## Introduction

Any PHP script is built out of a series of statements. A statement can be an assignment, a function call, a loop, a conditional statement or even a statement that does nothing (an empty statement). Statements usually end with a semicolon. In addition, statements can be grouped into a statement-group by encapsulating a group of statements with curly braces. A statement-group is a statement by itself as well. The various statement types are described in this chapter.

## if

(PHP 4, PHP 5)

The if construct is one of the most important features of many languages, PHP included. It allows for conditional execution of code fragments. PHP features an if structure that is similar to that of C:

if (expr)

statement

As described in [the section about expressions](http://php.net/manual/en/language.expressions.php), expression is evaluated to its Boolean value. If expression evaluates to **TRUE**, PHP will execute statement, and if it evaluates to **FALSE** - it'll ignore it. More information about what values evaluate to **FALSE** can be found in the ['Converting to boolean'](http://php.net/manual/en/language.types.boolean.php#language.types.boolean.casting) section.

The following example would display a is bigger than b if $a is bigger than $b:

<?php  
if ($a > $b)  
  echo "a is bigger than b";  
?>

Often you'd want to have more than one statement to be executed conditionally. Of course, there's no need to wrap each statement with an if clause. Instead, you can group several statements into a statement group. For example, this code would display a is bigger than b if $a is bigger than $b, and would then assign the value of $a into $b:

<?php  
if ($a > $b) {  
  echo "a is bigger than b";  
  $b = $a;  
}  
?>

If statements can be nested infinitely within other if statements, which provides you with complete flexibility for conditional execution of the various parts of your program.

## else

(PHP 4, PHP 5)

Often you'd want to execute a statement if a certain condition is met, and a different statement if the condition is not met. This is what else is for. else extends an if statement to execute a statement in case the expression in the if statement evaluates to **FALSE**. For example, the following code would display a is greater than b if $a is greater than $b, and a is NOT greater than b otherwise:

<?php  
if ($a > $b) {  
  echo "a is greater than b";  
} else {  
  echo "a is NOT greater than b";  
}  
?>

The else statement is only executed if the if expression evaluated to **FALSE**, and if there were any elseif expressions - only if they evaluated to **FALSE** as well (see [elseif](http://php.net/manual/en/control-structures.elseif.php)).

## elseif/else if

(PHP 4, PHP 5)

elseif, as its name suggests, is a combination of if and else. Like else, it extends an if statement to execute a different statement in case the original if expression evaluates to **FALSE**. However, unlike else, it will execute that alternative expression only if the elseif conditional expression evaluates to **TRUE**. For example, the following code would display a is bigger than b, a equal to b or a is smaller than b:

<?php  
if ($a > $b) {  
    echo "a is bigger than b";  
} elseif ($a == $b) {  
    echo "a is equal to b";  
} else {  
    echo "a is smaller than b";  
}  
?>

There may be several elseifs within the same if statement. The first elseif expression (if any) that evaluates to **TRUE** would be executed. In PHP, you can also write 'else if' (in two words) and the behavior would be identical to the one of 'elseif' (in a single word). The syntactic meaning is slightly different (if you're familiar with C, this is the same behavior) but the bottom line is that both would result in exactly the same behavior.

The elseif statement is only executed if the preceding if expression and any preceding elseif expressions evaluated to **FALSE**, and the current elseif expression evaluated to **TRUE**.

**Note**: Note that elseif and else if will only be considered exactly the same when using curly brackets as in the above example. When using a colon to define your if/elseif conditions, you must not separate else if into two words, or PHP will fail with a parse error.

<?php  
  
/\* Incorrect Method: \*/  
if($a > $b):  
    echo $a." is greater than ".$b;  
else if($a == $b): // Will not compile.  
    echo "The above line causes a parse error.";  
endif;  
  
  
/\* Correct Method: \*/  
if($a > $b):  
    echo $a." is greater than ".$b;  
elseif($a == $b): // Note the combination of the words.  
    echo $a." equals ".$b;  
else:  
    echo $a." is neither greater than or equal to ".$b;  
endif;  
  
?>

## Alternative syntax for control structures

(PHP 4, PHP 5)

PHP offers an alternative syntax for some of its control structures; namely, if, while, for, foreach, and switch. In each case, the basic form of the alternate syntax is to change the opening brace to a colon (:) and the closing brace to endif;, endwhile;, endfor;, endforeach;, or endswitch;, respectively.

<?php if ($a == 5): ?>  
A is equal to 5  
<?php endif; ?>

In the above example, the HTML block "A is equal to 5" is nested within an if statement written in the alternative syntax. The HTML block would be displayed only if $a is equal to 5.

The alternative syntax applies to else and elseif as well. The following is an if structure with elseif and else in the alternative format:

<?php  
if ($a == 5):  
    echo "a equals 5";  
    echo "...";  
elseif ($a == 6):  
    echo "a equals 6";  
    echo "!!!";  
else:  
    echo "a is neither 5 nor 6";  
endif;  
?>

**Note**:

Mixing syntaxes in the same control block is not supported.

**Warning**

Any output (including whitespace) between a switch statement and the first case will result in a syntax error. For example, this is invalid:

<?php switch ($foo): ?>  
    <?php case 1: ?>  
    ...  
<?php endswitch ?>

Whereas this is valid, as the trailing newline after the switch statement is considered part of the closing ?> and hence nothing is output between the switch and case:

<?php switch ($foo): ?>  
<?php case 1: ?>  
    ...  
<?php endswitch ?>

See also [while](http://php.net/manual/en/control-structures.while.php), [for](http://php.net/manual/en/control-structures.for.php), and [if](http://php.net/manual/en/control-structures.if.php) for further examples.

## while

(PHP 4, PHP 5)

while loops are the simplest type of loop in PHP. They behave just like their C counterparts. The basic form of a while statement is:

while (expr)

statement

The meaning of a while statement is simple. It tells PHP to execute the nested statement(s) repeatedly, as long as the while expression evaluates to **TRUE**. The value of the expression is checked each time at the beginning of the loop, so even if this value changes during the execution of the nested statement(s), execution will not stop until the end of the iteration (each time PHP runs the statements in the loop is one iteration). Sometimes, if the while expression evaluates to **FALSE** from the very beginning, the nested statement(s) won't even be run once.

Like with the if statement, you can group multiple statements within the same while loop by surrounding a group of statements with curly braces, or by using the alternate syntax:

while (expr):

statement

...

endwhile;

The following examples are identical, and both print the numbers 1 through 10:

<?php  
/\* example 1 \*/  
  
$i = 1;  
while ($i <= 10) {  
    echo $i++;  /\* the printed value would be  
                   $i before the increment  
                   (post-increment) \*/  
}  
  
/\* example 2 \*/  
  
$i = 1;  
while ($i <= 10):  
    echo $i;  
    $i++;  
endwhile;  
?>

## do-while [¶](http://php.net/manual/en/control-structures.do.while.php#control-structures.do.while)

(PHP 4, PHP 5)

do-while loops are very similar to while loops, except the truth expression is checked at the end of each iteration instead of in the beginning. The main difference from regular while loops is that the first iteration of a do-while loop is guaranteed to run (the truth expression is only checked at the end of the iteration), whereas it may not necessarily run with a regular while loop (the truth expression is checked at the beginning of each iteration, if it evaluates to **FALSE** right from the beginning, the loop execution would end immediately).

There is just one syntax for do-while loops:

<?php  
$i = 0;  
do {  
    echo $i;  
} while ($i > 0);  
?>

The above loop would run one time exactly, since after the first iteration, when truth expression is checked, it evaluates to **FALSE** ($i is not bigger than 0) and the loop execution ends.

Advanced C users may be familiar with a different usage of the do-while loop, to allow stopping execution in the middle of code blocks, by encapsulating them with do-while (0), and using the [break](http://php.net/manual/en/control-structures.break.php) statement. The following code fragment demonstrates this:

<?php  
do {  
    if ($i < 5) {  
        echo "i is not big enough";  
        break;  
    }  
    $i \*= $factor;  
    if ($i < $minimum\_limit) {  
        break;  
    }  
   echo "i is ok";  
  
    /\* process i \*/  
  
} while (0);  
?>

Don't worry if you don't understand this right away or at all. You can code scripts and even powerful scripts without using this 'feature'. Since PHP 5.3.0, it is possible to use [goto](http://php.net/manual/en/control-structures.goto.php) operator instead of this hack.

## for

(PHP 4, PHP 5)

for loops are the most complex loops in PHP. They behave like their C counterparts. The syntax of a for loop is:

for (expr1; expr2; expr3)

statement

The first expression (expr1) is evaluated (executed) once unconditionally at the beginning of the loop.

In the beginning of each iteration, expr2 is evaluated. If it evaluates to **TRUE**, the loop continues and the nested statement(s) are executed. If it evaluates to **FALSE**, the execution of the loop ends.

At the end of each iteration, expr3 is evaluated (executed).

Each of the expressions can be empty or contain multiple expressions separated by commas. In expr2, all expressions separated by a comma are evaluated but the result is taken from the last part. expr2 being empty means the loop should be run indefinitely (PHP implicitly considers it as **TRUE**, like C). This may not be as useless as you might think, since often you'd want to end the loop using a conditional [break](http://php.net/manual/en/control-structures.break.php) statement instead of using the for truth expression.

Consider the following examples. All of them display the numbers 1 through 10:

<?php  
/\* example 1 \*/  
  
for ($i = 1; $i <= 10; $i++) {  
    echo $i;  
}  
  
/\* example 2 \*/  
  
for ($i = 1; ; $i++) {  
    if ($i > 10) {  
        break;  
    }  
    echo $i;  
}  
  
/\* example 3 \*/  
  
$i = 1;  
for (; ; ) {  
    if ($i > 10) {  
        break;  
    }  
    echo $i;  
    $i++;  
}  
  
/\* example 4 \*/  
  
for ($i = 1, $j = 0; $i <= 10; $j += $i, print $i, $i++);  
?>

Of course, the first example appears to be the nicest one (or perhaps the fourth), but you may find that being able to use empty expressions in for loops comes in handy in many occasions.

PHP also supports the alternate "colon syntax" for for loops.

for (expr1; expr2; expr3):

statement

...

endfor;

It's a common thing to many users to iterate through arrays like in the example below.

<?php  
/\*  
 \* This is an array with some data we want to modify  
 \* when running through the for loop.  
 \*/  
$people = array(  
    array('name' => 'Kalle', 'salt' => 856412),  
    array('name' => 'Pierre', 'salt' => 215863)  
);  
  
for($i = 0; $i < count($people); ++$i) {  
    $people[$i]['salt'] = mt\_rand(000000, 999999);  
}  
?>

The above code can be slow, because the array size is fetched on every iteration. Since the size never changes, the loop be easily optimized by using an intermediate variable to store the size instead of repeatedly calling [count()](http://php.net/manual/en/function.count.php):

<?php  
$people = array(  
    array('name' => 'Kalle', 'salt' => 856412),  
    array('name' => 'Pierre', 'salt' => 215863)  
);  
  
for($i = 0, $size = count($people); $i < $size; ++$i) {  
    $people[$i]['salt'] = mt\_rand(000000, 999999);  
}  
?>

## foreach

(PHP 4, PHP 5)

The foreach construct provides an easy way to iterate over arrays. foreach works only on arrays and objects, and will issue an error when you try to use it on a variable with a different data type or an uninitialized variable. There are two syntaxes:

foreach (array\_expression as $value)

statement

foreach (array\_expression as $key => $value)

statement

The first form loops over the array given by array\_expression. On each iteration, the value of the current element is assigned to $value and the internal array pointer is advanced by one (so on the next iteration, you'll be looking at the next element).

The second form will additionally assign the current element's key to the $key variable on each iteration.

It is possible to [customize object iteration](http://php.net/manual/en/language.oop5.iterations.php).

**Note**:

When foreach first starts executing, the internal array pointer is automatically reset to the first element of the array. This means that you do not need to call [reset()](http://php.net/manual/en/function.reset.php) before a foreach loop.

As foreach relies on the internal array pointer, changing it within the loop may lead to unexpected behavior.

In order to be able to directly modify array elements within the loop precede $value with &. In that case the value will be assigned by [reference](http://php.net/manual/en/language.references.php).

<?php  
$arr = array(1, 2, 3, 4);  
foreach ($arr as &$value) {  
    $value = $value \* 2;  
}  
// $arr is now array(2, 4, 6, 8)  
unset($value); // break the reference with the last element  
?>

Referencing $value is only possible if the iterated array can be referenced (i.e. if it is a variable). The following code won't work:

<?php  
foreach (array(1, 2, 3, 4) as &$value) {  
    $value = $value \* 2;  
}  
?>

**Warning**

Reference of a $value and the last array element remain even after the foreach loop. It is recommended to destroy it by [unset()](http://php.net/manual/en/function.unset.php).

**Note**:

foreach does not support the ability to suppress error messages using '@'.

You may have noticed that the following are functionally identical:

<?php  
$arr = array("one", "two", "three");  
reset($arr);  
while (list(, $value) = each($arr)) {  
    echo "Value: $value<br />\n";  
}  
  
foreach ($arr as $value) {  
    echo "Value: $value<br />\n";  
}  
?>

The following are also functionally identical:

<?php  
$arr = array("one", "two", "three");  
reset($arr);  
while (list($key, $value) = each($arr)) {  
    echo "Key: $key; Value: $value<br />\n";  
}  
  
foreach ($arr as $key => $value) {  
    echo "Key: $key; Value: $value<br />\n";  
}  
?>

Some more examples to demonstrate usage:

<?php  
/\* foreach example 1: value only \*/  
  
$a = array(1, 2, 3, 17);  
  
foreach ($a as $v) {  
    echo "Current value of \$a: $v.\n";  
}  
  
/\* foreach example 2: value (with its manual access notation printed for illustration) \*/  
  
$a = array(1, 2, 3, 17);  
  
$i = 0; /\* for illustrative purposes only \*/  
  
foreach ($a as $v) {  
    echo "\$a[$i] => $v.\n";  
    $i++;  
}  
  
/\* foreach example 3: key and value \*/  
  
$a = array(  
    "one" => 1,  
    "two" => 2,  
    "three" => 3,  
    "seventeen" => 17  
);  
  
foreach ($a as $k => $v) {  
    echo "\$a[$k] => $v.\n";  
}  
  
/\* foreach example 4: multi-dimensional arrays \*/  
$a = array();  
$a[0][0] = "a";  
$a[0][1] = "b";  
$a[1][0] = "y";  
$a[1][1] = "z";  
  
foreach ($a as $v1) {  
    foreach ($v1 as $v2) {  
        echo "$v2\n";  
    }  
}  
  
/\* foreach example 5: dynamic arrays \*/  
  
foreach (array(1, 2, 3, 4, 5) as $v) {  
    echo "$v\n";  
}  
?>

### Unpacking nested arrays with list()

(PHP 5 >= 5.5.0)

PHP 5.5 added the ability to iterate over an array of arrays and unpack the nested array into loop variables by providing a [list()](http://php.net/manual/en/function.list.php) as the value.

For example:

<?php  
$array = [  
    [1, 2],  
    [3, 4],  
];  
  
foreach ($array as list($a, $b)) {  
    // $a contains the first element of the nested array,  
    // and $b contains the second element.  
    echo "A: $a; B: $b\n";  
}  
?>

The above example will output:

A: 1; B: 2

A: 3; B: 4

You can provide fewer elements in the [list()](http://php.net/manual/en/function.list.php) than there are in the nested array, in which case the leftover array values will be ignored:

<?php  
$array = [  
    [1, 2],  
    [3, 4],  
];  
  
foreach ($array as list($a)) {  
    // Note that there is no $b here.  
    echo "$a\n";  
}  
?>

The above example will output:

1

3

A notice will be generated if there aren't enough array elements to fill the [list()](http://php.net/manual/en/function.list.php):

<?php  
$array = [  
    [1, 2],  
    [3, 4],  
];  
  
foreach ($array as list($a, $b, $c)) {  
    echo "A: $a; B: $b; C: $c\n";  
}  
?>

The above example will output:

Notice: Undefined offset: 2 in example.php on line 7

A: 1; B: 2; C:

Notice: Undefined offset: 2 in example.php on line 7

A: 3; B: 4; C:

## break

(PHP 4, PHP 5)

break ends execution of the current for, foreach, while, do-while or switch structure.

break accepts an optional numeric argument which tells it how many nested enclosing structures are to be broken out of.

<?php  
$arr = array('one', 'two', 'three', 'four', 'stop', 'five');  
while (list(, $val) = each($arr)) {  
    if ($val == 'stop') {  
        break;    /\* You could also write 'break 1;' here. \*/  
    }  
    echo "$val<br />\n";  
}  
  
/\* Using the optional argument. \*/  
  
$i = 0;  
while (++$i) {  
    switch ($i) {  
    case 5:  
        echo "At 5<br />\n";  
        break 1;  /\* Exit only the switch. \*/  
    case 10:  
        echo "At 10; quitting<br />\n";  
        break 2;  /\* Exit the switch and the while. \*/  
    default:  
        break;  
    }  
}  
?>

| **Changelog for break** | |
| --- | --- |
| **Version** | **Description** |
| 5.4.0 | break 0; is no longer valid. In previous versions it was interpreted the same as break 1;. |
| 5.4.0 | Removed the ability to pass in variables (e.g., $num = 2; break $num;) as the numerical argument. |

## continue

(PHP 4, PHP 5)

continue is used within looping structures to skip the rest of the current loop iteration and continue execution at the condition evaluation and then the beginning of the next iteration.

**Note**: Note that in PHP the [switch](http://php.net/manual/en/control-structures.switch.php) statement is considered a looping structure for the purposes of continue.

continue accepts an optional numeric argument which tells it how many levels of enclosing loops it should skip to the end of. The default value is 1, thus skipping to the end of the current loop.

<?php  
while (list($key, $value) = each($arr)) {  
    if (!($key % 2)) { // skip odd members  
        continue;  
    }  
    do\_something\_odd($value);  
}  
  
$i = 0;  
while ($i++ < 5) {  
    echo "Outer<br />\n";  
    while (1) {  
        echo "Middle<br />\n";  
        while (1) {  
            echo "Inner<br />\n";  
            continue 3;  
        }  
        echo "This never gets output.<br />\n";  
    }  
    echo "Neither does this.<br />\n";  
}  
?>

Omitting the semicolon after continue can lead to confusion. Here's an example of what you shouldn't do.

<?php  
for ($i = 0; $i < 5; ++$i) {  
    if ($i == 2)  
        continue  
    print "$i\n";  
}  
?>

One can expect the result to be:

0

1

3

4

but this script will output:

2

because the entire continue print "$i\n"; is evaluated as a single expression, and so [print](http://php.net/manual/en/function.print.php) is called only when $i == 2 is true. (The return value of print is passed to continue as the numeric argument.)

| **Changelog for continue** | |
| --- | --- |
| **Version** | **Description** |
| 5.4.0 | continue 0; is no longer valid. In previous versions it was interpreted the same as continue 1;. |
| 5.4.0 | Removed the ability to pass in variables (e.g., $num = 2; continue $num;) as the numerical argument. |

## switch

(PHP 4, PHP 5)

The switch statement is similar to a series of IF statements on the same expression. In many occasions, you may want to compare the same variable (or expression) with many different values, and execute a different piece of code depending on which value it equals to. This is exactly what the switch statement is for.

**Note**: Note that unlike some other languages, the [continue](http://php.net/manual/en/control-structures.continue.php) statement applies to switch and acts similar to break. If you have a switch inside a loop and wish to continue to the next iteration of the outer loop, use continue 2.

**Note**:

Note that switch/case does [loose comparison](http://php.net/manual/en/types.comparisons.php#types.comparisions-loose).

The following two examples are two different ways to write the same thing, one using a series of if and elseif statements, and the other using the switch statement:

**Example #1 switch structure**

<?php  
if ($i == 0) {  
    echo "i equals 0";  
} elseif ($i == 1) {  
    echo "i equals 1";  
} elseif ($i == 2) {  
    echo "i equals 2";  
}  
  
switch ($i) {  
    case 0:  
        echo "i equals 0";  
        break;  
    case 1:  
        echo "i equals 1";  
        break;  
    case 2:  
        echo "i equals 2";  
        break;  
}  
?>

**Example #2 switch structure allows usage of strings**

<?php  
switch ($i) {  
    case "apple":  
        echo "i is apple";  
        break;  
    case "bar":  
        echo "i is bar";  
        break;  
    case "cake":  
        echo "i is cake";  
        break;  
}  
?>

It is important to understand how the switch statement is executed in order to avoid mistakes. The switch statement executes line by line (actually, statement by statement). In the beginning, no code is executed. Only when a case statement is found with a value that matches the value of the switch expression does PHP begin to execute the statements. PHP continues to execute the statements until the end of the switch block, or the first time it sees a break statement. If you don't write a break statement at the end of a case's statement list, PHP will go on executing the statements of the following case. For example:

<?php  
switch ($i) {  
    case 0:  
        echo "i equals 0";  
    case 1:  
        echo "i equals 1";  
    case 2:  
        echo "i equals 2";  
}  
?>

Here, if $i is equal to 0, PHP would execute all of the echo statements! If $i is equal to 1, PHP would execute the last two echo statements. You would get the expected behavior ('i equals 2' would be displayed) only if $i is equal to 2. Thus, it is important not to forget break statements (even though you may want to avoid supplying them on purpose under certain circumstances).

In a switch statement, the condition is evaluated only once and the result is compared to each case statement. In an elseif statement, the condition is evaluated again. If your condition is more complicated than a simple compare and/or is in a tight loop, a switch may be faster.

The statement list for a case can also be empty, which simply passes control into the statement list for the next case.

<?php  
switch ($i) {  
case 0:  
case 1:  
case 2:  
    echo "i is less than 3 but not negative";  
    break;  
case 3:  
    echo "i is 3";  
}  
?>

A special case is the default case. This case matches anything that wasn't matched by the other cases. For example:

<?php  
switch ($i) {  
    case 0:  
        echo "i equals 0";  
        break;  
    case 1:  
        echo "i equals 1";  
        break;  
    case 2:  
        echo "i equals 2";  
        break;  
    default:  
       echo "i is not equal to 0, 1 or 2";  
}  
?>

The case expression may be any expression that evaluates to a simple type, that is, integer or floating-point numbers and strings. Arrays or objects cannot be used here unless they are dereferenced to a simple type.

The alternative syntax for control structures is supported with switches. For more information, see [Alternative syntax for control structures](http://php.net/manual/en/control-structures.alternative-syntax.php).

<?php  
switch ($i):  
    case 0:  
        echo "i equals 0";  
        break;  
    case 1:  
        echo "i equals 1";  
        break;  
    case 2:  
        echo "i equals 2";  
        break;  
    default:  
        echo "i is not equal to 0, 1 or 2";  
endswitch;  
?>

It's possible to use a semicolon instead of a colon after a case like:

<?php  
switch($beer)  
{  
    case 'tuborg';  
    case 'carlsberg';  
    case 'heineken';  
        echo 'Good choice';  
    break;  
    default;  
        echo 'Please make a new selection...';  
    break;  
}  
?>

## return

(PHP 4, PHP 5)

return returns program control to the calling module. Execution resumes at the statement following the called module's invocation.

If called from within a function, the return statement immediately ends execution of the current function, and returns its argument as the value of the function call. return also ends the execution of an [eval()](http://php.net/manual/en/function.eval.php) statement or script file.

If called from the global scope, then execution of the current script file is ended. If the current script file was [include](http://php.net/manual/en/function.include.php)d or [require](http://php.net/manual/en/function.require.php)d, then control is passed back to the calling file. Furthermore, if the current script file was [include](http://php.net/manual/en/function.include.php)d, then the value given to return will be returned as the value of the [include](http://php.net/manual/en/function.include.php) call. If return is called from within the main script file, then script execution ends. If the current script file was named by the [auto\_prepend\_file](http://php.net/manual/en/ini.core.php#ini.auto-prepend-file) or [auto\_append\_file](http://php.net/manual/en/ini.core.php#ini.auto-append-file) configuration options in php.ini, then that script file's execution is ended.

**Note**: Note that since return is a language construct and not a function, the parentheses surrounding its arguments are not required. It is common to leave them out, and you actually should do so as PHP has less work to do in this case.

**Note**: If no parameter is supplied, then the parentheses must be omitted and **NULL** will be returned. Calling return with parentheses but with no arguments will result in a parse error.

**Note**: You should never use parentheses around your return variable when returning by reference, as this will not work. You can only return variables by reference, not the result of a statement. If you use return ($a); then you're not returning a variable, but the result of the expression ($a) (which is, of course, the value of $a).

## require

(PHP 4, PHP 5)

require is identical to [include](http://php.net/manual/en/function.include.php) except upon failure it will also produce a fatal **E\_COMPILE\_ERROR** level error. In other words, it will halt the script whereas [include](http://php.net/manual/en/function.include.php) only emits a warning (**E\_WARNING**) which allows the script to continue.

See the [include](http://php.net/manual/en/function.include.php) documentation for how this works.

## include

(PHP 4, PHP 5)

The include statement includes and evaluates the specified file.

The documentation below also applies to [require](http://php.net/manual/en/function.require.php).

Files are included based on the file path given or, if none is given, the [include\_path](http://php.net/manual/en/ini.core.php#ini.include-path) specified. If the file isn't found in the [include\_path](http://php.net/manual/en/ini.core.php#ini.include-path), include will finally check in the calling script's own directory and the current working directory before failing. The include construct will emit a warning if it cannot find a file; this is different behavior from [require](http://php.net/manual/en/function.require.php), which will emit a fatal error.

If a path is defined — whether absolute (starting with a drive letter or \ on Windows, or / on Unix/Linux systems) or relative to the current directory (starting with . or ..) — the [include\_path](http://php.net/manual/en/ini.core.php#ini.include-path) will be ignored altogether. For example, if a filename begins with ../, the parser will look in the parent directory to find the requested file.

For more information on how PHP handles including files and the include path, see the documentation for [include\_path](http://php.net/manual/en/ini.core.php#ini.include-path).

When a file is included, the code it contains inherits the [variable scope](http://php.net/manual/en/language.variables.scope.php) of the line on which the include occurs. Any variables available at that line in the calling file will be available within the called file, from that point forward. However, all functions and classes defined in the included file have the global scope.

**Example #1 Basic include example**

vars.php  
<?php  
  
$color = 'green';  
$fruit = 'apple';  
  
?>  
  
test.php  
<?php  
  
echo "A $color $fruit"; // A  
  
include 'vars.php';  
  
echo "A $color $fruit"; // A green apple  
  
?>

If the include occurs inside a function within the calling file, then all of the code contained in the called file will behave as though it had been defined inside that function. So, it will follow the variable scope of that function. An exception to this rule are [magic constants](http://php.net/manual/en/language.constants.predefined.php) which are evaluated by the parser before the include occurs.

**Example #2 Including within functions**

<?php  
  
function foo()  
{  
    global $color;  
  
    include 'vars.php';  
  
    echo "A $color $fruit";  
}  
  
/\* vars.php is in the scope of foo() so     \*  
\* $fruit is NOT available outside of this  \*  
\* scope.  $color is because we declared it \*  
\* as global.                               \*/  
  
foo();                    // A green apple  
echo "A $color $fruit";   // A green  
  
?>

When a file is included, parsing drops out of PHP mode and into HTML mode at the beginning of the target file, and resumes again at the end. For this reason, any code inside the target file which should be executed as PHP code must be enclosed within [valid PHP start and end tags](http://php.net/manual/en/language.basic-syntax.phpmode.php).

If "[URL include wrappers](http://php.net/manual/en/filesystem.configuration.php#ini.allow-url-include)" are enabled in PHP, you can specify the file to be included using a URL (via HTTP or other supported wrapper - see [Supported Protocols and Wrappers](http://php.net/manual/en/wrappers.php) for a list of protocols) instead of a local pathname. If the target server interprets the target file as PHP code, variables may be passed to the included file using a URL request string as used with HTTP GET. This is not strictly speaking the same thing as including the file and having it inherit the parent file's variable scope; the script is actually being run on the remote server and the result is then being included into the local script.

**Warning**

Windows versions of PHP prior to PHP 4.3.0 do not support access of remote files via this function, even if [allow\_url\_fopen](http://php.net/manual/en/filesystem.configuration.php#ini.allow-url-fopen) is enabled.

**Example #3 include through HTTP**

<?php  
  
/\* This example assumes that www.example.com is configured to parse .php  
\* files and not .txt files. Also, 'Works' here means that the variables  
\* $foo and $bar are available within the included file. \*/  
  
// Won't work; file.txt wasn't handled by www.example.com as PHP  
include 'http://www.example.com/file.txt?foo=1&bar=2';  
  
// Won't work; looks for a file named 'file.php?foo=1&bar=2' on the  
// local filesystem.  
include 'file.php?foo=1&bar=2';  
  
// Works.  
include 'http://www.example.com/file.php?foo=1&bar=2';  
  
$foo = 1;  
$bar = 2;  
include 'file.txt';  // Works.  
include 'file.php';  // Works.  
  
?>

**Warning**

# Security warning

Remote file may be processed at the remote server (depending on the file extension and the fact if the remote server runs PHP or not) but it still has to produce a valid PHP script because it will be processed at the local server. If the file from the remote server should be processed there and outputted only, [readfile()](http://php.net/manual/en/function.readfile.php) is much better function to use. Otherwise, special care should be taken to secure the remote script to produce a valid and desired code.

See also [Remote files](http://php.net/manual/en/features.remote-files.php), [fopen()](http://php.net/manual/en/function.fopen.php) and [file()](http://php.net/manual/en/function.file.php) for related information.

Handling Returns: include returns FALSE on failure and raises a warning. Successful includes, unless overridden by the included file, return 1. It is possible to execute a [return](http://php.net/manual/en/function.return.php) statement inside an included file in order to terminate processing in that file and return to the script which called it. Also, it's possible to return values from included files. You can take the value of the include call as you would for a normal function. This is not, however, possible when including remote files unless the output of the remote file has [valid PHP start and end tags](http://php.net/manual/en/language.basic-syntax.phpmode.php) (as with any local file). You can declare the needed variables within those tags and they will be introduced at whichever point the file was included.

Because include is a special language construct, parentheses are not needed around its argument. Take care when comparing return value.

**Example #4 Comparing return value of include**

<?php  
// won't work, evaluated as include(('vars.php') == 'OK'), i.e. include('')  
if (include('vars.php') == 'OK') {  
    echo 'OK';  
}  
  
// works  
if ((include 'vars.php') == 'OK') {  
    echo 'OK';  
}  
?>

**Example #5 include and the** [**return**](http://php.net/manual/en/function.return.php) **statement**

return.php  
<?php  
  
$var = 'PHP';  
  
return $var;  
  
?>  
  
noreturn.php  
<?php  
  
$var = 'PHP';  
  
?>  
  
testreturns.php  
<?php  
  
$foo = include 'return.php';  
  
echo $foo; // prints 'PHP'  
  
$bar = include 'noreturn.php';  
  
echo $bar; // prints 1  
  
?>

$bar is the value 1 because the include was successful. Notice the difference between the above examples. The first uses [return](http://php.net/manual/en/function.return.php) within the included file while the other does not. If the file can't be included, **FALSE** is returned and **E\_WARNING** is issued.

If there are functions defined in the included file, they can be used in the main file independent if they are before [return](http://php.net/manual/en/function.return.php) or after. If the file is included twice, PHP 5 issues fatal error because functions were already declared, while PHP 4 doesn't complain about functions defined after [return](http://php.net/manual/en/function.return.php). It is recommended to use [include\_once](http://php.net/manual/en/function.include-once.php) instead of checking if the file was already included and conditionally return inside the included file.

Another way to "include" a PHP file into a variable is to capture the output by using the [Output Control Functions](http://php.net/manual/en/ref.outcontrol.php) with include. For example:

**Example #6 Using output buffering to include a PHP file into a string**

<?php  
$string = get\_include\_contents('somefile.php');  
  
function get\_include\_contents($filename) {  
    if (is\_file($filename)) {  
        ob\_start();  
        include $filename;  
        return ob\_get\_clean();  
    }  
    return false;  
}  
  
?>

In order to automatically include files within scripts, see also the [auto\_prepend\_file](http://php.net/manual/en/ini.core.php#ini.auto-prepend-file) and [auto\_append\_file](http://php.net/manual/en/ini.core.php#ini.auto-append-file) configuration options in php.ini.

**Note**: Because this is a language construct and not a function, it cannot be called using [variable functions](http://php.net/manual/en/functions.variable-functions.php).

See also [require](http://php.net/manual/en/function.require.php), [require\_once](http://php.net/manual/en/function.require-once.php), [include\_once](http://php.net/manual/en/function.include-once.php), [get\_included\_files()](http://php.net/manual/en/function.get-included-files.php), [readfile()](http://php.net/manual/en/function.readfile.php), [virtual()](http://php.net/manual/en/function.virtual.php), and [include\_path](http://php.net/manual/en/ini.core.php#ini.include-path).

## require\_once

(PHP 4, PHP 5)

The require\_once statement is identical to [require](http://php.net/manual/en/function.require.php) except PHP will check if the file has already been included, and if so, not include (require) it again.

See the [include\_once](http://php.net/manual/en/function.include-once.php) documentation for information about the \_once behaviour, and how it differs from its non \_once siblings.

## include\_once

(PHP 4, PHP 5)

The include\_once statement includes and evaluates the specified file during the execution of the script. This is a behavior similar to the [include](http://php.net/manual/en/function.include.php) statement, with the only difference being that if the code from a file has already been included, it will not be included again. As the name suggests, it will be included just once.

include\_once may be used in cases where the same file might be included and evaluated more than once during a particular execution of a script, so in this case it may help avoid problems such as function redefinitions, variable value reassignments, etc.

See the [include](http://php.net/manual/en/function.include.php) documentation for information about how this function works.

**Note**:

With PHP 4, \_once functionality differs with case-insensitive operating systems (like Windows) so for example:

**Example #1 include\_once with a case insensitive OS in PHP 4**

<?php  
include\_once "a.php"; // this will include a.php  
include\_once "A.php"; // this will include a.php again! (PHP 4 only)  
?>

This behaviour changed in PHP 5, so for example with Windows the path is normalized first so that C:\PROGRA~1\A.php is realized the same as C:\Program Files\a.php and the file is included just once.

## goto

(PHP 5 >= 5.3.0)



Image courtesy of [» xkcd](http://xkcd.com/292)

The goto operator can be used to jump to another section in the program. The target point is specified by a label followed by a colon, and the instruction is given as goto followed by the desired target label. This is not a full unrestricted goto. The target label must be within the same file and context, meaning that you cannot jump out of a function or method, nor can you jump into one. You also cannot jump into any sort of loop or switch structure. You may jump out of these, and a common use is to use a goto in place of a multi-level break.

**Example #1 goto example**

<?php  
goto a;  
echo 'Foo';  
   
a:  
echo 'Bar';  
?>

The above example will output:

Bar

**Example #2 goto loop example**

<?php  
for($i=0,$j=50; $i<100; $i++) {  
  while($j--) {  
    if($j==17) goto end;   
  }    
}  
echo "i = $i";  
end:  
echo 'j hit 17';  
?>

The above example will output:

j hit 17

**Example #3 This will not work**

<?php  
goto loop;  
for($i=0,$j=50; $i<100; $i++) {  
  while($j--) {  
    loop:  
  }  
}  
echo "$i = $i";  
?>

The above example will output:

Fatal error: 'goto' into loop or switch statement is disallowed in

script on line 2

**Note**:

The goto operator is available as of PHP 5.3.

EVAL Example

<?php  
$string = 'cup';  
$name = 'coffee';  
$str = 'This is a $string with my $name in it.';  
echo $str. "\n";  
eval("\$str = \"$str\";");  
echo $str. "\n";  
?>

**Simple Object Iteration**

<?php  
class MyClass  
{  
    public $var1 = 'value 1';  
    public $var2 = 'value 2';  
    public $var3 = 'value 3';  
  
    protected $protected = 'protected var';  
    private   $private   = 'private var';  
  
    function iterateVisible() {  
       echo "MyClass::iterateVisible:\n";  
       foreach($this as $key => $value) {  
           print "$key => $value\n";  
       }  
    }  
}  
  
$class = new MyClass();  
  
foreach($class as $key => $value) {  
    print "$key => $value\n";  
}  
echo "\n";  
  
  
$class->iterateVisible();  
  
?>

**Next Topics**

## Language Constructs and Functions

## Namespaces

## Extensions

## Config

## Performance/bytecode caching \*