name* представляет собой имя вызываемого метода. Аргумент *\$arguments* представляет собой нумерованный массив, содержащий параметры, переданные в вызываемый метод *\$name*.

### Пример #2 Перегрузка методов с помощью методов [\\_\\_call\(\)](#) и [\\_\\_callStatic\(\)](#)

```

<?php
class MethodTest {
public function __call($name, $arguments) {
// Замечание: значение $name регистрозависимо.
echo "Вызов метода '$name' "
. implode(' ', $arguments). "\n";
}

public static function __callStatic($name, $arguments) {
// Замечание: значение $name регистрозависимо.
echo "Вызов статического метода '$name' "
. implode(' ', $arguments). "\n";
}
}

$obj = new MethodTest;
$obj->runTest('в контексте объекта');

```

```
MethodTest::runTest('в статическом контексте');
?>
```

Результат выполнения приведённого примера:

Вызов метода 'runTest' в контексте объекта  
Вызов статического метода 'runTest' в статическом контексте

[+add a note](#)

## User Contributed Notes 27 notes

[up](#)

[down](#)

358

[theaceofthespade at gmail dot com ¶](#)

**11 years ago**

A word of warning! It may seem obvious, but remember, when deciding whether to use `__get`, `__set`, and `__call` as a way to access the data in your class (as opposed to hard-coding getters and setters), keep in mind that this will prevent any sort of autocomplete, highlighting, or documentation that your ide mite do.

Furthermore, it beyond personal preference when working with other people. Even without an ide, it can be much easier to go through and look at hardcoded member and method definitions in code, than having to sift through code and piece together the method/member names that are assembled in `__get` and `__set`.

If you still decide to use `__get` and `__set` for everything in your class, be sure to include detailed comments and documenting, so that the people you are working with (or the people who inherit the code from you at a later date) don't have to waste time interpreting your code just to be able to use it.

[up](#)

[down](#)

256

[Anonymous ¶](#)

**7 years ago**

First off all, if you read this, please upvote the first comment on this list that states that “overloading” is a bad term for this behaviour. Because it REALLY is a bad name. You’re giving new definition to an already accepted IT-branch terminology.

Second, I concur with all criticism you will read about this functionality. Just as naming it “overloading”, the functionality is also very bad practice. Please don’t use this in a production environment. To be honest, avoid to use it at all. Especially if you are a beginner at PHP. It can make your code react very unexpectedly. In which case you MIGHT be learning invalid coding!

And last, because of `__get`, `__set` and `__call` the following code executes. Which is abnormal behaviour. And can cause a lot of problems/bugs.

```
<?php
```

```
class BadPractice {
// Two real properties
public $DontAllowVariableNameWithTypos = true;
protected $Number = 0;
// One private method
private function veryPrivateMethod() { }
// And three very magic methods that will make everything look inconsistent
// with all you have ever learned about PHP.
public function __get($n) {}
public function __set($n, $v) {}
public function __call($n, $v) {}
}
```

```
// Let's see our BadPractice in a production environment!
```

```
$UnexpectedBehaviour = new BadPractice;
```

```
// No syntax highlighting on most IDE's
$UnexpectedBehaviour->SynTaxHighlighting = false;

// No autocompletion on most IDE's
$UnexpectedBehaviour->AutoCompletion = false;

// Which will lead to problems waiting to happen
$UnexpectedBehaviour->DontAllowVariableNameWithTypos = false; // see if below

// Get, Set and Call anything you want!
$UnexpectedBehaviour->EveryPossibleMethodCallAllowed(true, 'Why Not?');

// And sure, why not use the most illegal property names you can think off
$UnexpectedBehaviour->{'100%Illegal+Names'} = 'allowed';

// This Very confusing syntax seems to allow access to $Number but because of
// the lowered visibility it goes to __set()
$UnexpectedBehaviour->Number = 10;

// We can SEEM to increment it too! (that's really dynamic! :-) NULL++ LMAO
$UnexpectedBehaviour->Number++;

// this ofcourse outputs NULL (through __get) and not the PERHAPS expected 11
var_dump($UnexpectedBehaviour->Number);

// and sure, private method calls LOOK valid now!
// (this goes to __call, so no fatal error)
$UnexpectedBehaviour->veryPrivateMethod();

// Because the previous was __set to false, next expression is true
// if we didn't had __set, the previous assignment would have failed
// then you would have corrected the typho and this code will not have
// been executed. (This can really be a BIG PAIN)
if ($UnexpectedBehaviour->DontAllowVariableNameWithTypos) {
// if this code block would have deleted a file, or do a deletion on
// a database, you could really be VERY SAD for a long time!
$UnexpectedBehaviour->executeStuffYouDontWantHere(true);
}
?>
```

[up](#)

[down](#)

172

[egingell at sisna dot com ¶](#)

**16 years ago**

Small vocabulary note: This is *not* "overloading", this is "overriding".

Overloading: Declaring a function multiple times with a different set of parameters like this:

```
<?php
```

```
function foo($a) {
return $a;
}
```

```
function foo($a, $b) {
return $a + $b;
}
```

```
echo foo(5); // Prints "5"
echo foo(5, 2); // Prints "7"
```

```
?>
```

Overriding: Replacing the parent class's method(s) with a new method by redeclaring it like this:

```
<?php
```

```
class foo {
function new($args) {
// Do something.
}
}

class bar extends foo {
function new($args) {
// Do something different.
}
}
```

```
?>
```

[up](#)

[down](#)

62

[Anonymous ¶](#)

**8 years ago**

Using magic methods, especially `__get()`, `__set()`, and `__call()` will effectively disable autocomplete in most IDEs (eg.: IntelliJSense) for the affected classes.

To overcome this inconvenience, use phpDoc to let the IDE know about these magic methods and properties: `@method`, `@property`, `@property-read`, `@property-write`.

```
/**
 * @property-read name
 * @property-read price
 */
class MyClass
{
private $properties = array('name' => 'IceFruit', 'price' => 2.49)

public function __get($name)
{
return $this->properties($name);
}
}
```

[up](#)

[down](#)

69

[pogregoire##live.fr ¶](#)

**7 years ago**

It is important to understand that encapsulation can be very easily violated in PHP. for example :

```
class Object{

}

$Object = new Object();
$Object->barbarianProperties = 'boom';

var_dump($Object);// object(Object)#1 (1) { ["barbarianProperties"]=> string(7) "boom" }
```

Hence it is possible to add a propertie out form the class definition.

It is then a necessity in order to protect encapsulation to introduce `__set()` in the class :

```
class Objet{
public function __set($name,$value){
throw new Exception ('no');
}
```



}  
[up](#)  
[down](#)  
3  
[johannes dot kingma at gmail dot com ¶](#)  
2 years ago

One interesting use of the `__get` function is property / function colaescence, using the same name for a property and a function.

Example:

```
<?php
class prop_fun {
private $prop = 123;

public function __get( $property ) {
if( property_exists( $this, $property ) ){
return $this-> $property;
}
throw new Exception( "no such property $property." );
}

public function prop() {
return 456;
}
}

$o = new prop_fun();

echo $o-> prop . '<br>' . PHP_EOL;
echo $o-> prop() . '<br>' . PHP_EOL;
?>
```

This will output 123 and 456. This does look like a funny cludge but I used it of a class containing a date type property and function allowing me to write

```
<?php
class date_class {
/** @property int $date */
private $the_date;

public function __get( $property ) {
if( property_exists( $this, $property ) ){
return $this-> $property;
}
throw new Exception( "no such property $property." );
}

public function the_date( $datetime ) {
return strtotime( $datetime, $this-> the_date );
}

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

$date_object = new date_class();

$today = $date_object-> the_date;
$nextyear = $date_object-> the_date("+1 year");

echo date( "d/m/Y", $today) . '<br>';
echo date( "d/m/Y", $nextyear );
?>
```

Which I like because its self documenting properties. I used this in a utility class for user input.

[up](#)

[down](#)

15

[Ant P. ¶](#)

**15 years ago**

Be extra careful when using `__call()`: if you typo a function call somewhere it won't trigger an undefined function error, but get passed to `__call()` instead, possibly causing all sorts of bizarre side effects.

In versions before 5.3 without `__callStatic`, static calls to nonexistent functions also fall through to `__call`!

This caused me hours of confusion, hopefully this comment will save someone else from the same.

[up](#)

[down](#)

24

[navarr at gtaero dot net ¶](#)

**13 years ago**

If you want to make it work more naturally for arrays `$obj->variable[]` etc you'll need to return `__get` by reference.

```
<?php
class Variables
{
public function __construct()
{
if(session_id() === "")
{
session_start();
}
}
public function __set($name,$value)
{
$_SESSION["Variables"][$name] = $value;
}
public function &__get($name)
{
return $_SESSION["Variables"][$name];
}
public function __isset($name)
{
return isset($_SESSION["Variables"][$name]);
}
}
?>
```

[up](#)

[down](#)

1

[turabgarip at gmail dot com ¶](#)

**2 years ago**

I concur that "overloading" is a wrong term for this functionality. But I disagree that this functionality is completely wrong. You can do "bad practice" with right code too.

For example `__call()` is very well applicable to external integration implementations which I am using to relay calls to SOAP methods which doesn't need local implementation. So you don't have to write "empty body" functions. Consider the SOAP service you connect has a "stock update" method. All you have to do is passing product code and stock count to SOAP.

```
<?php
```

```
class Inventory {

public __construct() {
// configure and connect to SOAP service
$this->soap = new SoapClient();
}
```

```

public __call($soapMethod, $params) {
$this->soap->{$soapMethod}(params);
}
}

// Now you can use any SOAP method without needing a wrapper
$stock = new Inventory();
$stock->updatePrice($product_id, 20);
$stock->saveProduct($product_info);

?>

```

Of course you'd need a parameter mapping but it's in my honest opinion a lot better then having a plenty of mirror methods like:

```

<?php

class Inventory {

public function updateStock($product_id, $stock) {
$soapClient->updateStock($product_id, $stock;
}
public function updatePrice($product_id, $price) {
$soapClient->updateStock($product_id, $price;
}
// ...
}

?>

```

[up](#)  
[down](#)

9

[gabe at fijiwebdesign dot com ¶](#)

**9 years ago**

Note that you can enable "overloading" on a class instance at runtime for an existing property by unset()ing that property.

eg:

```

<?php
class Test {

public $property1;

public function __get($name)
{
return "Get called for " . get_class($this) . "->\$$name \n";
}

}

?>

```

The public property \$property1 can be unset() so that it can be dynamically handled via \_\_get().

```

<?php
$Test = new Test();
unset($Test->property1); // enable overloading
echo $Test->property1; // Get called for Test->$property1
?>

```

Useful if you want to proxy or lazy load properties yet want to have documentation and visibility in the code and

debugging compared to `__get()`, `__isset()`, `__set()` on non-existent inaccessible properties.

[up](#)

[down](#)

11

[php at lanar dot com dot au ¶](#)

**13 years ago**

Note that `__isset` is not called on chained checks.

If `isset( $x->a->b )` is executed where `$x` is a class with `__isset()` declared, `__isset()` is not called.

```
<?php
```

```
class demo
{
    var $id ;
    function __construct( $id = 'who knows' )
    {
        $this->id = $id ;
    }
    function __get( $prop )
    {
        echo "\n", __FILE__, ':', __LINE__, ' ', __METHOD__, '(', $prop, ') instance ', $this->id ;
        return new demo( 'autocreated' ) ; // return a class anyway for the demo
    }
    function __isset( $prop )
    {
        echo "\n", __FILE__, ':', __LINE__, ' ', __METHOD__, '(', $prop, ') instance ', $this->id ;
        return FALSE ;
    }
}

$x = new demo( 'demo' ) ;
echo "\n", 'Calls __isset() on demo as expected when executing isset( $x->a )' ;
$ret = isset( $x->a ) ;
echo "\n", 'Calls __get() on demo without call to __isset() when executing isset( $x->a->b )' ;
$ret = isset( $x->a->b ) ;
?>
```

Outputs

```
Calls __isset() on demo as expected when executing isset( $x->a )
C:\htdocs\test.php:31 demo::__isset(a) instance demo
Calls __get() on demo without call to __isset() when executing isset( $x->a->b )
C:\htdocs\test.php:26 demo::__get(a) instance demo
C:\htdocs\test.php:31 demo::__isset(b) instance autocreated
```

[up](#)

[down](#)

9

[PHP at jyopp dot Komm ¶](#)

**18 years ago**

Here's a useful class for logging function calls. It stores a sequence of calls and arguments which can then be applied to objects later. This can be used to script common sequences of operations, or to make "pluggable" operation sequences in header files that can be replayed on objects later.

If it is instantiated with an object to shadow, it behaves as a mediator and executes the calls on this object as they come in, passing back the values from the execution.

This is a very general implementation; it should be changed if error codes or exceptions need to be handled during the Replay process.

```
<?php
```

```
class MethodCallLog {
    private $calllog = array();
    private $object;
```

```

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

public function __call($m, $a) {
    $this->callLog[] = array($m, $a);
    if ($this->object) return call_user_func_array(array(&$this->object,$m),$a);
    return true;
}

public function Replay(&$object) {
    foreach ($this->callLog as $c) {
        call_user_func_array(array(&$object,$c[0]), $c[1]);
    }
}

public function GetEntries() {
    $rVal = array();
    foreach ($this->callLog as $c) {
        $rVal[] = "$c[0](".implode(' ', $c[1]).").";";
    }
    return $rVal;
}

public function Clear() {
    $this->callLog = array();
}
}

```

```

$log = new MethodCallLog();
$log->Method1();
$log->Method2("Value");
$log->Method1($a, $b, $c);
// Execute these method calls on a set of objects...
foreach ($array as $o) $log->Replay($o);
?>

```

[up](#)

[down](#)

4

[cottton at i-stats dot net ¶](#)

**9 years ago**

Actually you dont need \_\_set ect imo.

You could use it to set (pre-defined) protected (and in "some" cases private) properties . But who wants that?

(test it by uncommenting private or protected)

(pastebin because long ...) => <http://pastebin.com/By4gHrt5>

[up](#)

[down](#)

6

[jstubbs at work-at dot co dot jp ¶](#)

**17 years ago**

```
<?php $myclass->foo['bar'] = 'baz'; ?>
```

When overriding \_\_get and \_\_set, the above code can work (as expected) but it depends on your \_\_get implementation rather than your \_\_set. In fact, \_\_set is never called with the above code. It appears that PHP (at least as of 5.1) uses a reference to whatever was returned by \_\_get. To be more verbose, the above code is essentially identical to:

```

<?php
$tmp_array = &$myclass->foo;
$tmp_array['bar'] = 'baz';
unset($tmp_array);
?>

```

Therefore, the above won't do anything if your \_\_get implementation resembles this:

```

<?php
function __get($name) {

```

```

return array_key_exists($name, $this->values)
? $this->values[$name] : null;
}
?>

```

You will actually need to set the value in `__get` and return that, as in the following code:

```

<?php
function __get($name) {
if (!array_key_exists($name, $this->values))
$this->values[$name] = null;
return $this->values[$name];
}
?>

```

[up](#)

[down](#)

4

[justmyoponion at gmail dot com ¶](#)

**4 years ago**

If you are not focused enough, then don't use it.

Otherwise it is very powerful and you can build very complex code that handle a lot of things like zend framework did.

[up](#)

[down](#)

4

[matthijs at yourmediafactory dot com ¶](#)

**16 years ago**

While PHP does not support true overloading natively, I have to disagree with those that state this can't be achieved through `__call`.

Yes, it's not pretty but it is definately possible to overload a member based on the type of its argument. An example:

```

<?php
class A {

public function __call ($member, $arguments) {
if(is_object($arguments[0]))
$member = $member . 'Object';
if(is_array($arguments[0]))
$member = $member . 'Array';
$this -> $member($arguments);
}

private function testArray () {
echo "Array.";
}

private function testObject () {
echo "Object.";
}
}

class B {
}

$class = new A;
$class -> test(array()); // echo's 'Array.'
$class -> test(new B); // echo's 'Object.'
?>

```

Of course, the use of this is questionable (I have never needed it myself, but then again, I only have a very minimalistic C++ & JAVA background). However, using this general principle and optionally building forth on other suggestions a 'form' of overloading is definately possible, provided you have some strict naming conventions in your functions.

It would of course become a LOT easier once PHP'd let you declare the same member several times but with different arguments, since if you combine that with the reflection class 'real' overloading comes into the grasp of a good OO programmer. Lets keep our fingers crossed!

[up](#)

[down](#)

10

[alexandre at nospam dot gaigalas dot net ¶](#)

**16 years ago**

PHP 5.2.1

Its possible to call magic methods with invalid names using variable method/property names:

```
<?php

class foo
{
function __get($n)
{
print_r($n);
}
function __call($m, $a)
{
print_r($m);
}
}

$test = new foo;
$varname = 'invalid,variable+name';
$test->$varname;
$test->$varname();

?>
```

I just don't know if it is a bug or a feature :)

[up](#)

[down](#)

4

[timshaw at mail dot NOSPAMusa dot com ¶](#)

**16 years ago**

The \_\_get overload method will be called on a declared public member of an object if that member has been unset.

```
<?php

class c {
public $p ;
public function __get($name) { return "__get of $name" ; }
}

$c = new c ;
echo $c->p, "\n" ; // declared public member value is empty
$c->p = 5 ;
echo $c->p, "\n" ; // declared public member value is 5
unset($c->p) ;
echo $c->p, "\n" ; // after unset, value is "__get of p"

?>
```

[up](#)

[down](#)

3

[Marius ¶](#)

**18 years ago**

for anyone who's thinking about traversing some variable tree by using \_\_get() and \_\_set(). i tried to do this and found one problem: you can handle couple of \_\_get() in a row by returning

an object which can handle consequential `__get()`, but you can't handle `__get()` and `__set()` that way.

i.e. if you want to:

```
<?php
print($obj->val1->val2->val3); // three __get() calls
?> - this will work,
but if you want to:
<?php
$obj->val1->val2 = $val; // one __get() and one __set() call
?> - this will fail with message:
"Fatal error: Cannot access undefined property for object with
overloaded property access"
however if you don't mix __get() and __set() in one expression,
it will work:
```

```
<?php
$obj->val1 = $val; // only one __set() call
$val2 = $obj->val1->val2; // two __get() calls
$val2->val3 = $val; // one __set() call
?>
```

as you can see you can split `__get()` and `__set()` parts of expression into two expressions to make it work.

by the way, this seems like a bug to me, will have to report it.

[up](#)  
[down](#)

4

[daevid at daevid dot com ¶](#)

**14 years ago**

Here's a handy little routine to suggest properties you're trying to set that don't exist. For example:

Attempted to `__get()` non-existant property/variable 'operator\_id' in class 'User'.

checking for operator and suggesting the following:

```
* id_operator
* operator_name
* operator_code
```

enjoy.

```
<?php
/**
 * Suggests alternative properties should a __get() or __set() fail
 *
 * @param string $property
 * @return string
 * @author Daevid Vincent [daevid@daevid.com]
 * @date 05/12/09
 * @see __get(), __set(), __call()
 */
public function suggest_alternative($property)
{
    $parts = explode('_', $property);
    foreach($parts as $i => $p) if ($p == '_' || $p == 'id') unset($parts[$i]);

    echo 'checking for <b>'.implode(' ', $parts)."</b> and suggesting the following:<br/>\n";

    echo "<ul>";
    foreach($this as $key => $value)
    foreach($parts as $p)
    if (stripos($key, $p) !== false) print '<li>'. $key."</li>\n";
```



```
echo "</ul>";
}
```

just put it in your \_\_get() or \_\_set() like so:

```
public function __get($property)
{
    echo "<p><font color='#ff0000'>Attempted to __get() non-existant property/variable '". $property.'" in class '". $this->get_class_name()."'</font><p>\n";
    $this->suggest_alternative($property);
    exit;
}
?>
```

[up](#)

[down](#)

4

[Adeel Khan](#)

16 years ago

Observe:

```
<?php
class Foo {
function __call($m, $a) {
die($m);
}
}
```

```
$foo = new Foo;
print $foo->{'wow!'}();
```

```
// outputs 'wow!'
?>
```

This method allows you to call functions with invalid characters.

[up](#)

[down](#)

4

[Daniel Smith](#)

12 years ago

Be careful of \_\_call in case you have a protected/private method. Doing this:

```
<?php
class TestMagicCallMethod {
public function foo()
{
    echo __METHOD__.PHP_EOL;
}

public function __call($method, $args)
{
    echo __METHOD__.PHP_EOL;
    if(method_exists($this, $method))
    {
        $this->$method();
    }
}
```

```
protected function bar()
{
    echo __METHOD__.PHP_EOL;
}
```

```

private function baz()
{
echo __METHOD__.PHP_EOL;
}
}

$test = new TestMagicCallMethod();
$test->foo();
/**
 * Outputs:
 * TestMagicCallMethod::foo
 */

$test->bar();
/**
 * Outputs:
 * TestMagicCallMethod::__call
 * TestMagicCallMethod::bar
 */

$test->baz();
/**
 * Outputs:
 * TestMagicCallMethod::__call
 * TestMagicCallMethod::baz
 */
?>

```

..is probably not what you should be doing. Always make sure that the methods you call in \_\_call are allowed as you probably dont want all the private/protected methods to be accessed by a typo or something.

[up](#)

[down](#)

2

[Nanhe Kumar ¶](#)

**10 years ago**

<?php

//How can implement \_\_call function you understand better

```
class Employee {
```

```
protected $_name;
```

```
protected $_email;
```

```
protected $_compony;
```

```
public function __call($name, $arguments) {
```

```
$action = substr($name, 0, 3);
```

```
switch ($action) {
```

```
case 'get':
```

```
$property = '_' . strtolower(substr($name, 3));
```

```
if(property_exists($this,$property)){
```

```
return $this->{$property};
```

```
}else{
```

```
$trace = debug_backtrace();
```

```
trigger_error('Undefined property ' . $name . ' in ' . $trace[0]['file'] . ' on line ' . $trace[0]['line'],
```

```
E_USER_NOTICE);
```

```
return null;
```

```
}
```

```
break;
```

```
case 'set':
```

```
$property = '_' . strtolower(substr($name, 3));
```

```
if(property_exists($this,$property)){
```

```
$this->{$property} = $arguments[0];
```

```
}else{
```

```

$trace = debug_backtrace();
trigger_error('Undefined property ' . $name . ' in ' . $trace[0]['file'] . ' on line ' . $trace[0]['line'],
E_USER_NOTICE);
return null;
}

break;
default :
return FALSE;
}
}

}

```

```

$s = new Employee();
$s->setName('Nanhe Kumar');
$s->setEmail('nanhe.kumar@gmail.com');
echo $s->getName(); //Nanhe Kumar
echo $s->getEmail(); // nanhe.kumar@gmail.com
$s->setAge(10); //Notice: Undefined property setAge in
?>

```

[up](#)

[down](#)

2

[strata ranger at hotmail dot com ¶](#)

**14 years ago**

Combining two things noted previously:

- 1 - Unsetting an object member removes it from the object completely, subsequent uses of that member will be handled by magic methods.
- 2 - PHP will not recursively call one magic method from within itself (at least for the same \$name).

This means that if an object member has been unset(), it IS possible to re-declare that object member (as public) by creating it within your object's \_\_set() method, like this:

```

<?php
class Foo
{
function __set($name, $value)
{
// Add a new (public) member to this object.
// This works because __set() will not recursively call itself.
$this->$name= $value;
}
}

$foo = new Foo();

// $foo has zero members at this point
var_dump($foo);

// __set() will be called here
$foo->bar = 'something'; // Calls __set()

// $foo now contains one member
var_dump($foo);

// Won't call __set() because 'bar' is now declared
$foo->bar = 'other thing';

?>

```

Also be mindful that if you want to break a reference involving an object member without triggering magic functionality, DO NOT unset() the object member directly. Instead use =& to bind the object member to any convenient null variable.

[up](#)

[down](#)

2

[DevilDude at darkmaker dot com ¶](#)

**19 years ago**

Php 5 has a simple recursion system that stops you from using overloading within an overloading function, this means you cannot get an overloaded variable within the \_\_get method, or within any functions/methods called by the \_get method, you can however call \_\_get manually within itself to do the same thing.

[up](#)

[down](#)

1

[dans at dansheps dot com ¶](#)

**12 years ago**

Since this was getting me for a little bit, I figure I better pipe in here...

For nested calls to private/protected variables(probably functions too) what it does is call a \_\_get() on the first object, and if you return the nested object, it then calls a \_\_get() on the nested object because, well it is protected as well.

EG:

```
<?php
class A
{
protected $B

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

public function __get($variable)
{
echo "Class A::Variable " . $variable . "\n\r";
$retval = $this->{$variable};
return $retval;
}
}

class B
{
protected $val

public function __construct()
{
$this->val = 1;
}

public function __get($variable)
{
echo "Class B::Variable " . $variable . "\n\r";
$retval = $this->{$variable};
return $retval;
}
}

$A = new A();

echo "Final Value: " . $A->B->val;
?>
```

That will return something like...

```
Class A::Variable B
Class B::Variable val
Final Value: 1
```

It separates the calls into \$A->B and \$B->val

Hope this helps someone

[up](#)

[down](#)

0

[\*php at sleep is the enemy dot co dot uk ¶\*](#)

**16 years ago**

Just to reinforce and elaborate on what DevilDude at darkmaker dot com said way down there on 22-Sep-2004 07:57.

The recursion detection feature can prove especially perilous when using \_\_set. When PHP comes across a statement that would usually call \_\_set but would lead to recursion, rather than firing off a warning or simply not executing the statement it will act as though there is no \_\_set method defined at all. The default behaviour in this instance is to dynamically add the specified property to the object thus breaking the desired functionality of all further calls to \_\_set or \_\_get for that property.

Example:

```
<?php
```

```
class TestClass{
```

```
public $values = array();
```

```
public function __get($name){
return $this->values[$name];
}
```

```
public function __set($name, $value){
$this->values[$name] = $value;
$this->validate($name);
}
```

```
public function validate($name){
/*
```

```
__get will be called on the following line
```

```
but as soon as we attempt to call __set
```

```
again PHP will refuse and simply add a
```

```
property called $name to $this
```

```
*/
```

```
$this->$name = trim($this->$name);
```

```
}
```

```
}
```

```
$tc = new TestClass();
```

```
$tc->foo = 'bar';
```

```
$tc->values['foo'] = 'boing';
```

```
echo '$tc->foo == ' . $tc->foo . '<br>';
```

```
echo '$tc ' . (property_exists($tc, 'foo')) ? 'now has' : 'still does not have') . ' a property called "foo"<br>';
```

```
/*
```

```
OUTPUTS:
```

```
$tc->foo == bar
```

```
$tc now has a property called "foo"
```

[+ add a note](#)

- [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](#)
- [Copyright © 2001-2024 The PHP Group](#)
- [My PHP.net](#)
- [Contact](#)
- [Other PHP.net sites](#)
- [Privacy policy](#)

