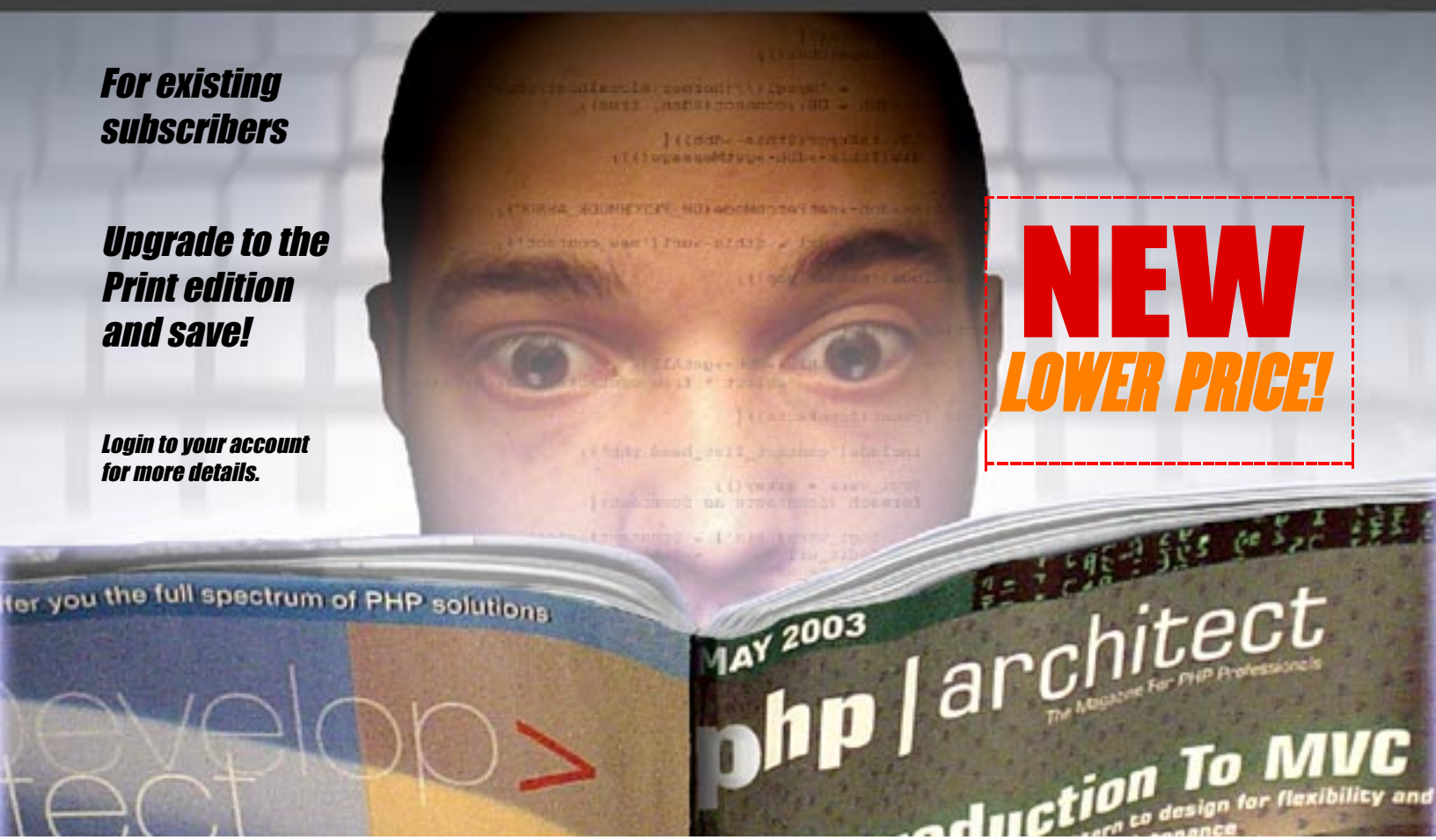
You'll never know what we'll come up with next



php | architect
The Magazine For PHP Professionals

Visit: ***http://www.phparch.com/print*** for more information or to subscribe online.

php|architect Subscription Dept.
P.O. Box 54526
1771 Avenue Road
Toronto, ON M5M 4N5
Canada

Your charge will appear under the name "Marco Tabini & Associates, Inc." Please allow up to 4 to 6 weeks for your subscription to be established and your first issue to be mailed to you.

*US Pricing is approximate and for illustration purposes only.

NEW LOWER PRICE!

Name: _____

Address: _____

City: _____

State/Province: _____

ZIP/Postal Code: _____

Country: _____

Payment type:

VISA	Mastercard	American Express
☐	☐	☐

Credit Card Number: _____

Expiration Date: _____

E-mail address: _____

Phone Number: _____

Choose a Subscription type:

☐	Canada/USA	$ 77.99 CAD	($59.99 US*)
☐	International Air	$105.19 CAD	($80.89 US*)
☐	Combo edition add-on (print + PDF edition)	$ 14.00 CAD	($10.00 US)

Signature: **Date:**

*By signing this order form, you agree that we will charge your account in Canadian dollars for the "CAD" amounts indicated above. Because of fluctuations in the exchange rates, the actual amount charged in your currency on your credit card statement may vary slightly.

To subscribe via snail mail - please detach/copy this form, fill it out and mail to the address above or fax to +1-416-630-5057

References in PHP:

An In-Depth Look

by Derick Rethans

PHP's handling of variables can be non-obvious, at times. Have you ever wondered what happens at the engine level when a variable is copied to another? How about when a function returns a variable "by reference?" If so, read on.

Every computer language needs some form of container to hold data—variables. In some languages, those variables have a specific type attached to them. They can be a string, a number, an array, an object or something else. Examples of such statically-typed languages are C and pascal. Variables in PHP do not have this specific restraint. They can be a string in one line, but a number in the next line. Converting between types is also easy to do, and often, even automatic. These loosely-typed variables are one of the properties that make PHP such an easy and powerful language, although they can sometimes also cause interesting problems.

Internally, in PHP, those variables are all stored in a similar container, called a zval container (also called "variable container"). This container keeps track of several things that are related to a specific value. The most important things that a variable container contains are the value of the "variable", but also the type of the variable. Python is similar to PHP in this regard as it also labels each variable with a type. The variable container contains a few more fields that the PHP engine uses to keep track of whether a value is a reference or not. It also keeps reference count of its value.

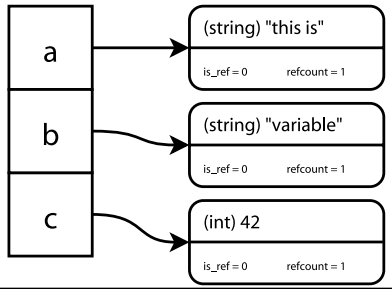
Variables are stored in a symbol table, which is quite analogous to an associative array. This array has keys that represent the name of the variable, and those keys point to variable containers that contain the value (and type) of the variables. See Figure 1 for an example of this.

Reference Counting

PHP tries to be smart when it deals with copying variables like in `$a = $b`. Using the `=` operator is also called an "assign-by-value" operation. While assigning by value, the PHP engine will not actually create a copy of the variable container, but it will merely increase the `refcount` field in the variable container. As you can imagine this saves a lot of memory in case you have a large string of text, or a large array. Figure 2 shows how this "looks". In Step 1 there is one variable, `a`, which

REQUIREMENTS	
PHP	**4.3.0+**
OS	**Any**
Other Software	**N/A**
Code Directory	**references**

Figure 1

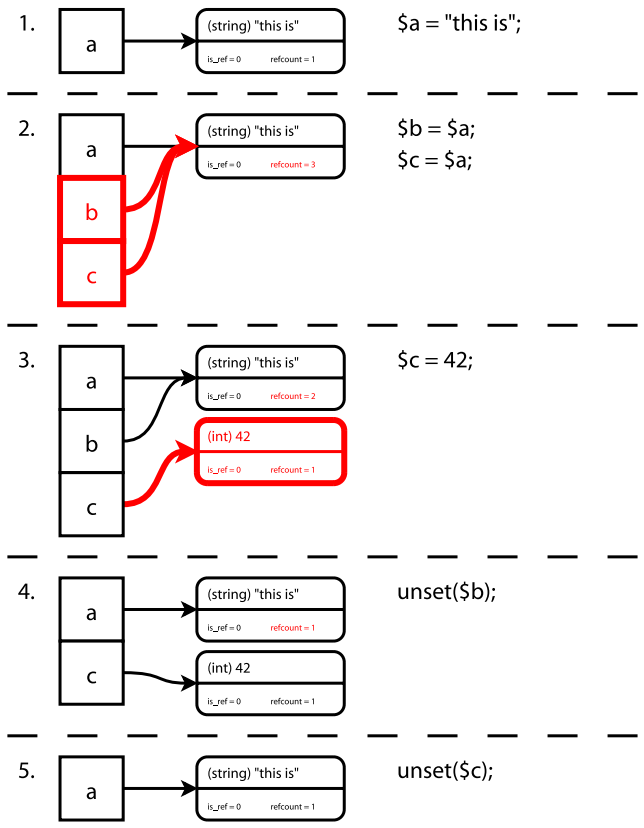


contains the text `this is` and it has (by default) a reference count of 1. In step 2, we assign variable `$a` to variable `$b` and `$c`. Here, no copy of the variable container is made, only the *refcount* value gets updated with 1 for each variable that is assigned to the container. Because we assign two more variables here, the `refcount` gets updated to 2 and ends up being 3 after the two assignment statements.

Now, you might wonder what would happen if the variable `$c` gets changed. Two things might happen, depending on the value of the `refcount`. If the value is 1, then the container simply gets updated with its new value (and possibly its type, too). In case the `refcount` value is larger than 1, a new variable container gets created containing the new value (and type). You can see this in step 3 of Figure 2. The `refcount` value for the variable container that is linked to the variable `$a` is decreased by one so that the variable container that belongs to variable `$a` and `$b` now has a `refcount` of 2, and the newly created container has a `refcount` of 1.

When `unset()` is called on a variable the `refcount` value of the variable container that is linked to the variable that is unset will be decreased by one. This happens when we call `unset($b)` in step 4. If the `refcount` value drops below 1, the PHP Engine will free the variable container. The variable container is then destroyed, as you can see in step 5.

Figure 2



Passing Variables to Functions

Besides the global symbol table that every script has, every call to a user defined function creates a symbol table where a function locally stores its variables. Every time a function is called, such a symbol table is created, and every time a function returns, this symbol table is destroyed. A function returns by either using the `return` statement, or by implicitly returning because the end of the function has been reached.

In Figure 3, I illustrate exactly how variables are passed to functions. In step 1, we assign a value to the variable `$a`, again—"this is". We pass this variable to the `do_something()` function, where it is received in the variable `$s`. In step 2, you can see that it is practically the same operation as assigning a variable to another one (like we did in the previous section with `$b = $a`), except that the variable is stored in a different symbol table—the one that belongs to the called function—and that the reference count is increased twice, instead the normal once. The reason for this is that the function's stack also contains a reference to the variable container.

When we assign a new value to the variable `$s` in step 3, the `refcount` of the original variable container is decreased by one and a new variable container is created, containing the new variable. In step 4, we return the variable with the `return` statement. The returned variable gets an entry in the global symbol table and the `refcount` value is increased by 1. When the function ends, the function's symbol table will be destroyed. During the destruction, the engine will go over all variables in the symbol table and decrease the `refcount` of each variable container. When a `refcount` of a variable container reaches 0, the variable container is destroyed. As you see, the variable container is again not copied when returning it from the function due to PHP's reference counting mechanism.

If the variable `$s` would not have been modified in step 3 then variable `$a` and `$b` would still point to the same variable container which would have a `refcount` value of 2. In this situation, a copy of the variable container that was created with the statement `$a = "this is"` would not have been made.

Introducing References

References are a method of having two names for the same variable. A more technical description would be: references are a method of having two keys in a symbol table pointing to the same zval container. References can be created with the *reference assignment* operator `&=`.

Figure 3

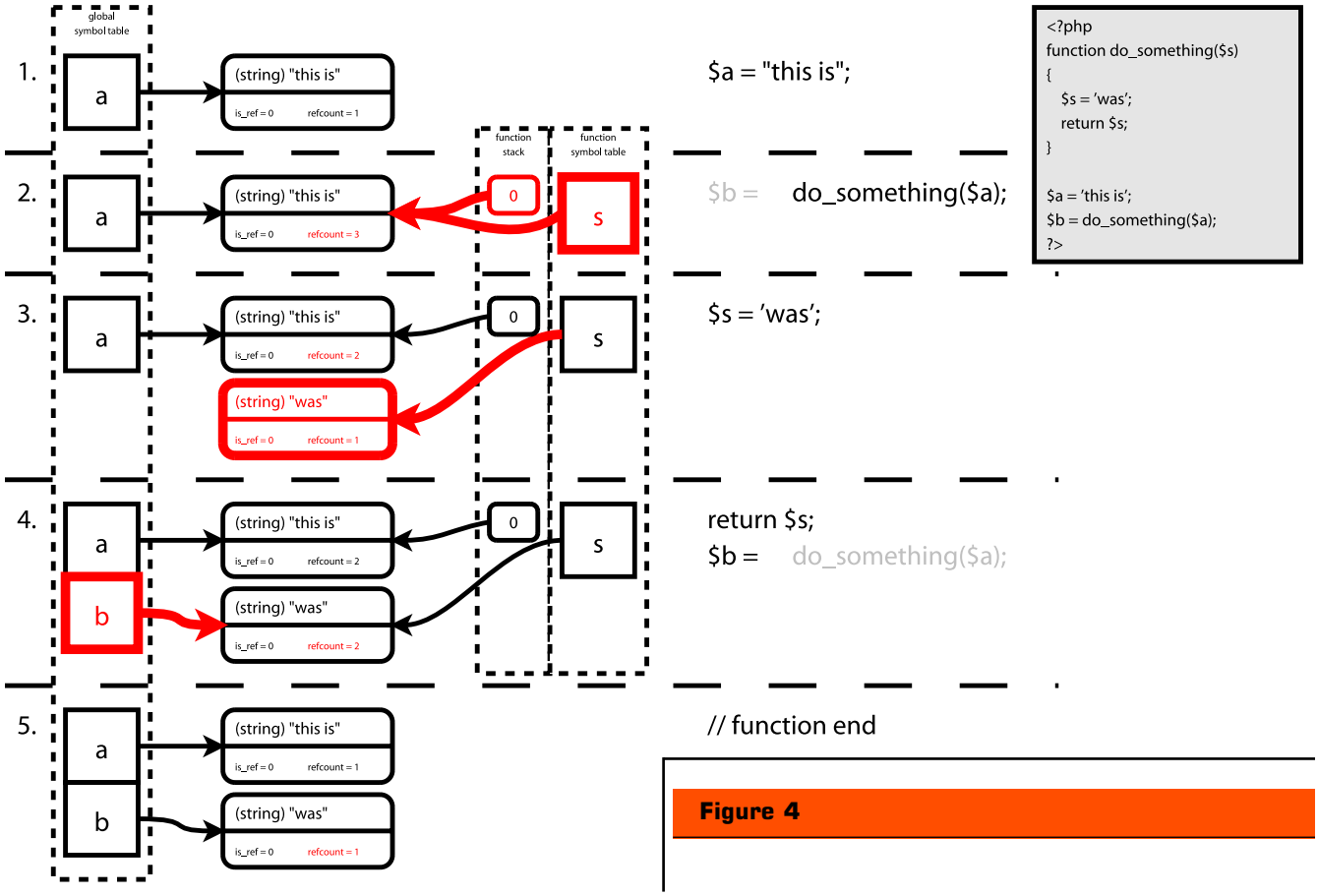
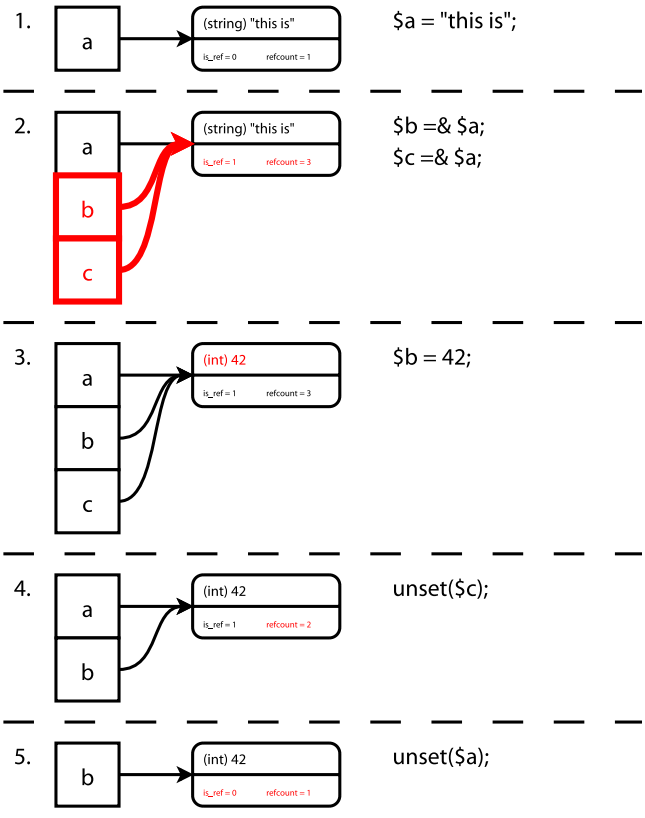


Figure 4 gives a schematic overview of how references work in combination with reference counting. In step 1, we create a variable **$a** that contains the string "this is". Then in step two we create two references (**$b** and **$c**) to the same variable container. The **refcount** increases normally for each assignment making the final **refcount** 3, after both assignments by reference (**$b =& $a** and **$c =& $a**), but because the *reference assignment* operator is used, the other value **is_ref** is now set to 1. This value is important for two reasons. The second one I will divulge a little bit later in this article, and the first reason that makes this value important is when we are reassigning a new value to one of the three variables that all point to the same variable container.

If the **is_ref** value is set to 0 when a new value is set for a specific variable, the PHP engine will create a new variable container as you could see in step 3 of Figure 2. But if the **is_ref** value is set to 1, then the PHP engine will not create a new variable container and simply only update the value to which one of the variable names point as you can see in step 2 of Figure 4. The exact same result would be reached when the statement **$a = 42** was used instead of **$b = 42**. After the variable container is modified, all three variables **$a**, **$b**

Figure 4



and $c will contain the value 42.

In step 4, we use the unset() language construct to remove a variable—in this case variable $c. Using unset() on a variable means that the refcount value of the variable container that the variable points to gets decreased by 1. This works exactly the same for referenced variables. There is one difference, though, that shows in step 5. When the reference count of a variable container reaches 1 and the is_ref value is set to 1, the is_ref value is reset to 0. The reason for this is that a variable container can only be marked as a referenced variable container when there is more than one variable pointing to the variable container.

Mixing Assign-by-Value and Assign-by-Reference

Something interesting—and perhaps unexpected—happens if you mix an assign-by-value call and an assign-by-reference call. This shows in Figure 5. In the first step we create two variables $a and $b, where the latter is assigned-by-value to the former. This creates a situation where there is one variable container with is_ref set to 0 and refcount set to 2. This should be familiar by now.

In step 2 we proceed by assigning variable $c by reference to variable $b. Here, the PHP engine will create a copy of the variable container. The variable $a keeps pointing to the original variable container but the refcount is, of course, decreased to 1 as there is only one variable pointing the this variable container now. The variables $b and $c point to the copied container which has now a refcount of 2 and the is_ref value is set to 1.

Figure 5

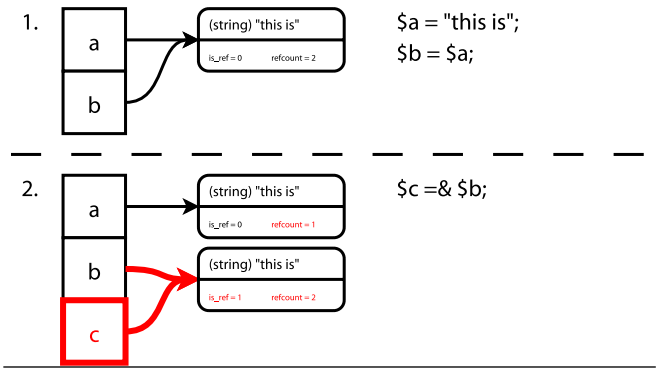
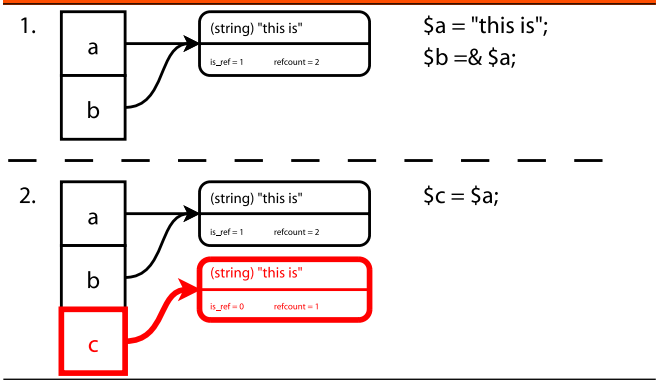


Figure 6



You can see that in this case, using a reference does *not* save you any memory, it actually uses more memory, as it had to duplicate the original variable container. The container had to be copied, otherwise the PHP engine would have no way of knowing how to deal with the reassignment of one of the three variables as two of them were references to the same container $b and $c, while the other was not supposed to be a reference. If there is only one container with refcount set to 3, and is_ref set to 1, then it is impossible to figure that out. That is the reason why the PHP engine needs to create a copy of the container when you do an assignment-by-reference.

If we switch the order of assignments—first we assign $a by reference to $b and then we assign $a by value to $c—then something similar happens. Figure 6 shows how this is handled. In the first step we assign the variable $a to the string "this is" and then we proceed to assign $a by reference to variable $b. We now have one variable container where is_ref is 1 and refcount is 2. In step 2, we assign variable $a by value to variable $c, now a copy of the variable container is made in order for the PHP engine to be able to handle modifications to the variables, correctly, with the same reasons as stated in the previous paragraph.

But if you go back to step 2 of Figure 2, where we assign the variable $a to both $b and $c, you see that no copy is made here.

Passing References to Functions

Variables can also be passed-by-reference to functions. This is useful when a function needs to modify the value of a specific variable when it is called. The script in Figure 7 is a slightly modified version of the script that you have already seen in Figure 3. The only difference is the ampersand (&) in front of the $s variable in the declaration of the function do_something(). This ampersand instructs the PHP engine that the variable to which the ampersand is applied is going to be passed by reference and not by value. A different name for a passed-by-reference variable is an "out variable".

When a variable is passed by reference to a function the new variable in the function's symbol table is pointed to the old container and the refcount value is increased by 2 (one for the symbol table, and one for the stack). Just as in a normal assignment-by-reference the is_ref value inside the variable container is also set to 1 as you can see in step 2. From here on, the same things happen as with a normal reference like in step 3,

where no copy of the variable container is made if we assign a new value to the variable $s.

The return $s; statement is basically the same as the $c = $a statement in step 2 of Figure 6. The global variable $a and the local variable $s are both references to the same variable container and the logic dictates that if is_ref is set to 1 for a specific container and this container is assigned to another variable by-value, the container does not need to be duplicated. This is exactly what happens here, except that the newly created variable is created in the global symbol table by the assignment of the return value of the function with the statement $b = do_something($s).

Returning by Reference

Another feature in PHP is the ability to "return by reference". This is useful, for example, if you want to select a variable for modification with a function, such as selecting an array element or a node in a tree structure. In Figure 8 we show how returning by references work by means of an example. In this example (step 1), we define a $tree variable (which is actually not a tree, but a simple array) that contains three elements. The three elements have key values of 1, 2 and 3, and all of them point to a string describing the English word that matches with the key's value (ie. one, two and three).

This array gets passed to the find_node() function by reference, along with the key of the element that the find_node() function should look for and return. We need to pass by reference here, otherwise we can not return a reference to one of the elements, as we will be returning a reference to a copy of the $tree. When $tree is passed to the function it has a refcount of 3 and is_ref is set to 1. Nothing new here.

The first statement in the function, $item =& $node[$key], causes a new variable to be created in the symbol table of the function, which points to the array element where the key is "3" (because the variable $key is set to 3). In this step 3 you see that the creation of the $item by assigning it by reference to the array element causes the refcount value of the variable container that belongs to the array element to be increased by 1. The is_ref value of that variable container is now 1, too, of course.

The interesting things happen in step 4 where we return $item (by reference) back to the calling scope and assign it (by reference) to $node. This causes the refcount of the variable container to which the 3rd array key points to be set to 3. At this point $tree[3], $item (from the function's scope) and $node (global scope) all point to this variable container. When the symbol table of the function is destroyed (in step 5), the refcount value decreases from 1 to 2. $node is now a reference to the third element in the array.

If the variable $item would not have been assigned by reference to the return value of the do_something()

Figure 7

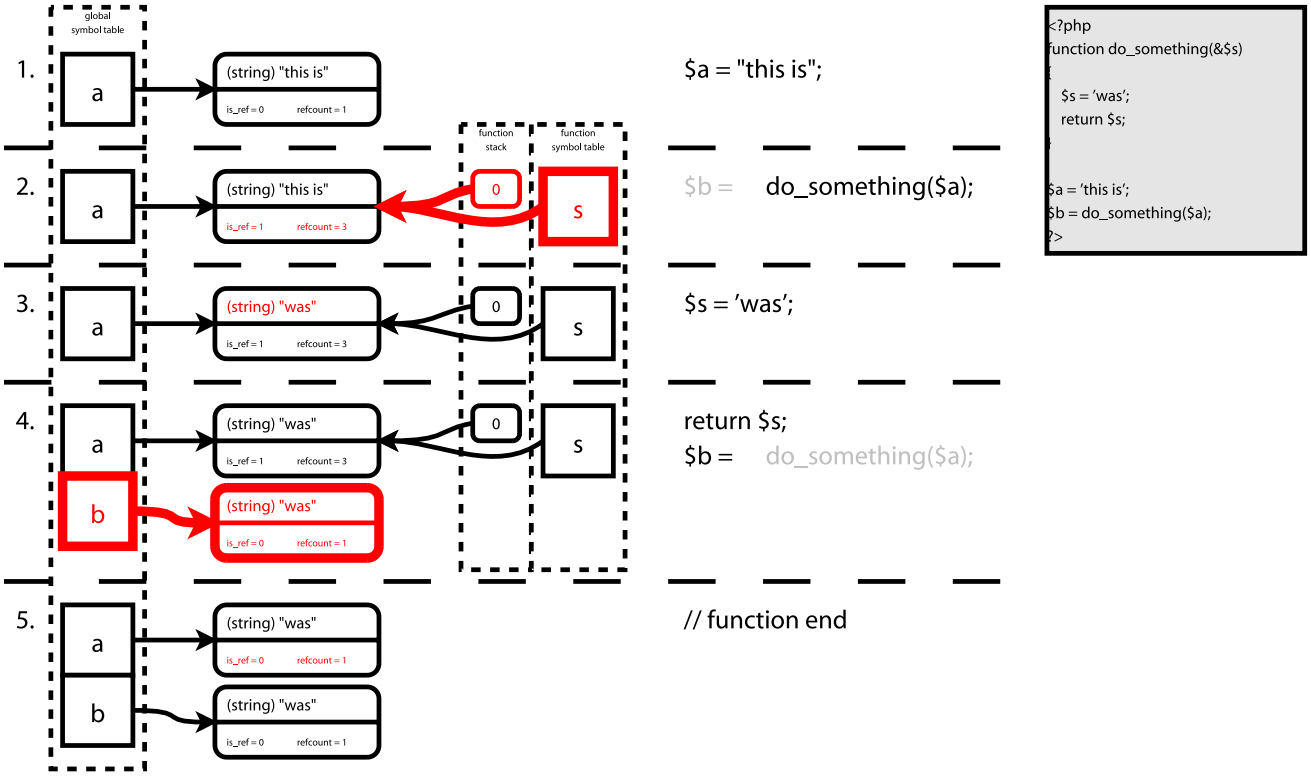
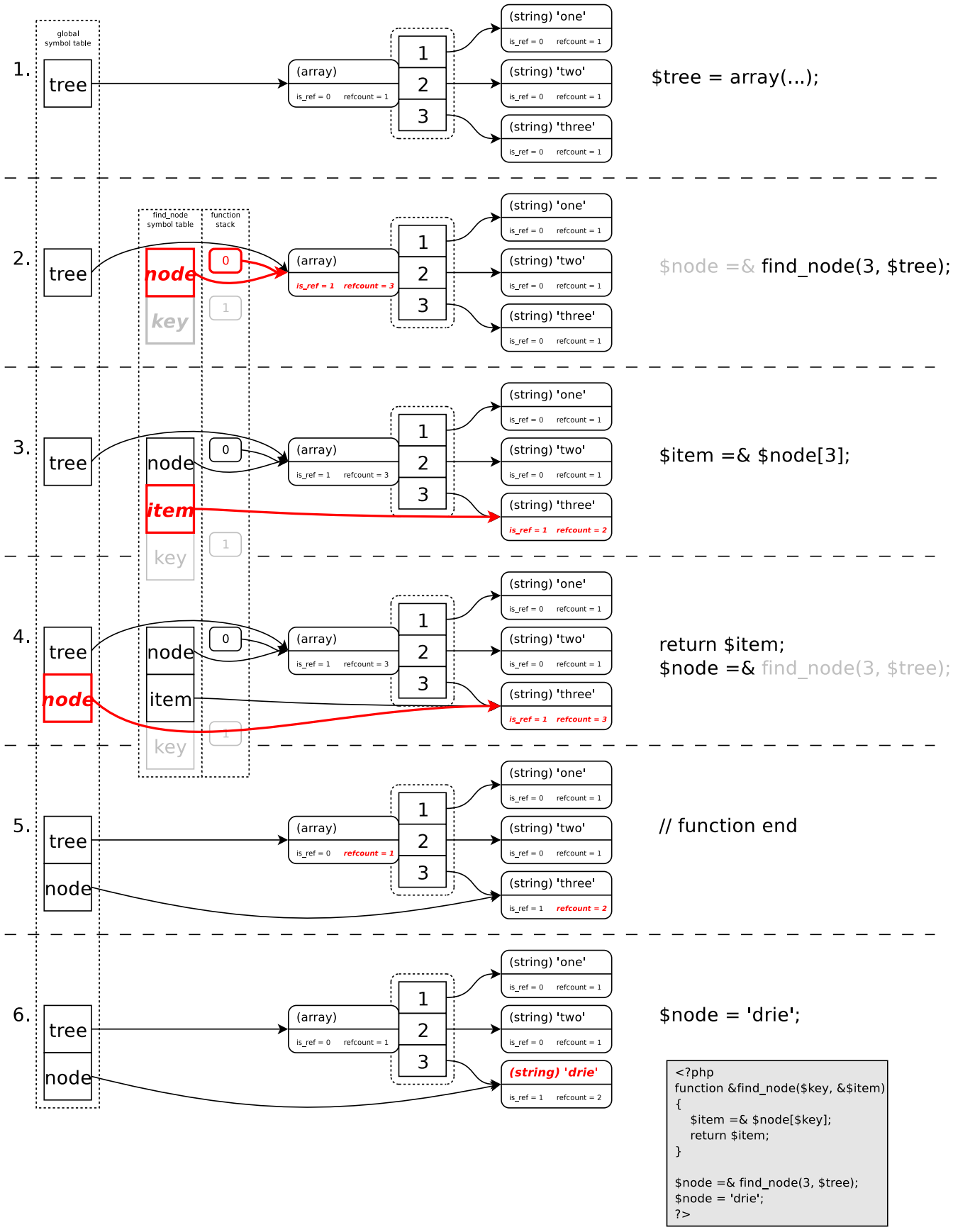


Figure 8


1.
global symbol table
tree
(array)
is_ref = 0 refcount = 1
1
2
3
(string) 'one'
is_ref = 0 refcount = 1
(string) 'two'
is_ref = 0 refcount = 1
(string) 'three'
is_ref = 0 refcount = 1
$tree = array(...);
2.
find_node symbol table
function stack
node
key
0
1
is_ref = 1 refcount = 3
$node =& find_node(3, $tree);
3.
item
is_ref = 1 refcount = 2
$item =& $node[3];
4.
return $item;
$node =& find_node(3, $tree);
5.
refcount = 1
refcount = 2
// function end
6.
(string) 'drie'
$node = 'drie';
<?php
function &find_node($key, &$item)
{
$item =& $node[$key];
return $item;
}
$node =& find_node(3, $tree);
$node = 'drie';
?>

function, but instead would have been assigned by value, then `$node` would not have been a reference to `$tree[3]`. In this case, the `refcount` value of the variable container to which `$tree[3]` points is then 1 after the function ends, but for some strange reason the `is_ref` value is not reset to 0 as you might expect. My tests did not find any problems with this, though, in this simple example. If the function `do_something()` would not have been a "return-by-reference function", then again the `$node` variable would not be a reference to `$tree[3]`. In this case, the `is_ref` value of the variable container would have been reset to 0.

Finally, in step 6, we modify the value in the variable container to which both `$node` and `$tree[3]` point.

Please do note that it *is* harmful not to accept a reference from a function that returns a reference. In some cases, PHP will get confused and cause memory corruptions which are very hard to find and debug. It is also not a good idea to return a static value as reference, as the PHP engine has problems with that too. In PHP 4.3, both cases can lead to very hard to reproduce bugs and crashes of PHP and the web server. In PHP 5, this works all a little bit better. Here you can expect a warning and it will behave "properly". Hopefully, a backported fix for this problem makes it into a new minor version of PHP 4—PHP 4.4.

The Global Keyword

PHP has a feature that allows the use of a global variable inside a function: you can make this connection with the `global` keyword. This keyword will create a reference between the local variable and the global one. Figure 9 shows this in an example.

In step 1 and 2, we create the variable `$var` and call

Figure 9

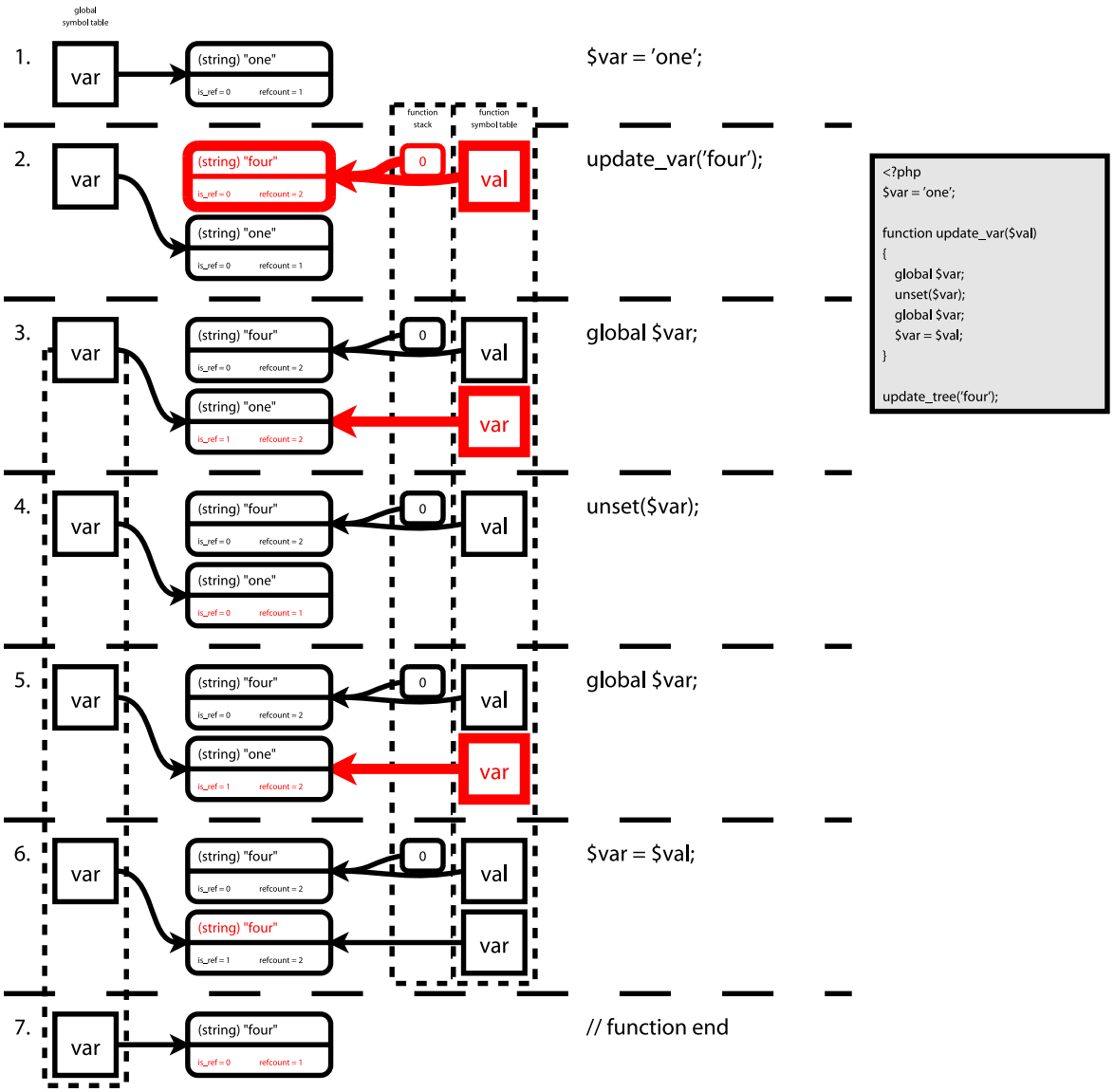
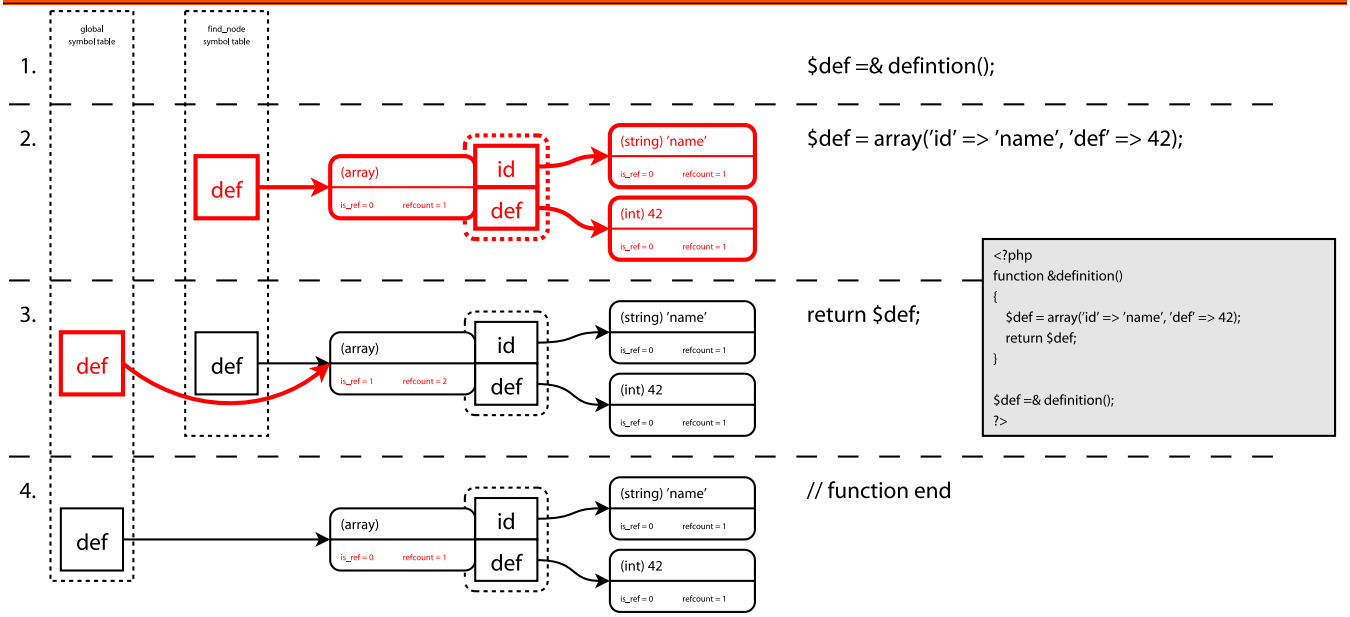


Figure 10



the function update_var() with the string literal "one" as the sole parameter. At this point, we have two variable containers. The first one is pointed to from the global variable $var, and the second one is the $val variable in the called function. The latter variable container has a refcount value of 2, as both the variable on the stack and the local variable $val point to it.

The global $var statement, in the function, creates a new variable in the local scope, which is created as a reference to the variable with the same name in the global scope. As you can see in step 3, this increases the refcount of the variable container from 1 to 2 and this also sets the is_ref value to 1.

In step 4, we unset the variable $var. Against some people's expectation, the global variable $var does not get unset—as the unset() was done on a reference to the global variable $var and not that variable itself. To reestablish the reference, we employ the global keyword, again in step 5. As you can see, we have re-created the same situation as in step 3. Instead of using global $var we could just as well have used $var =& $GLOBALS['var'] as it would have created the exact same situation.

In step 6, we continue to reassign the $var variable to the function's $val argument. This changes the value to which both the global variable $var and the local variable $var point; this is what you would expect from a referenced variable. When the function ends, in step 7, the reference from the variable in the scope of the function disappears, and we end up with one variable container with a refcount of 1 and an is_ref value of 0.

Listing 1

```
1 <?php
2     function &split_list($emails)
3     {
4         $emails =& preg_split("/[,;]/", $emails);
5         return $emails;
6     }
7
8     $emails =
split_list('derick@php.net;derick@derickrethans.nl;dr@ez.no');
9 ?>
10
```

Abusing References

In this section, I will give a few examples that show you how references should not be used—in some cases these examples might even create memory corruptions in PHP 4.3 and lower.

Example 1: "Returning static values by-reference". In Figure 10, we have a very small script with a return-by-reference function called definition(). This function simply returns an array that contains some elements. Returning by reference makes no sense here, as the exact same things would happen internally if the variable container holding the array was returned by value, except that in the intermediate step (step 3) the is_ref value of the container would not be set to 1, of course. In case the $def variable in the function's scope would have been referenced by another variable, something that might happen in a class method where you do $def = $this->def then the return-by-reference properties of the function would have copied the array, because this creates a similar situation as in step 2 of Figure 5.

Example 2: "Accepting references from a function that doesn't return references". This is potentially dan-

gerous; PHP 4.3 (and lower) does not handle this properly. In Listing 1, you see an example of something that is not going to work properly. This function was implemented with performance in mind, trying not to copy variable containers by using references. As you should know after reading this article, this is not going to buy you anything. There are a few reasons why it doesn't work. The first reason is that the PHP internal function `preg_split()` does not return by reference—actually, no internal function in PHP can return anything by reference. So, assigning the return value by reference from a function that doesn't return a reference is pointless. The second reason why there is no performance benefit, here, is the same one as in Example 1, in the previous paragraph: you're returning a static value—not a reference to a variable—it does not make sense to make the `split_list()` function to return-by-reference.

Conclusion

After reading this article, I hope that you now fully understand how references, refcounting, and variables work in PHP. It should also have explained that assigning by reference does not always save you memory—it's better to let the PHP engine handle this optimization. Do not try to outsmart PHP yourself here and only use references when they are really needed.

In PHP 4.3, there are still some problems with references, for which patches are in the works. These patches are backports from PHP 5-specific code, and although they work fine, they will break binary compatibility—meaning that compiled extensions no longer work after those patches are put into PHP. In my opinion, those hard to produce memory corruption errors should be fixed in PHP 4 too, though, so perhaps this creates the need for a PHP 4.4 release. If you're having problems, you can try to use the patch located at *http://files.derickrethans.nl/patches/ze1-return-reference-20050429.diff.txt*

The PHP Manual also has some information on references, although it does not explain the internals very well. The URL for the section in PHP's Manual is *http://php.net/language.references*

About the Author ?>

Derick Rethans provides solutions for Internet related problems. He has contributed in a number of ways to the PHP project, including the mcrypt extension, bug fixes, additions and leading the QA team. He now works as developer for eZ systems A.S.. In his spare time he likes to work on SRM: Script Running Machine and Xdebug, watch movies and travel. You can reach him at derick@derickrethans.nl

To Discuss this article:
http://forums.phparch.com/228