Variable variables »
« Predefined Variables

- PHP Manual
- Language Reference
- Variables

Change language: English

Submit a Pull Request Report a Bug

Variable scope ¶

The scope of a variable is the context within which it is defined. For the most part all PHP variables only have a single scope. This single scope spans included and required files as well. For example:

```
<?php
$a = 1;
include 'b.inc';
?>
```

Here the *$a* variable will be available within the included *b.inc* script. However, within user-defined functions a local function scope is introduced. Any variable used inside a function is by default limited to the local function scope. For example:

```
<?php
$a = 1; /* global scope */

function test()
{
    echo $a; /* reference to local scope variable */
}

test();
?>
```

This script will not produce any output because the echo statement refers to a local version of the *$a* variable, and it has not been assigned a value within this scope. You may notice that this is a little bit different from the C language in that global variables in C are automatically available to functions unless specifically overridden by a local definition. This can cause some problems in that people may inadvertently change a global variable. In PHP global variables must be declared global inside a function if they are going to be used in that function.

The `global` keyword ¶

First, an example use of `global`:

Example #1 Using `global`

```
<?php
$a = 1;
$b = 2;

function Sum()
{
    global $a, $b;
```

```
    $b = $a + $b;
}

Sum();
echo $b;
?>
```

The above script will output `3`. By declaring *$a* and *$b* global within the function, all references to either variable will refer to the global version. There is no limit to the number of global variables that can be manipulated by a function.

A second way to access variables from the global scope is to use the special PHP-defined *$GLOBALS* array. The previous example can be rewritten as:

Example #2 Using *$GLOBALS* instead of global

```
<?php
$a = 1;
$b = 2;

function Sum()
{
    $GLOBALS['b'] = $GLOBALS['a'] + $GLOBALS['b'];
}

Sum();
echo $b;
?>
```

The *$GLOBALS* array is an associative array with the name of the global variable being the key and the contents of that variable being the value of the array element. Notice how *$GLOBALS* exists in any scope, this is because *$GLOBALS* is a superglobal. Here's an example demonstrating the power of superglobals:

Example #3 Example demonstrating superglobals and scope

```
<?php
function test_superglobal()
{
    echo $_POST['name'];
}
?>
```

> **Note**:
>
> Using `global` keyword outside a function is not an error. It can be used if the file is included from inside a function.

Using `static` variables ¶

Another important feature of variable scoping is the *static* variable. A static variable exists only in a local function scope, but it does not lose its value when program execution leaves this scope. Consider the following example:

Example #4 Example demonstrating need for static variables

```
<?php
function test()
{
    $a = 0;
```

```
    echo $a;
    $a++;
}
?>
```

This function is quite useless since every time it is called it sets *$a* to `0` and prints `0`. The *$a*++ which increments the variable serves no purpose since as soon as the function exits the *$a* variable disappears. To make a useful counting function which will not lose track of the current count, the *$a* variable is declared static:

Example #5 Example use of static variables

```
<?php
function test()
{
    static $a = 0;
    echo $a;
    $a++;
}
?>
```

Now, *$a* is initialized only in first call of function and every time the `test()` function is called it will print the value of *$a* and increment it.

Static variables also provide one way to deal with recursive functions. A recursive function is one which calls itself. Care must be taken when writing a recursive function because it is possible to make it recurse indefinitely. You must make sure you have an adequate way of terminating the recursion. The following simple function recursively counts to 10, using the static variable *$count* to know when to stop:

Example #6 Static variables with recursive functions

```
<?php
function test()
{
    static $count = 0;

    $count++;
    echo $count;
    if ($count < 10) {
        test();
    }
    $count--;
}
?>
```

Static variables can be assigned values which are the result of constant expressions, but dynamic expressions, such as function calls, will cause a parse error.

Example #7 Declaring static variables

```
<?php
function foo(){
    static $int = 0;          // correct
    static $int = 1+2;        // correct
    static $int = sqrt(121);  // wrong  (as it is a function)

    $int++;
    echo $int;
```

```
}
?>
```

As of PHP 8.1.0, when a method using static variables is inherited (but not overridden), the inherited method will now share static variables with the parent method. This means that static variables in methods now behave the same way as static properties.

Example #8 Usage of static Variables in Inherited Methods

```
<?php
class Foo {
    public static function counter() {
        static $counter = 0;
        $counter++;
        return $counter;
    }
}
class Bar extends Foo {}
var_dump(Foo::counter()); // int(1)
var_dump(Foo::counter()); // int(2)
var_dump(Bar::counter()); // int(3), prior to PHP 8.1.0 int(1)
var_dump(Bar::counter()); // int(4), prior to PHP 8.1.0 int(2)
?>
```

> **Note**:
>
> Static declarations are resolved in compile-time.

References with `global` and `static` variables ¶

PHP implements the static and global modifier for variables in terms of references. For example, a true global variable imported inside a function scope with the `global` statement actually creates a reference to the global variable. This can lead to unexpected behaviour which the following example addresses:

```
<?php
function test_global_ref() {
    global $obj;
    $new = new stdclass;
    $obj = &$new;
}

function test_global_noref() {
    global $obj;
    $new = new stdclass;
    $obj = $new;
}

test_global_ref();
var_dump($obj);
test_global_noref();
var_dump($obj);
?>
```

The above example will output:

```
NULL
object(stdClass)#1 (0) {
}
```

A similar behaviour applies to the `static` statement. References are not stored statically:

```php
<?php
function &get_instance_ref() {
    static $obj;

    echo 'Static object: ';
    var_dump($obj);
    if (!isset($obj)) {
        $new = new stdclass;
        // Assign a reference to the static variable
        $obj = &$new;
    }
    if (!isset($obj->property)) {
        $obj->property = 1;
    } else {
        $obj->property++;
    }
    return $obj;
}

function &get_instance_noref() {
    static $obj;

    echo 'Static object: ';
    var_dump($obj);
    if (!isset($obj)) {
        $new = new stdclass;
        // Assign the object to the static variable
        $obj = $new;
    }
    if (!isset($obj->property)) {
        $obj->property = 1;
    } else {
        $obj->property++;
    }
    return $obj;
}

$obj1 = get_instance_ref();
$still_obj1 = get_instance_ref();
echo "\n";
$obj2 = get_instance_noref();
$still_obj2 = get_instance_noref();
?>
```

The above example will output:

```
Static object: NULL
Static object: NULL

Static object: NULL
Static object: object(stdClass)#3 (1) {
  ["property"]=>
  int(1)
}
```

This example demonstrates that when assigning a reference to a static variable, it's not *remembered* when you call the `&get_instance_ref()` function a second time.

+ add a note

User Contributed Notes 13 notes

up
down
165
dodothedreamer at gmail dot com ¶
11 years ago

Note that unlike Java and C++, variables declared inside blocks such as loops or if's, will also be recognized and accessible outside of the block, so:

```
<?php
for($j=0; $j<3; $j++)
{
     if($j == 1)
        $a = 4;
}
echo $a;
?>
```

Would print 4.

up
down
165
warhog at warhog dot net ¶
16 years ago

Some interesting behavior (tested with PHP5), using the static-scope-keyword inside of class-methods.

```
<?php

class sample_class
{
  public function func_having_static_var($x = NULL)
  {
    static $var = 0;
    if ($x === NULL)
    { return $var; }
    $var = $x;
  }
}

$a = new sample_class();
$b = new sample_class();

echo $a->func_having_static_var()."\n";
echo $b->func_having_static_var()."\n";
// this will output (as expected):
//  0
//  0

$a->func_having_static_var(3);

echo $a->func_having_static_var()."\n";
echo $b->func_having_static_var()."\n";
// this will output:
//  3
```

```
//  3
// maybe you expected:
//  3
//  0

?>
```

One could expect "3 0" to be outputted, as you might think that $a->func_having_static_var(3); only alters the value of the static $var of the function "in" $a - but as the name says, these are class-methods. Having an object is just a collection of properties, the functions remain at the class. So if you declare a variable as static inside a function, it's static for the whole class and all of its instances, not for each object.

Maybe it's senseless to post that.. cause if you want to have the behaviour that I expected, you can simply use a variable of the object itself:

```
<?php
class sample_class
{ protected $var = 0;
  function func($x = NULL)
  { $this->var = $x; }
} ?>
```

I believe that all normal-thinking people would never even try to make this work with the static-keyword, for those who try (like me), this note maybe helpfull.

up
down
25

Michael Bailey (jinxidoru at byu dot net) ¶
18 years ago

Static variables do not hold through inheritance. Let class A have a function Z with a static variable. Let class B extend class A in which function Z is not overwritten. Two static variables will be created, one for class A and one for class B.

Look at this example:

```
<?php
class A {
    function Z() {
        static $count = 0;
        printf("%s: %d\n", get_class($this), ++$count);
    }
}

class B extends A {}

$a = new A();
$b = new B();
$a->Z();
$a->Z();
$b->Z();
$a->Z();
?>
```

This code returns:

```
A: 1
A: 2
```

```
B: 1
A: 3
```

As you can see, class A and B are using different static variables even though the same function was being used.

up
down
25

andrew at planetubh dot com ¶

13 years ago

Took me longer than I expected to figure this out, and thought others might find it useful.

I created a function (safeinclude), which I use to include files; it does processing before the file is actually included (determine full path, check it exists, etc).

Problem: Because the include was occurring inside the function, all of the variables inside the included file were inheriting the variable scope of the function; since the included files may or may not require global variables that are declared else where, it creates a problem.

Most places (including here) seem to address this issue by something such as:

```
<?php
//declare this before include
global $myVar;
//or declare this inside the include file
$nowglobal = $GLOBALS['myVar'];
?>
```

But, to make this work in this situation (where a standard PHP file is included within a function, being called from another PHP script; where it is important to have access to whatever global variables there may be)... it is not practical to employ the above method for EVERY variable in every PHP file being included by 'safeinclude', nor is it practical to staticly name every possible variable in the "global $this" approach. (namely because the code is modulized, and 'safeinclude' is meant to be generic)

My solution: Thus, to make all my global variables available to the files included with my safeinclude function, I had to add the following code to my safeinclude function (before variables are used or file is included)

```
<?php
foreach ($GLOBALS as $key => $val) { global $$key; }
?>
```

Thus, complete code looks something like the following (very basic model):

```
<?php
function safeinclude($filename)
{
    //This line takes all the global variables, and sets their scope within the function:
    foreach ($GLOBALS as $key => $val) { global $$key; }
    /* Pre-Processing here: validate filename input, determine full path
        of file, check that file exists, etc. This is obviously not
        necessary, but steps I found useful. */
    if ($exists==true) { include("$file"); }
    return $exists;
}
?>
```

In the above, 'exists' & 'file' are determined in the pre-processing. File is the full server path

to the file, and exists is set to true if the file exists. This basic model can be expanded of course. In my own, I added additional optional parameters so that I can call safeinclude to see if a file exists without actually including it (to take advantage of my path/etc preprocessing, verses just calling the file exists function).

Pretty simple approach that I could not find anywhere online; only other approach I could find was using PHP's eval().

up
down
15

larax at o2 dot pl ¶

16 years ago

About more complex situation using global variables..

Let's say we have two files:

a.php

```
<?php
    function a() {
        include("b.php");
    }
    a();
?>
```

b.php

```
<?php
    $b = "something";
    function b() {
        global $b;
        $b = "something new";
    }
    b();
    echo $b;
?>
```

You could expect that this script will return "something new" but no, it will return "something". To make it working properly, you must add global keyword in $b definition, in above example it will be:

```
global $b;
$b = "something";
```

up
down
3

gried at NOSPAM dot nsys dot by ¶

6 years ago

In fact all variables represent pointers that hold address of memory area with data that was assigned to this variable. When you assign some variable value by reference you in fact write address of source variable to recepient variable. Same happens when you declare some variable as global in function, it receives same address as global variable outside of function. If you consider forementioned explanation it's obvious that mixing usage of same variable declared with keyword global and via superglobal array at the same time is very bad idea. In some cases they can point to different memory areas, giving you headache. Consider code below:

```
<?php

error_reporting(E_ALL);

$GLOB = 0;
```

```
function test_references() {
    global $GLOB; // get reference to global variable using keyword global, at this point local variable $GLOB points to same address as global variable $GLOB
    $test = 1; // declare some local var
    $GLOBALS['GLOB'] = &$test; // make global variable reference to this local variable using superglobal array, at this point global variable $GLOB points to new memory address, same as local variable $test

    $GLOB = 2; // set new value to global variable via earlier set local representation, write to old address

    echo "Value of global variable (via local representation set by keyword global): $GLOB <hr>";
    // check global variable via local representation => 2 (OK, got value that was just written to it, cause old address was used to get value)

    echo "Value of global variable (via superglobal array GLOBALS): $GLOBALS[GLOB] <hr>";
    // check global variable using superglobal array => 1 (got value of local variable $test, new address was used)

    echo "Value ol local variable \$test: $test <hr>";
    // check local variable that was linked with global using superglobal array => 1 (its value was not affected)

    global $GLOB; // update reference to global variable using keyword global, at this point we update address that held in local variable $GLOB and it gets same address as local variable $test
    echo "Value of global variable (via updated local representation set by keyword global): $GLOB <hr>";
    // check global variable via local representation => 1 (also value of local variable $test, new address was used)
}

test_references();
echo "Value of global variable outside of function: $GLOB <hr>";
// check global variable outside function => 1 (equal to value of local variable $test from function, global variable also points to new address)
?>
```

up
down
5
dexen dot devries at gmail dot com ¶
5 years ago

If you have a static variable in a method of a class, all DIRECT instances of that class share that one static variable.

However if you create a derived class, all DIRECT instances of that derived class will share one, but DISTINCT, copy of that static variable in method.

To put it the other way around, a static variable in a method is bound to a class (not to instance). Each subclass has own copy of that variable, to be shared among its instances.

To put it yet another way around, when you create a derived class, it 'seems to' create a copy of methods from the base class, and thusly create copy of the static variables in those methods.

Tested with PHP 7.0.16.

```
<?php
```

```
require 'libs.php';
require 'setup.php';

class Base {
    function test($delta = 0) {
        static $v = 0;
        $v += $delta;
        return $v;
    }
}

class Derived extends Base {}

$base1 = new Base();
$base2 = new Base();
$derived1 = new Derived();
$derived2 = new Derived();

$base1->test(3);
$base2->test(4);
$derived1->test(5);
$derived2->test(6);

var_dump([ $base1->test(), $base2->test(), $derived1->test(), $derived2->test() ]);

# => array(4) { [0]=> int(7) [1]=> int(7) [2]=> int(11) [3]=> int(11) }

# $base1 and $base2 share one copy of static variable $v
# derived1 and $derived2 share another copy of static variable $v
```

up
down
1

jakub dot lopuszanski at nasza-klasa dot pl ¶

12 years ago

If you use __autoload function to load classes' definitons, beware that "static local variables are resolved at compile time" (whatever it really means) and the order in which autoloads occur may impact the semantic.

For example if you have:

```
<?php
class Singleton{
  static public function get_instance(){
     static $instance = null;
     if($instance === null){
        $instance = new static();
     }
     return $instance;
  }
}
?>
```

and two separate files A.php and B.php:

```
class A extends Singleton{}
class B extends A{}
```

then depending on the order in which you access those two classes, and consequently, the order in which __autoload includes them, you can get strange results of calling B::get_instance() and A::get_instance().

It seems that static local variables are alocated in as many copies as there are classes that inherit a method at the time of inclusion of parsing Singleton.

up
down
1
moraesdno at gmail dot com ¶
12 years ago

Use the superglobal array $GLOBALS is faster than the global keyword. See:

```
<?php
//Using the keyword global
$a=1;
$b=2;
function sum() {
    global $a, $b;
    $a += $b;
}

$t = microtime(true);
for($i=0; $i<1000; $i++) {
     sum();
}
echo microtime(true)-$t;
echo " -- ".$a."<br>";

//Using the superglobal array
$a=1;
$b=2;
function sum2() {
    $GLOBALS['a'] += $GLOBALS['b'];
}

  $t = microtime(true);
for($i=0; $i<1000; $i++) {
     sum2();
}
echo microtime(true)-$t;
echo " -- ".$a."<br>";
?>
```

up
down
0
simon dot barotte at gmail dot com ¶
5 years ago

To be vigilant, unlike Java or C++, variables declared inside blocks such as loops (for, while,...) or if's, will also be recognized and accessible outside of the block, the only valid block is the BLOCK function so:

```
<?php
for($j=0; $j<5; $j++)
{
     if($j == 1){
        $a = 6;
     }
}

echo $a;
```

```
?>

Would print 6.
```

up
down
0

pogregoire##live.fr ¶

6 years ago

```
writing : global $var; is exactely the samething that writing : $var =& $GLOBALS['var'];
It creates a reference on $GLOBALS['var'];

<?php
$var =1;
function teste_global(){
    global $var;
    for ($var=0; $var<5; $var++){

    }
}

teste_global();
var_dump($var);// return : int(5).
?>
```

up
down
1

jameslee at cs dot nmt dot edu ¶

17 years ago

```
It should be noted that a static variable inside a method is static across all instances of that
class, i.e., all objects of that class share the same static variable.  For example the code:

<?php
class test {
    function z() {
        static $n = 0;
        $n++;
        return $n;
    }
}

$a =& new test();
$b =& new test();
print $a->z();  // prints 1, as it should
print $b->z();  // prints 2 because $a and $b have the same $n
?>

somewhat unexpectedly prints:
1
2
```

up
down
-1

jake dot tunaley at berkeleyit dot com ¶

3 years ago

```
Beware of using $this in anonymous functions assigned to a static variable.

<?php
class Foo {
```

```
    public function bar() {
        static $anonymous = null;
        if ($anonymous === null) {
            // Expression is not allowed as static initializer workaround
            $anonymous = function () {
                return $this;
            };
        }
        return $anonymous();
    }
}

$a = new Foo();
$b = new Foo();
var_dump($a->bar() === $a); // True
var_dump($b->bar() === $a); // Also true
?>
```

In a static anonymous function, $this will be the value of whatever object instance that method was called on first.

To get the behaviour you're probably expecting, you need to pass the $this context into the function.

```
<?php
class Foo {
    public function bar() {
        static $anonymous = null;
        if ($anonymous === null) {
            // Expression is not allowed as static initializer workaround
            $anonymous = function (self $thisObj) {
                return $thisObj;
            };
        }
        return $anonymous($this);
    }
}

$a = new Foo();
$b = new Foo();
var_dump($a->bar() === $a); // True
var_dump($b->bar() === $a); // False
?>
```