Language Reference

Basic syntax

Escaping from HTML

When PHP parses a file, it looks for opening and closing tags, which tell PHP to start and stop interpreting the code between them. Parsing in this manner allows php to be embedded in all sorts of different documents, as everything outside of a pair of opening and closing tags is ignored by the PHP parser. Most of the time you will see php embedded in HTML documents, as in this example.

```
<p>This is going to be ignored.</p>
<?php echo 'While this is going to be parsed.'; ?>
<p>This will also be ignored.</p>
```

You can also use more advanced structures:

Example #1 - Advanced escaping

```
<?php
if ($expression) {
   ?>
   <strong>This is true.</strong>
   <?php
} else {
   ?>
   <strong>This is false.</strong>
   <?php
}
?>
```

This works as expected, because when PHP hits the ?> closing tags, it simply starts outputting whatever it finds (except for an immediately following newline - see instruction separation) until it hits another opening tag. The example given here is contrived, of course, but for outputting large blocks of text, dropping out of PHP parsing mode is generally more efficient than sending all of the text through echo() or print().

There are four different pairs of opening and closing tags which can be used in php. Two of those, <?php ?> and <script language="php"> </script>, are always available. The other two are short tags and ASP style tags, and can be turned on and off from the *php.ini* configuration file. As such, while some people find short tags and ASP style tags convenient, they are less portable, and generally not recommended.

Note

Also note that if you are embedding PHP within XML or XHTML you will need to use the <?php ?> tags to remain compliant with standards.

Example #2 - PHP Opening and Closing Tags

```
1.  <?php echo 'if you want to serve XHTML or XML documents, do like this';
?>

2.  <script language="php">
       echo 'some editors (like FrontPage) don\'t
             like processing instructions';
   </script>

3.  <? echo 'this is the simplest, an SGML processing instruction'; ?>
   <?= expression ?> This is a shortcut for "<? echo expression ?>"

4.  <% echo 'You may optionally use ASP-style tags'; %>
   <%= $variable; # This is a shortcut for "<% echo . . ." %>
```

While the tags seen in examples one and two are both always available, example one is the most commonly used, and recommended, of the two.

Short tags (example three) are only available when they are enabled via the short_open_tag *php.ini* configuration file directive, or if php was configured with the *--enable-short-tags* option.

ASP style tags (example four) are only available when they are enabled via the asp_tags *php.ini* configuration file directive.

Note

Using short tags should be avoided when developing applications or libraries that are meant for redistribution, or deployment on PHP servers which are not under your control, because short tags may not be supported on the target server. For portable, redistributable code, be sure not to use short tags.

Instruction separation

As in C or Perl, PHP requires instructions to be terminated with a semicolon at the end of each statement. The closing tag of a block of PHP code automatically implies a semicolon; you do not need to have a semicolon terminating the last line of a PHP block. The closing tag for the block will include the immediately trailing newline if one is present.

```
<?php
   echo 'This is a test';
?>

<?php echo 'This is a test' ?>
```

```
<?php echo 'We omitted the last closing tag';
```

Note
The closing tag of a PHP block at the end of a file is optional, and in some cases omitting it is helpful when using **include()** or **require()**, so unwanted whitespace will not occur at the end of files, and you will still be able to add headers to the response later. It is also handy if you use output buffering, and would not like to see added unwanted whitespace at the end of the parts generated by the included files.

Comments

PHP supports 'C', 'C++' and Unix shell-style (Perl style) comments. For example:

```
<?php
   echo 'This is a test'; // This is a one-line c++ style comment
   /* This is a multi line comment
      yet another line of comment */
   echo 'This is yet another test';
   echo 'One Final Test'; # This is a one-line shell-style comment
?>
```

The "one-line" comment styles only comment to the end of the line or the current block of PHP code, whichever comes first. This means that HTML code after *// ... ?>* or *# ... ?>* WILL be printed: ?> breaks out of PHP mode and returns to HTML mode, and *//* or *#* cannot influence that. If the asp_tags configuration directive is enabled, it behaves the same with *// %>* and *# %>*. However, the *</script>* tag doesn't break out of PHP mode in a one-line comment.

```
<h1>This is an <?php # echo 'simple';?> example.</h1>
<p>The header above will say 'This is an  example'.</p>
```

'C' style comments end at the first */* encountered. Make sure you don't nest 'C' style comments. It is easy to make this mistake if you are trying to comment out a large block of code.

```
<?php
/*
   echo 'This is a test'; /* This comment will cause a problem */
*/
?>
```

Types

Introduction

PHP supports eight primitive types.

Four scalar types:

- boolean
- integer
- float (floating-point number, aka double)
- string

Two compound types:

- array
- object

And finally two special types:

- resource
- NULL

This manual also introduces some pseudo-types for readability reasons:

- mixed
- number
- callback

And the pseudo-variable `$...`.

Some references to the type "double" may remain in the manual. Consider double the same as float; the two names exist only for historic reasons.

The type of a variable is not usually set by the programmer; rather, it is decided at runtime by PHP depending on the context in which that variable is used.

Note

To check the type and value of an expression, use the var_dump() function.

To get a human-readable representation of a type for debugging, use the gettype() function. To check for a certain type, do *not* use gettype(), but rather the *is_ type* functions. Some examples:

```
<?php
$a_bool = TRUE;   // a boolean
$a_str  = "foo";  // a string
$a_str2 = 'foo';  // a string
$an_int = 12;     // an integer

echo gettype($a_bool); // prints out:  boolean
echo gettype($a_str);  // prints out:  string

// If this is an integer, increment it by four
if (is_int($an_int)) {
   $an_int += 4;
}

// If $bool is a string, print it out
// (does not print out anything)
if (is_string($a_bool)) {
   echo "String: $a_bool";
}
?>
```

To forcibly convert a variable to a certain type, either cast the variable or use the settype() function on it.

Note that a variable may be evaluated with different values in certain situations, depending on what type it is at the time. For more information, see the section on Type Juggling. The type comparison tables may also be useful, as they show examples of various type-related comparisons.

Booleans

This is the simplest type. A boolean expresses a truth value. It can be either **TRUE** or **FALSE**.

Note

The boolean type was introduced in PHP 4.

Syntax

To specify a boolean literal, use the keywords **TRUE** or **FALSE**. Both are case-insensitive.

```
<?php
$foo = True; // assign the value TRUE to $foo
?>
```

Typically, some kind of operator which returns a boolean value, and the value is passed on to a control structure.

```
<?php
// == is an operator which test
// equality and returns a boolean
if ($action == "show_version") {
   echo "The version is 1.23";
}

// this is not necessary...
if ($show_separators == TRUE) {
   echo "<hr>\n";
}

// ...because instead, this can be used:
if ($show_separators) {
   echo "<hr>\n";
}
?>
```

Converting to boolean

To explicitly convert a value to boolean, use the *(bool)* or *(boolean)* casts. However, in most cases the cast is unncecessary, since a value will be automatically converted if an operator, function or control structure requires a boolean argument.

See also Type Juggling.

When converting to boolean, the following values are considered **FALSE**:

- the boolean **FALSE** itself
- the integer 0 (zero)
- the float 0.0 (zero)
- the empty string, and the string "0"
- an array with zero elements
- an object with zero member variables (PHP 4 only)
- the special type NULL (including unset variables)

- SimpleXML objects created from empty tags

Every other value is considered **TRUE** (including any resource).

Warning
-1 is considered **TRUE**, like any other non-zero (whether negative or positive) number!

```
<?php
var_dump((bool) "");        // bool(false)
var_dump((bool) 1);         // bool(true)
var_dump((bool) -2);        // bool(true)
var_dump((bool) "foo");     // bool(true)
var_dump((bool) 2.3e5);     // bool(true)
var_dump((bool) array(12)); // bool(true)
var_dump((bool) array());   // bool(false)
var_dump((bool) "false");   // bool(true)
?>
```

Integers

An integer is a number of the set Z = {..., -2, -1, 0, 1, 2, ...}.

See also:

- Arbitrary length integer / GMP
- Floating point numbers
- Arbitrary precision / BCMath

Syntax

Integer s can be specified in decimal (base 10), hexadecimal (base 16), or octal (base 8) notation, optionally preceded by a sign (- or +).

To use octal notation, precede the number with a *0* (zero). To use hexadecimal notation precede the number with *0x*.

Example #3 - Integer literals

```
<?php
$a = 1234; // decimal number
$a = -123; // a negative number
$a = 0123; // octal number (equivalent to 83 decimal)
$a = 0x1A; // hexadecimal number (equivalent to 26 decimal)
```

```
?>
```

Formally, the structure for integer literals is:

```
decimal     : [1-9][0-9]*
            | 0

hexadecimal : 0[xX][0-9a-fA-F]+

octal       : 0[0-7]+

integer     : [+-]?decimal
            | [+-]?hexadecimal
            | [+-]?octal
```

The size of an integer is platform-dependent, although a maximum value of about two billion is the usual value (that's 32 bits signed). PHP does not support unsigned integer s. Integer size can be determined using the constant **PHP_INT_SIZE**, and maximum value using the constant **PHP_INT_MAX** since PHP 4.4.0 and PHP 5.0.5.

Warning

If an invalid digit is given in an octal integer (i.e. 8 or 9), the rest of the number is ignored.

Example #4 - Octal weirdness

```
<?php
var_dump(01090); // 010 octal = 8 decimal
?>
```

Integer overflow

If PHP encounters a number beyond the bounds of the integer type, it will be interpreted as a float instead. Also, an operation which results in a number beyond the bounds of the integer type will return a float instead.

```
<?php
$large_number =  2147483647;
var_dump($large_number);
// output: int(2147483647)

$large_number =  2147483648;
var_dump($large_number);
// output: float(2147483648)

// it's true also for hexadecimal specified integers between 2^31 and 2^32-1:
```

```
var_dump( 0xffffffff );
// output: float(4294967295)

// this doesn't go for hexadecimal specified integers above 2^32-1:
var_dump( 0x100000000 );
// output: int(2147483647)

$million = 1000000;
$large_number =  50000 * $million;
var_dump($large_number);
// output: float(50000000000)
?>
```

Warning
Unfortunately, there was a bug in PHP which caused this to not always work correctly when negative numbers were involved. For example, the result of *-50000 * $million* is *-429496728*. However, when both operands were positive, there was no problem. This was fixed in PHP 4.1.0.

There is no integer division operator in PHP. *1/2* yields the float *0.5*. The value can be casted to an integer to round it downwards, or the round() function provides finer control over rounding.

```
<?php
var_dump(25/7);         // float(3.5714285714286)
var_dump((int) (25/7)); // int(3)
var_dump(round(25/7));  // float(4)
?>
```

Converting to integer

To explicitly convert a value to integer, use either the *(int)* or *(integer)* casts. However, in most cases the cast is not needed, since a value will be automatically converted if an operator, function or control structure requires an integer argument. A value can also be converted to integer with the intval() function.

See also: type-juggling.

From booleans

FALSE will yield *0* (zero), and **TRUE** will yield *1* (one).

From floating point numbers

When converting from float to integer, the number will be rounded *towards zero*.

If the float is beyond the boundaries of integer (usually *+/- 2.15e+9 = 2^31*), the result is undefined, since the float doesn't have enough precision to give an exact integer result. No warning, not even a notice will be issued when this happens!

Warning

Never cast an unknown fraction to integer, as this can sometimes lead to unexpected results.
 	
`<?php`
`echo (int) ( (0.1+0.7) * 10 ); // echoes 7!`
`?>`
 	
See also the warning about float precision.

From strings

See String conversion to numbers

From other types

Caution
The behaviour of converting to integer is undefined for other types. Do *not* rely on any observed behaviour, as it can change without notice.

Floating point numbers

Floating point numbers (also known as "floats", "doubles", or "real numbers") can be specified using any of the following syntaxes:

```
<?php
$a = 1.234;
$b = 1.2e3;
$c = 7E-10;
?>
```

Formally:

```
LNUM          [0-9]+
DNUM          ([0-9]*[\.]{LNUM}) | ({LNUM}[\.][0-9]*)
EXPONENT_DNUM (({LNUM} | {DNUM}) [eE][+-]? {LNUM})
```

The size of a float is platform-dependent, although a maximum of ~1.8e308 with a precision of roughly 14 decimal digits is a common value (the 64 bit IEEE format).

Warning

Floating point precision

It is typical that simple decimal fractions like *0.1* or *0.7* cannot be converted into their internal binary counterparts without a small loss of precision. This can lead to confusing results: for example, *floor((0.1+0.7)*10)* will usually return *7* instead of the expected *8*, since the internal representation will be something like *7.9*.

This is due to the fact that it is impossible to express some fractions in decimal notation with a finite number of digits. For instance, *1/3* in decimal form becomes *0.3*.

So never trust floating number results to the last digit, and never compare floating point numbers for equality. If higher precision is necessary, the arbitrary precision math functions and gmp functions are available.

Converting to float

For information on converting string s to float, see String conversion to numbers. For values of other types, the conversion is performed by converting the value to integer first and then to float. See Converting to integer for more information. As of PHP 5, a notice is thrown if an object is converted to float.

Strings

A string is series of characters. Before PHP 6, a character is the same as a byte. That is, there are exactly 256 different characters possible. This also implies that PHP has no native support of Unicode. See utf8_encode() and utf8_decode() for some basic Unicode functionality.

Note

It is no problem for a string to become very large. PHP imposes no boundary on the size of a string; the only limit is the available memory of the computer on which PHP is running.

Syntax

A string literal can be specified in four different ways:

- single quoted
- double quoted
- heredoc syntax
- nowdoc syntax (since PHP 5.3.0)

Single quoted

The simplest way to specify a string is to enclose it in single quotes (the character ').

To specify a literal single quote, escape it with a backslash (\ ). To specify a literal backslash before a single quote, or at the end of the string, double it (\\ ). Note that attempting to escape any other character will print the backslash too.

Note
Unlike the two other syntaxes, variables and escape sequences for special characters will *not* be expanded when they occur in single quoted string s.

```
<?php
echo 'this is a simple string';

echo 'You can also have embedded newlines in
strings this way as it is
okay to do';

// Outputs: Arnold once said: "I'll be back"
echo 'Arnold once said: "I\'ll be back"';

// Outputs: You deleted C:\*.*?
echo 'You deleted C:\\*.*?';

// Outputs: You deleted C:\*.*?
echo 'You deleted C:\*.*?';

// Outputs: This will not expand: \n a newline
echo 'This will not expand: \n a newline';

// Outputs: Variables do not $expand $either
echo 'Variables do not $expand $either';
?>
```

Double quoted

If the string is enclosed in double-quotes ("), PHP will interpret more escape sequences for

special characters:

Escaped characters

Sequence	Meaning
\n	linefeed (LF or 0x0A (10) in ASCII)
\r	carriage return (CR or 0x0D (13) in ASCII)
\t	horizontal tab (HT or 0x09 (9) in ASCII)
\v	vertical tab (VT or 0x0B (11) in ASCII) (since PHP 5.2.5)
\f	form feed (FF or 0x0C (12) in ASCII) (since PHP 5.2.5)
**	backslash
\$	dollar sign
\"	double-quote
\[0-7]{1,3}	the sequence of characters matching the regular expression is a character in octal notation
\x[0-9A-Fa-f]{1,2}	the sequence of characters matching the regular expression is a character in hexadecimal notation

As in single quoted string s, escaping any other character will result in the backslash being printed too. Before PHP 5.1.1, the backslash in *\{$var}* was not been printed.

The most important feature of double-quoted string s is the fact that variable names will be expanded. See string parsing for details.

Heredoc

A third way to delimit string s is the heredoc syntax: <<<. After this operator, an identifier is provided, then a newline. The string itself follows, and then the same identifier again to close the quotation.

The closing identifier *must* begin in the first column of the line. Also, the identifier must follow the same naming rules as any other label in PHP: it must contain only alphanumeric characters and underscores, and must start with a non-digit character or underscore.

Warning

It is very important to note that the line with the closing identifier must contain no other characters, except *possibly* a semicolon (*;*). That means especially that the identifier *may not be indented*, and there may not be any spaces or tabs before or after the semicolon. It's also important to realize that the first character before the closing identifier must be a newline as defined by the local operating system. This is *\n* on UNIX systems, including Mac OS X. The closing delimiter (possibly followed by a semicolon) must also be followed by a newline.

If this rule is broken and the closing identifier is not "clean", it will not be considered a closing identifier, and PHP will continue looking for one. If a proper closing identifier is not found before the end of the current file, a parse error will result at the last line.

Heredocs can not be used for initializing class members. Use nowdocs instead.

Example #5 - Invalid example

```
<?php
class foo {
   public $bar = <<<EOT
bar
EOT;
}
?>
```

Heredoc text behaves just like a double-quoted string, without the double quotes. This means that quotes in a heredoc do not need to be escaped, but the escape codes listed above can still be used. Variables are expanded, but the same care must be taken when expressing complex variables inside a heredoc as with string s.

Example #6 - Heredoc string quoting example

```
<?php
$str = <<<EOD
Example of string
spanning multiple lines
using heredoc syntax.
EOD;

/* More complex example, with variables. */
class foo
{
   var $foo;
   var $bar;

   function foo()
   {
       $this->foo = 'Foo';
       $this->bar = array('Bar1', 'Bar2', 'Bar3');
```

```
   }
}

$foo = new foo();
$name = 'MyName';

echo <<<EOT
My name is "$name". I am printing some $foo->foo.
Now, I am printing some {$foo->bar[1]}.
This should print a capital 'A': \x41
EOT;
?>
```

The above example will output:

```
My name is "MyName". I am printing some Foo.
Now, I am printing some Bar2.
This should print a capital 'A': A
```

Note

Heredoc support was added in PHP 4.

Nowdoc

Nowdocs are to single-quoted strings what heredocs are to double-quoted strings. A nowdoc is specified similarly to a heredoc, but *no parsing is done* inside a nowdoc. The construct is ideal for embedding PHP code or other large blocks of text without the need for escaping. It shares some features in common with the SGML *<![CDATA[ ]]>* construct, in that it declares a block of text which is not for parsing.

A nowdoc is identified with the same <<< seqeuence used for heredocs, but the identifier which follows is enclosed in single quotes, e.g. *<<<'EOT'*. All the rules for heredoc identifiers also apply to nowdoc identifiers, especially those regarding the appearance of the closing identifier.

Example #7 - Nowdoc string quoting example

```
<?php
$str = <<<'EOD'
Example of string
spanning multiple lines
using nowdoc syntax.
EOD;

/* More complex example, with variables. */
class foo
{
   public $foo;
   public $bar;
```

```
    function foo()
    {
        $this->foo = 'Foo';
        $this->bar = array('Bar1', 'Bar2', 'Bar3');
    }
}

$foo = new foo();
$name = 'MyName';

echo <<<'EOT'
My name is "$name". I am printing some $foo->foo.
Now, I am printing some {$foo->bar[1]}.
This should not print a capital 'A': \x41
EOT;
?>
```

The above example will output:

```
My name is "$name". I am printing some $foo->foo.
Now, I am printing some {$foo->bar[1]}.
This should not print a capital 'A': \x41
```

Note

Unlike heredocs, nowdocs can be used in any static data context. The typical example is initializing class members or constants:

Example #8 - Static data example

```
<?php
class foo {
    public $bar = <<<'EOT'
bar
EOT;
}
?>
```

Note

Nowdoc support was added in PHP 5.3.0.

Variable parsing

When a string is specified in double quotes or with heredoc, variables are parsed within it.

There are two types of syntax: a simple one and a complex one. The simple syntax is the most common and convenient. It provides a way to embed a variable, an array value, or

an object property in a string with a minimum of effort.

The complex syntax was introduced in PHP 4, and can be recognised by the curly braces surrounding the expression.

Simple syntax

If a dollar sign (*$*) is encountered, the parser will greedily take as many tokens as possible to form a valid variable name. Enclose the variable name in curly braces to explicitly specify the end of the name.

```
<?php
$beer = 'Heineken';
echo "$beer's taste is great"; // works; "'" is an invalid character for
variable names
echo "He drank some $beers";   // won't work; 's' is a valid character for
variable names
echo "He drank some ${beer}s"; // works
echo "He drank some {$beer}s"; // works
?>
```

Similarly, an array index or an object property can be parsed. With array indices, the closing square bracket (*]*) marks the end of the index. The same rules apply to object properties as to simple variables.

```
<?php
// These examples are specific to using arrays inside of strings.
// When outside of a string, always quote array string keys and do not use
// {braces}.

// Show all errors
error_reporting(E_ALL);

$fruits = array('strawberry' => 'red', 'banana' => 'yellow');

// Works, but note that this works differently outside a string
echo "A banana is $fruits[banana].";

// Works
echo "A banana is {$fruits['banana']}.";

// Works, but PHP looks for a constant named banana first, as described below.
echo "A banana is {$fruits[banana]}.";

// Won't work, use braces.  This results in a parse error.
echo "A banana is $fruits['banana'].";

// Works
echo "A banana is " . $fruits['banana'] . ".";

// Works
echo "This square is $square->width meters broad.";
```

```
// Won't work. For a solution, see the complex syntax.
echo "This square is $square->width00 centimeters broad.";
?>
```

For anything more complex, you should use the complex syntax.

Complex (curly) syntax

This isn't called complex because the syntax is complex, but because it allows for the use of complex expressions.

In fact, any value in the namespace can be included in a string with this syntax. Simply write the expression the same way as it would appeared outside the string, and then wrap it in *{* and *}*. Since *{* can not be escaped, this syntax will only be recognised when the *$* immediately follows the *{*. Use *{\$*to get a literal *{$*. Some examples to make it clear:

```
<?php
// Show all errors
error_reporting(E_ALL);

$great = 'fantastic';

// Won't work, outputs: This is { fantastic}
echo "This is { $great}";

// Works, outputs: This is fantastic
echo "This is {$great}";
echo "This is ${great}";

// Works
echo "This square is {$square->width}00 centimeters broad.";

// Works
echo "This works: {$arr[4][3]}";

// This is wrong for the same reason as $foo[bar] is wrong  outside a string.
// In other words, it will still work, but only because PHP first looks for a
// constant named foo; an error of level E_NOTICE (undefined constant) will be
// thrown.
echo "This is wrong: {$arr[foo][3]}";

// Works. When using multi-dimensional arrays, always use braces around arrays
// when inside of strings
echo "This works: {$arr['foo'][3]}";

// Works.
echo "This works: " . $arr['foo'][3];

echo "This works too: {$obj->values[3]->name}";

echo "This is the value of the var named $name: {${$name}}";

echo "This is the value of the var named by the return value of getName():
{${getName()}}";
```

```
echo "This is the value of the var named by the return value of
\$object->getName(): {${$object->getName()}}";
?>
```

Note
Functions and method calls inside *{$}* work since PHP 5.

String access and modification by character

Characters within string s may be accessed and modified by specifying the zero-based offset of the desired character after the string using square array brackets, as in *$str[42]*. Think of a string as an array of characters for this purpose.

Note
String s may also be accessed using braces, as in *$str{42}*, for the same purpose. However, this syntax is deprecated as of PHP 6. Use square brackets instead.

Example #9 - Some string examples

```
<?php
// Get the first character of a string
$str = 'This is a test.';
$first = $str[0];

// Get the third character of a string
$third = $str[2];

// Get the last character of a string.
$str = 'This is still a test.';
$last = $str[strlen($str)-1];

// Modify the last character of a string
$str = 'Look at the sea';
$str[strlen($str)-1] = 'e';

?>
```

Note
Accessing variables of other types using *[]* or *{}* silently returns **NULL**.

Useful functions and operators

String s may be concatenated using the '.' (dot) operator. Note that the '+' (addition) operator will *not* work for this. See String operators for more information.

There are a number of useful functions for string manipulation.

See the string functions section for general functions, and the regular expression functions or the Perl-compatible regular expression functions for advanced find & replace functionality.

There are also functions for URL strings, and functions to encrypt/decrypt strings (mcrypt and mhash).

Finally, see also the character type functions.

Converting to string

A value can be converted to a string using the *(string)* cast or the strval() function. String conversion is automatically done in the scope of an expression where a string is needed. This happens when using the echo() or print() functions, or when a variable is compared to a string. The sections on Types and Type Juggling will make the following clearer. See also the settype() function.

A boolean **TRUE** value is converted to the string *"1"*. Boolean **FALSE** is converted to *""* (the empty string). This allows conversion back and forth between boolean and string values.

An integer or float is converted to a string representing the number textually (including the exponent part for float s). Floating point numbers can be converted using exponential notation (*4.1E+6*).

Note
The decimal point character is defined in the script's locale (category LC_NUMERIC). See the setlocale() function.

Array s are always converted to the string *"Array"*; because of this, echo() and print() can not by themselves show the contents of an array. To view a single element, use a construction such as *echo $arr['foo']*. See below for tips on viewing the entire contents.

Object s in PHP 4 are always converted to the string *"Object"*. To print the values of object members for debugging reasons, read the paragraphs below. To get an object's class name, use the get_class() function. As of PHP 5, the __toString method is used when applicable.

Resource s are always converted to string s with the structure *"Resource id #1"*, where *1* is the unique number assigned to the resource by PHP at runtime. Do not rely upon this structure; it is subject to change. To get a resource 's type, use the get_resource_type() function.

NULL is always converted to an empty string.

As stated above, directly converting an array, object, or resource to a string does not provide any useful information about the value beyond its type. See the functions print_r() and var_dump() for more effective means of inspecting the contents of these types.

Most PHP values can also be converted to string s for permanent storage. This method is called serialization, and is performed by the serialize() function. If the PHP engine was built with WDDX support, PHP values can also be serialized as well-formed XML text.

String conversion to numbers

When a string is evaluated in a numeric context, the resulting value and type are determined as follows.

The string will be evaluated as a float if it contains any of the characters '.', 'e', or 'E'. Otherwise, it will be evaluated as an integer.

The value is given by the initial portion of the string. If the string starts with valid numeric data, this will be the value used. Otherwise, the value will be 0 (zero). Valid numeric data is an optional sign, followed by one or more digits (optionally containing a decimal point), followed by an optional exponent. The exponent is an 'e' or 'E' followed by one or more digits.

```
<?php
$foo = 1 + "10.5";                // $foo is float (11.5)
$foo = 1 + "-1.3e3";              // $foo is float (-1299)
$foo = 1 + "bob-1.3e3";           // $foo is integer (1)
$foo = 1 + "bob3";                // $foo is integer (1)
$foo = 1 + "10 Small Pigs";       // $foo is integer (11)
$foo = 4 + "10.2 Little Piggies"; // $foo is float (14.2)
$foo = "10.0 pigs " + 1;          // $foo is float (11)
$foo = "10.0 pigs " + 1.0;        // $foo is float (11)
?>
```

For more information on this conversion, see the Unix manual page for strtod(3).

To test any of the examples in this section, cut and paste the examples and insert the following line to see what's going on:

```
<?php
echo "\$foo==$foo; type is " . gettype ($foo) . "<br />\n";
?>
```

Do not expect to get the code of one character by converting it to integer, as is done in C. Use the ord() and chr() functions to convert between ASCII codes and characters.

Arrays

An array in PHP is actually an ordered map. A map is a type that associates *values* to *keys*. This type is optimized for several different uses; it can be treated as an array, list (vector), hash table (an implementation of a map), dictionary, collection, stack, queue, and probably more. As array values can be other array s, trees and multidimensional array s are also possible.

Explanation of those data structures is beyond the scope of this manual, but at least one example is provided for each of them. For more information, look towards the considerable literature that exists about this broad topic.

Syntax

Specifying with array()

An array can be created by the array() language construct. It takes as parameters any number of comma-separated *key => value* pairs.

```
array( key => value
     , ...
     )
// key may only be an integer or string
// value may be any value of any type
```

```
<?php
$arr = array("foo" => "bar", 12 => true);

echo $arr["foo"]; // bar
echo $arr[12];    // 1
?>
```

A *key* may be either an integer or a string. If a key is the standard representation of an integer, it will be interpreted as such (i.e. *"8"* will be interpreted as *8*, while *"08"* will be interpreted as *"08"*). Float s in *key* are truncated to integer. The indexed and associative array types are the same type in PHP, which can both contain integer and string indices.

A value can be any PHP type.

```
<?php
$arr = array("somearray" => array(6 => 5, 13 => 9, "a" => 42));

echo $arr["somearray"][6];    // 5
echo $arr["somearray"][13];   // 9
echo $arr["somearray"]["a"];  // 42
?>
```

If a key is not specified for a value, the maximum of the integer indices is taken and the new key will be that value plus 1. If a key that already has an assigned value is specified, that value will be overwritten.

```php
<?php
// This array is the same as ...
array(5 => 43, 32, 56, "b" => 12);

// ...this array
array(5 => 43, 6 => 32, 7 => 56, "b" => 12);
?>
```

Warning
Before PHP 4.3.0, appending to an array in which the current maximum key was negative would create a new key as described above. Since PHP 4.3.0, the new key will be *0*.

Using **TRUE** as *key* will evaluate to integer *1* as a key. Using **FALSE** as *key* will evaluate to integer *0* as a key. Using **NULL** as a key will evaluate to the empty string. Using the empty string as a key will create (or overwrite) a key with the empty string and its value; it is *not* the same as using empty brackets.

Array s and object s can not be used as keys. Doing so will result in a warning: *Illegal offset type*.

Creating/modifying with square bracket syntax

An existing array can be modified by explicitly setting values in it.

This is done by assigning values to the array, specifying the key in brackets. The key can also be omitted, resulting in an empty pair of brackets (*[]*).

```
$arr[key] = value;
$arr[] = value;
// key may be an  integer  or  string
// value may be any value of any type
```

If *$arr* doesn't exist yet, it will be created, so this is also an alternative way to create an array. To change a certain value, assign a new value to that element using its key. To remove a key/value pair, call the unset() function on it.

```php
<?php
$arr = array(5 => 1, 12 => 2);

$arr[] = 56;    // This is the same as $arr[13] = 56;
                // at this point of the script

$arr["x"] = 42; // This adds a new element to
                // the array with key "x"

unset($arr[5]); // This removes the element from the array
```

```
unset($arr);    // This deletes the whole array
?>
```

Note

As mentioned above, if no key is specified, the maximum of the existing integer indices is taken, and the new key will be that maximum value plus 1. If no integer indices exist yet, the key will be *0* (zero). If a key that already has a value is specified, that value will be overwritten.

Note that the maximum integer key used for this *need not currently exist in the* array. It need only have existed in the array at some time since the last time the array was re-indexed. The following example illustrates:

```
<?php
// Create a simple array.
$array = array(1, 2, 3, 4, 5);
print_r($array);

// Now delete every item, but leave the array itself intact:
foreach ($array as $i => $value) {
   unset($array[$i]);
}
print_r($array);

// Append an item (note that the new key is 5, instead of 0).
$array[] = 6;
print_r($array);

// Re-index:
$array = array_values($array);
$array[] = 7;
print_r($array);
?>
```

The above example will output:

```
Array
(
   [0] => 1
   [1] => 2
   [2] => 3
   [3] => 4
   [4] => 5
)
Array
(
)
Array
(
   [5] => 6
)
Array
(
   [0] => 6
```

```
   [1] => 7
)
```

Useful functions

There are quite a few useful functions for working with arrays. See the array functions section.

Note

The unset() function allows removing keys from an array. Be aware that the array will *not* be reindexed. If a true "remove and shift" behavior is desired, the array can be reindexed using the array_values() function.

```
<?php
$a = array(1 => 'one', 2 => 'two', 3 => 'three');
unset($a[2]);
/* will produce an array that would have been defined as
  $a = array(1 => 'one', 3 => 'three');
  and NOT
  $a = array(1 => 'one', 2 =>'three');
*/

$b = array_values($a);
// Now $b is array(0 => 'one', 1 =>'three')
?>
```

The foreach control structure exists specifically for array s. It provides an easy way to traverse an array.

Array do's and don'ts

Why is *$foo[bar]* wrong?

Always use quotes around a string literal array index. For example, *$foo['bar']* is correct, while *$foo[bar]* is not. But why? It is common to encounter this kind of syntax in old scripts:

```
<?php
$foo[bar] = 'enemy';
echo $foo[bar];
// etc
?>
```

This is wrong, but it works. The reason is that this code has an undefined constant (bar)

rather than a string ('bar' - notice the quotes). PHP may in future define constants which, unfortunately for such code, have the same name. It works because PHP automatically converts a *bare string* (an unquoted string which does not correspond to any known symbol) into a string which contains the bare string. For instance, if there is no defined constant named **bar**, then PHP will substitute in the string *'bar'* and use that.

Note

This does not mean to *always* quote the key. Do not quote keys which are constants or variables, as this will prevent PHP from interpreting them.

```
<?php
error_reporting(E_ALL);
ini_set('display_errors', true);
ini_set('html_errors', false);
// Simple array:
$array = array(1, 2);
$count = count($array);
for ($i = 0; $i < $count; $i++) {
   echo "\nChecking $i: \n";
   echo "Bad: " . $array['$i'] . "\n";
   echo "Good: " . $array[$i] . "\n";
   echo "Bad: {$array['$i']}\n";
   echo "Good: {$array[$i]}\n";
}
?>
```

The above example will output:

```
Checking 0:
Notice: Undefined index:  $i in /path/to/script.html on line 9
Bad:
Good: 1
Notice: Undefined index:  $i in /path/to/script.html on line 11
Bad:
Good: 1

Checking 1:
Notice: Undefined index:  $i in /path/to/script.html on line 9
Bad:
Good: 2
Notice: Undefined index:  $i in /path/to/script.html on line 11
Bad:
Good: 2
```

More examples to demonstrate this behaviour:

```
<?php
// Show all errors
error_reporting(E_ALL);

$arr = array('fruit' => 'apple', 'veggie' => 'carrot');
```

```
// Correct
print $arr['fruit'];  // apple
print $arr['veggie']; // carrot

// Incorrect.  This works but also throws a PHP error of level E_NOTICE because
// of an undefined constant named fruit
//
// Notice: Use of undefined constant fruit - assumed 'fruit' in...
print $arr[fruit];    // apple

// This defines a constant to demonstrate what's going on.  The value 'veggie'
// is assigned to a constant named fruit.
define('fruit', 'veggie');

// Notice the difference now
print $arr['fruit'];  // apple
print $arr[fruit];    // carrot

// The following is okay, as it's inside a string. Constants are not looked for
// within strings, so no E_NOTICE occurs here
print "Hello $arr[fruit]";      // Hello apple

// With one exception: braces surrounding arrays within strings allows constants
// to be interpreted
print "Hello {$arr[fruit]}";    // Hello carrot
print "Hello {$arr['fruit']}";  // Hello apple

// This will not work, and will result in a parse error, such as:
// Parse error: parse error, expecting T_STRING' or T_VARIABLE' or T_NUM_STRING'
// This of course applies to using superglobals in strings as well
print "Hello $arr['fruit']";
print "Hello $_GET['foo']";

// Concatenation is another option
print "Hello " . $arr['fruit']; // Hello apple
?>
```

When error_reporting is set to show **E_NOTICE** level errors (by setting it to **E_ALL**, for example), such uses will become immediately visible. By default, error_reporting is set not to show notices.

As stated in the syntax section, what's inside the square brackets ('*[*' and '*]*') must be an expression. This means that code like this works:

```
<?php
echo $arr[somefunc($bar)];
?>
```

This is an example of using a function return value as the array index. PHP also knows about constants:

```
<?php
$error_descriptions[E_ERROR]   = "A fatal error has occured";
```

```
$error_descriptions[E_WARNING] = "PHP issued a warning";
$error_descriptions[E_NOTICE]  = "This is just an informal notice";
?>
```

Note that *E_ERROR* is also a valid identifier, just like *bar* in the first example. But the last example is in fact the same as writing:

```
<?php
$error_descriptions[1] = "A fatal error has occured";
$error_descriptions[2] = "PHP issued a warning";
$error_descriptions[8] = "This is just an informal notice";
?>
```

because *E_ERROR* equals *1*, etc.

So why is it bad then?

At some point in the future, the PHP team might want to add another constant or keyword, or a constant in other code may interfere. For example, it is already wrong to use the words *empty* and *default* this way, since they are reserved keywords.

Note
To reiterate, inside a double-quoted string, it's valid to not surround array indexes with quotes so *"$foo[bar]"* is valid. See the above examples for details on why as well as the section on variable parsing in strings.

Converting to array

For any of the types: integer, float, string, boolean and resource, converting a value to an array results in an array with a single element with index zero and the value of the scalar which was converted. In other words, *(array)$scalarValue* is exactly the same as *array($scalarValue)*.

If an object is converted to an array, the result is an array whose elements are the object 's properties. The keys are the member variable names, with a few notable exceptions: private variables have the class name prepended to the variable name; protected variables have a '*' prepended to the variable name. These prepended values have null bytes on either side. This can result in some unexpected behaviour:

```
<?php

class A {
   private $A; // This will become '\0A\0A'
}
```

```
class B extends A {
   private $A; // This will become '\0B\0A'
   public $AA; // This will become 'AA'
}

var_dump((array) new B());
?>
```

The above will appear to have two keys named 'AA', although one of them is actually named '\0A\0A'.

Converting **NULL** to an array results in an empty array.

Comparing

It is possible to compare arrays with the array_diff() function and with array operators.

Examples

The array type in PHP is very versatile. Here are some examples:

```
<?php
// this
$a = array( 'color' => 'red',
            'taste' => 'sweet',
            'shape' => 'round',
            'name'  => 'apple',
                       4        // key will be 0
          );

// is completely equivalent with
$a['color'] = 'red';
$a['taste'] = 'sweet';
$a['shape'] = 'round';
$a['name']  = 'apple';
$a[]        = 4;        // key will be 0

$b[] = 'a';
$b[] = 'b';
$b[] = 'c';
// will result in the array array(0 => 'a' , 1 => 'b' , 2 => 'c'),
// or simply array('a', 'b', 'c')
?>
```

Example #10 - Using array()

```
<?php
// Array as (property-)map
$map = array( 'version'    => 4,
              'OS'         => 'Linux',
              'lang'       => 'english',
```

```
             'short_tags' => true
           );

// strictly numerical keys
$array = array( 7,
                8,
                0,
                156,
                -10
              );
// this is the same as array(0 => 7, 1 => 8, ...)

$switching = array(         10, // key = 0
                    5    =>  6,
                    3    =>  7,
                    'a'  =>  4,
                             11, // key = 6 (maximum of integer-indices was 5)
                    '8'  =>  2, // key = 8 (integer!)
                    '02' => 77, // key = '02'
                    0    => 12  // the value 10 will be overwritten by 12
                  );

// empty array
$empty = array();
?>
```

Example #11 - Collection

```
<?php
$colors = array('red', 'blue', 'green', 'yellow');

foreach ($colors as $color) {
   echo "Do you like $color?\n";
}

?>
```

The above example will output:

```
Do you like red?
Do you like blue?
Do you like green?
Do you like yellow?
```

Changing the values of the array directly is possible since PHP 5 by passing them by reference. Before that, a workaround is necessary:

Example #12 - Collection

```
<?php
// PHP 5
foreach ($colors as &$color) {
   $color = strtoupper($color);
}
```

```
unset($color); /* ensure that following writes to
$color will not modify the last array element */

// Workaround for older versions
foreach ($colors as $key => $color) {
   $colors[$key] = strtoupper($color);
}

print_r($colors);
?>
```

The above example will output:

```
Array
(
   [0] => RED
   [1] => BLUE
   [2] => GREEN
   [3] => YELLOW
)
```

This example creates a one-based array.

Example #13 - One-based index

```
<?php
$firstquarter  = array(1 => 'January', 'February', 'March');
print_r($firstquarter);
?>
```

The above example will output:

```
Array
(
   [1] => 'January'
   [2] => 'February'
   [3] => 'March'
)
```

Example #14 - Filling an array

```
<?php
// fill an array with all items from a directory
$handle = opendir('.');
while (false !== ($file = readdir($handle))) {
   $files[] = $file;
}
closedir($handle);
?>
```

Array s are ordered. The order can be changed using various sorting functions. See the array functions section for more information. The count() function can be used to count the

number of items in an array.

Example #15 - Sorting an array

```
<?php
sort($files);
print_r($files);
?>
```

Because the value of an array can be anything, it can also be another array. This enables the creation of recursive and multi-dimensional array s.

Example #16 - Recursive and multi-dimensional arrays

```
<?php
$fruits = array ( "fruits"  => array ( "a" => "orange",
                                       "b" => "banana",
                                       "c" => "apple"
                                     ),
                  "numbers" => array ( 1,
                                       2,
                                       3,
                                       4,
                                       5,
                                       6
                                     ),
                  "holes"   => array (      "first",
                                       5 => "second",
                                            "third"
                                     )
                );

// Some examples to address values in the array above
echo $fruits["holes"][5];    // prints "second"
echo $fruits["fruits"]["a"]; // prints "orange"
unset($fruits["holes"][0]);  // remove "first"

// Create a new multi-dimensional array
$juices["apple"]["green"] = "good";
?>
```

Array assignment always involves value copying. It also means that the internal array pointer used by current() and similar functions is reset. Use the reference operator to copy an array by reference.

```
<?php
$arr1 = array(2, 3);
$arr2 = $arr1;
$arr2[] = 4; // $arr2 is changed,
             // $arr1 is still array(2, 3)

$arr3 = &$arr1;
```

```
$arr3[] = 4; // now $arr1 and $arr3 are the same
?>
```

Objects

Object Initialization

To create a new object, use the *new* statement to instantiate a class:

```
<?php
class foo
{
   function do_foo()
   {
       echo "Doing foo.";
   }
}

$bar = new foo;
$bar->do_foo();
?>
```

For a full discussion, see the Classes and Objects chapter.

Converting to object

If an object is converted to an object, it is not modified. If a value of any other type is converted to an object, a new instance of the *stdClass* built-in class is created. If the value was **NULL**, the new instance will be empty. Array s convert to an object with properties named by keys, and corresponding values. For any other value, a member variable named *scalar* will contain the value.

```
<?php
$obj = (object) 'ciao';
echo $obj->scalar;  // outputs 'ciao'
?>
```

Resources

A resource is a special variable, holding a reference to an external resource. Resources are created and used by special functions. See the appendix for a listing of all these functions and the corresponding resource types.

Note
The resource type was introduced in PHP 4

See also the get_resource_type() function.

Converting to resource

As resource variables hold special handlers to opened files, database connections, image canvas areas and the like, converting to a resource makes no sense.

Freeing resources

Thanks to the reference-counting system introduced with PHP 4's Zend Engine, a resource with no more references to it is detected automatically, and it is freed by the garbage collector. For this reason, it is rarely necessary to free the memory manually.

Note
Persistent database links are an exception to this rule. They are *not* destroyed by the garbage collector. See the persistent connections section for more information.

NULL

The special **NULL** value represents a variable with no value. **NULL** is the only possible value of type NULL.

Note
The null type was introduced in PHP 4.

A variable is considered to be null if:

- it has been assigned the constant **NULL**.
- it has not been set to any value yet.
- it has been unset().

Syntax

There is only one value of type null, and that is the case-insensitive keyword **NULL**.

```
<?php
$var = NULL;
?>
```

See also the functions is_null() and unset().

Pseudo-types and variables used in this documentation

mixed

mixed indicates that a parameter may accept multiple (but not necessarily all) types.

gettype() for example will accept all PHP types, while str_replace() will accept string s and array s.

number

number indicates that a parameter can be either integer or float.

callback

Some functions like call_user_func() or usort() accept user-defined callback functions as a parameter. Callback functions can not only be simple functions, but also object methods, including static class methods.

A PHP function is passed by its name as a string. Any built-in or user-defined function can be used, except language constructs such as: array(), echo(), empty(), eval(), exit(), isset(), list(), print() or unset().

A method of an instantiated object is passed as an array containing an object at index 0 and the method name at index 1.

Static class methods can also be passed without instantiating an object of that class by passing the class name instead of an object at index 0.

Apart from common user-defined function, create_function() can also be used to create an anonymous callback function.

Example #17 - Callback function examples

```
<?php

// An example callback function
function my_callback_function() {
```

```
    echo 'hello world!';
}

// An example callback method
class MyClass {
    static function myCallbackMethod() {
        echo 'Hello World!';
    }
}

// Type 1: Simple callback
call_user_func('my_callback_function');

// Type 2: Static class method call
call_user_func(array('MyClass', 'myCallbackMethod'));

// Type 3: Object method call
$obj = new MyClass();
call_user_func(array($obj, 'myCallbackMethod'));

// Type 4: Static class method call (As of PHP 5.2.3)
call_user_func('MyClass::myCallbackMethod');

// Type 5: Relative static class method call (As of PHP 5.3.0)
class A {
    public static function who() {
        echo "A\n";
    }
}

class B extends A {
    public static function who() {
        echo "B\n";
    }
}

call_user_func(array('B', 'parent::who')); // A
?>
```

Note
In PHP4, it was necessary to use a reference to create a callback that points to the actual object, and not a copy of it. For more details, see References Explained.

void

void as a return type means that the return value is useless. *void* in a parameter list means that the function doesn't accept any parameters.

...

`$...` in function prototypes means *and so on*. This variable name is used when a function

can take an endless number of arguments.

Type Juggling

PHP does not require (or support) explicit type definition in variable declaration; a variable's type is determined by the context in which the variable is used. That is to say, if a string value is assigned to variable *$var*, *$var* becomes a string. If an integer value is then assigned to *$var*, it becomes an integer.

An example of PHP's automatic type conversion is the addition operator '+'. If either operand is a float, then both operands are evaluated as float s, and the result will be a float . Otherwise, the operands will be interpreted as integer s, and the result will also be an integer. Note that this does *not* change the types of the operands themselves; the only change is in how the operands are evaluated and what the type of the expression itself is.

```
<?php
$foo = "0";  // $foo is string (ASCII 48)
$foo += 2;   // $foo is now an integer (2)
$foo = $foo + 1.3;  // $foo is now a float (3.3)
$foo = 5 + "10 Little Piggies"; // $foo is integer (15)
$foo = 5 + "10 Small Pigs";     // $foo is integer (15)
?>
```

If the last two examples above seem odd, see String conversion to numbers.

To force a variable to be evaluated as a certain type, see the section on Type casting. To change the type of a variable, see the settype() function.

To test any of the examples in this section, use the var_dump() function.

Note

The behaviour of an automatic conversion to array is currently undefined.

Also, because PHP supports indexing into string s via offsets using the same syntax as array indexing, the following example holds true for all PHP versions:

```
<?php
$a    = 'car'; // $a is a string
$a[0] = 'b';   // $a is still a string
echo $a;       // bar
?>
```

See the section titled String access by character for more information.

Type Casting

Type casting in PHP works much as it does in C: the name of the desired type is written in parentheses before the variable which is to be cast.

```
<?php
$foo = 10;   // $foo is an integer
$bar = (boolean) $foo;   // $bar is a boolean
?>
```

The casts allowed are:

- (int), (integer) - cast to integer
- (bool), (boolean) - cast to boolean
- (float), (double), (real) - cast to float
- (string) - cast to string
- (binary) - cast to binary string (PHP 6)
- (array) - cast to array
- (object) - cast to object

(binary) casting and b prefix forward support was added in PHP 5.2.1

Note that tabs and spaces are allowed inside the parentheses, so the following are functionally equivalent:

```
<?php
$foo = (int) $bar;
$foo = ( int ) $bar;
?>
```

Casting literal string s and variables to binary string s:

```
<?php
$binary = (binary)$string;
$binary = b"binary string";
?>
```

Note

Instead of casting a variable to a string, it is also possible to enclose the variable in double quotes.
 	
`<?php`
`$foo = 10;  // $foo is an integer`

```
$str = "$foo";        // $str is a string
$fst = (string) $foo; // $fst is also a string

// This prints out that "they are the same"
if ($fst === $str) {
   echo "they are the same";
}
?>
```

It may not be obvious exactly what will happen when casting between certain types. For more information, see these sections:

- Converting to boolean
- Converting to integer
- Converting to float
- Converting to string
- Converting to array
- Converting to object
- Converting to resource
- The type comparison tables

Variables

Basics

Variables in PHP are represented by a dollar sign followed by the name of the variable. The variable name is case-sensitive.

Variable names follow the same rules as other labels in PHP. A valid variable name starts with a letter or underscore, followed by any number of letters, numbers, or underscores. As a regular expression, it would be expressed thus:
'[a-zA-Z_\x7f-\xff][a-zA-Z0-9_\x7f-\xff]*'

Note
For our purposes here, a letter is a-z, A-Z, and the ASCII characters from 127 through 255 (0x7f-0xff).

Note
$this is a special variable that can't be assigned.

Tip
See also the Userland Naming Guide.

For information on variable related functions, see the Variable Functions Reference.

```
<?php
$var = 'Bob';
$Var = 'Joe';
echo "$var, $Var";      // outputs "Bob, Joe"

$4site = 'not yet';     // invalid; starts with a number
$_4site = 'not yet';    // valid; starts with an underscore
$täyte = 'mansikka';    // valid; 'ä' is (Extended) ASCII 228.
?>
```

By default, variables are always assigned by value. That is to say, when you assign an expression to a variable, the entire value of the original expression is copied into the destination variable. This means, for instance, that after assigning one variable's value to

another, changing one of those variables will have no effect on the other. For more information on this kind of assignment, see the chapter on Expressions.

PHP also offers another way to assign values to variables: assign by reference. This means that the new variable simply references (in other words, "becomes an alias for" or "points to") the original variable. Changes to the new variable affect the original, and vice versa.

To assign by reference, simply prepend an ampersand (&) to the beginning of the variable which is being assigned (the source variable). For instance, the following code snippet outputs 'My name is Bob' twice:

```
<?php
$foo = 'Bob';              // Assign the value 'Bob' to $foo
$bar = &$foo;              // Reference $foo via $bar.
$bar = "My name is $bar";  // Alter $bar...
echo $bar;
echo $foo;                 // $foo is altered too.
?>
```

One important thing to note is that only named variables may be assigned by reference.

```
<?php
$foo = 25;
$bar = &$foo;      // This is a valid assignment.
$bar = &(24 * 7);  // Invalid; references an unnamed expression.

function test()
{
  return 25;
}

$bar = &test();    // Invalid.
?>
```

It is not necessary to initialize variables in PHP however it is a very good practice. Uninitialized variables have a default value of their type - **FALSE**, zero, empty string or an empty array.

Example #18 - Default values of uninitialized variables

```
<?php
echo ($unset_bool ? "true" : "false"); // false
$unset_int += 25; // 0 + 25 => 25
echo $unset_string . "abc"; // "" . "abc" => "abc"
$unset_array[3] = "def"; // array() + array(3 => "def") => array(3 => "def")
?>
```

Relying on the default value of an uninitialized variable is problematic in the case of including one file into another which uses the same variable name. It is also a major security risk with register_globals turned on. E_NOTICE level error is issued in case of working with uninitialized variables, however not in the case of appending elements to the uninitialized array. isset() language construct can be used to detect if a variable has been already initialized.

Predefined variables

PHP provides a large number of predefined variables to any script which it runs. Many of these variables, however, cannot be fully documented as they are dependent upon which server is running, the version and setup of the server, and other factors. Some of these variables will not be available when PHP is run on the command line. For a listing of these variables, please see the section on Reserved Predefined Variables.

Warning

In PHP 4.2.0 and later, the default value for the PHP directive register_globals is *off*. This is a major change in PHP. Having register_globals *off* affects the set of predefined variables available in the global scope. For example, to get *DOCUMENT_ROOT* you'll use *$_SERVER['DOCUMENT_ROOT']* instead of *$DOCUMENT_ROOT*, or *$_GET['id']* from the URL *http://www.example.com/test.php?id=3* instead of *$id*, or *$_ENV['HOME']* instead of *$HOME*.

For related information on this change, read the configuration entry for register_globals , the security chapter on Using Register Globals, as well as the PHP » 4.1.0 and » 4.2.0 Release Announcements.

Using the available PHP Reserved Predefined Variables, like the superglobal arrays, is preferred.

From version 4.1.0 onward, PHP provides an additional set of predefined arrays containing variables from the web server (if applicable), the environment, and user input. These new arrays are rather special in that they are automatically global--i.e., automatically available in every scope. For this reason, they are often known as "superglobals". (There is no mechanism in PHP for user-defined superglobals.) The superglobals are listed below; however, for a listing of their contents and further discussion on PHP predefined variables and their natures, please see the section Reserved Predefined Variables. Also, you'll notice how the older predefined variables (*$HTTP_*_VARS*) still exist. As of PHP 5.0.0, the long PHP predefined variable arrays may be disabled with the register_long_arrays directive.

Note

Variable variables

Superglobals cannot be used as variable variables inside functions or class methods.

Note
Even though both the superglobal and HTTP_*_VARS can exist at the same time; they are not identical, so modifying one will not change the other.

If certain variables in variables_order are not set, their appropriate PHP predefined arrays are also left empty.

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 #19 - 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 #20 - 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 #21 - Example demonstrating superglobals and scope

```
<?php
```

```
function test_global()
{
   // Most predefined variables aren't "super" and require
   // 'global' to be available to the functions local scope.
   global $HTTP_POST_VARS;

   echo $HTTP_POST_VARS['name'];

   // Superglobals are available in any scope and do
   // not require 'global'. Superglobals are available
   // as of PHP 4.1.0, and HTTP_POST_VARS is now
   // deemed deprecated.
   echo $_POST['name'];
}
?>
```

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 #22 - 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 #23 - Example use of static variables

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

```
?>
```

Now, 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 #24 - Static variables with recursive functions

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

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

Note

Static variables may be declared as seen in the examples above. Trying to assign values to these variables which are the result of expressions will cause a parse error.

Example #25 - Declaring static variables

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

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

References with global and static variables

The Zend Engine 1, driving PHP 4, 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;
   $obj = &new stdclass;
}

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

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

Executing this example will result in the following output:

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

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

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

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

function &get_instance_noref() {
   static $obj;

   echo 'Static object: ';
   var_dump($obj);
   if (!isset($obj)) {
       // Assign the object to the static variable
```

```
        $obj = new stdclass;
    }
    $obj->property++;
    return $obj;
}

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

Executing this example will result in the following output:

```
Static object: NULL
Static object: NULL

Static object: NULL
Static object: object(stdClass)(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.

Variable variables

Sometimes it is convenient to be able to have variable variable names. That is, a variable name which can be set and used dynamically. A normal variable is set with a statement such as:

```
<?php
$a = 'hello';
?>
```

A variable variable takes the value of a variable and treats that as the name of a variable. In the above example, *hello*, can be used as the name of a variable by using two dollar signs. i.e.

```
<?php
$$a = 'world';
?>
```

At this point two variables have been defined and stored in the PHP symbol tree: *$a* with contents "hello" and *$hello* with contents "world". Therefore, this statement:

```php
<?php
echo "$a ${$a}";
?>
```

produces the exact same output as:

```php
<?php
echo "$a $hello";
?>
```

i.e. they both produce: hello world.

In order to use variable variables with arrays, you have to resolve an ambiguity problem. That is, if you write *$$a[1]* then the parser needs to know if you meant to use *$a[1]* as a variable, or if you wanted *$$a* as the variable and then the [1] index from that variable. The syntax for resolving this ambiguity is: *${$a[1]}* for the first case and *${$a}[1]* for the second.

Warning

Please note that variable variables cannot be used with PHP's Superglobal arrays within functions or class methods. The variable *$this* is also a special variable that cannot be referenced dynamically.

Variables From External Sources

HTML Forms (GET and POST)

When a form is submitted to a PHP script, the information from that form is automatically made available to the script. There are many ways to access this information, for example:

Example #26 - A simple HTML form

```php
<form action="foo.php" method="post">
   Name:  <input type="text" name="username" /><br />
   Email: <input type="text" name="email" /><br />
   <input type="submit" name="submit" value="Submit me!" />
</form>
```

Depending on your particular setup and personal preferences, there are many ways to access data from your HTML forms. Some examples are:

Example #27 - Accessing data from a simple POST HTML form

```
<?php
// Available since PHP 4.1.0

  echo $_POST['username'];
  echo $_REQUEST['username'];

  import_request_variables('p', 'p_');
  echo $p_username;

// Unavailable since PHP 6. As of PHP 5.0.0, these long predefined
// variables can be disabled with the register_long_arrays directive.

  echo $HTTP_POST_VARS['username'];

// Available if the PHP directive register_globals = on. As of
// PHP 4.2.0 the default value of register_globals = off.
// Using/relying on this method is not preferred.

  echo $username;
?>
```

Using a GET form is similar except you'll use the appropriate GET predefined variable instead. GET also applies to the QUERY_STRING (the information after the '?' in a URL). So, for example, *http://www.example.com/test.php?id=3* contains GET data which is accessible with *$_GET['id']*. See also $_REQUEST and import_request_variables().

Note

Superglobal arrays, like *$_POST* and *$_GET*, became available in PHP 4.1.0

As shown, before PHP 4.2.0 the default value for register_globals was *on*. The PHP community is encouraging all to not rely on this directive as it's preferred to assume it's *off* and code accordingly.

Note

The magic_quotes_gpc configuration directive affects Get, Post and Cookie values. If turned on, value (It's "PHP!") will automagically become (It\'s \"PHP!\"). Escaping is needed for DB insertion. See also addslashes(), stripslashes() and magic_quotes_sybase.

PHP also understands arrays in the context of form variables (see the related faq). You may, for example, group related variables together, or use this feature to retrieve values from a multiple select input. For example, let's post a form to itself and upon submission display the data:

Example #28 - More complex form variables

```
<?php
if ($_POST) {
   echo '<pre>';
   echo htmlspecialchars(print_r($_POST, true));
   echo '</pre>';
}
?>
<form action="" method="post">
   Name:  <input type="text" name="personal[name]" /><br />
   Email: <input type="text" name="personal[email]" /><br />
   Beer: <br />
   <select multiple name="beer[]">
       <option value="warthog">Warthog</option>
       <option value="guinness">Guinness</option>
       <option value="stuttgarter">Stuttgarter Schwabenbräu</option>
   </select><br />
   <input type="submit" value="submit me!" />
</form>
```

IMAGE SUBMIT variable names

When submitting a form, it is possible to use an image instead of the standard submit button with a tag like:

```
<input type="image" src="image.gif" name="sub" />
```

When the user clicks somewhere on the image, the accompanying form will be transmitted to the server with two additional variables, sub_x and sub_y. These contain the coordinates of the user click within the image. The experienced may note that the actual variable names sent by the browser contains a period rather than an underscore, but PHP converts the period to an underscore automatically.

HTTP Cookies

PHP transparently supports HTTP cookies as defined by » Netscape's Spec. Cookies are a mechanism for storing data in the remote browser and thus tracking or identifying return users. You can set cookies using the setcookie() function. Cookies are part of the HTTP header, so the SetCookie function must be called before any output is sent to the browser. This is the same restriction as for the header() function. Cookie data is then available in the appropriate cookie data arrays, such as *$_COOKIE*, *$HTTP_COOKIE_VARS* as well as in *$_REQUEST*. See the setcookie() manual page for more details and examples.

If you wish to assign multiple values to a single cookie variable, you may assign it as an array. For example:

```
<?php
 setcookie("MyCookie[foo]", 'Testing 1', time()+3600);
 setcookie("MyCookie[bar]", 'Testing 2', time()+3600);
?>
```

That will create two separate cookies although MyCookie will now be a single array in your script. If you want to set just one cookie with multiple values, consider using serialize() or explode() on the value first.

Note that a cookie will replace a previous cookie by the same name in your browser unless the path or domain is different. So, for a shopping cart application you may want to keep a counter and pass this along. i.e.

Example #29 - A setcookie() example

```
<?php
if (isset($_COOKIE['count'])) {
   $count = $_COOKIE['count'] + 1;
} else {
   $count = 1;
}
setcookie('count', $count, time()+3600);
setcookie("Cart[$count]", $item, time()+3600);
?>
```

Dots in incoming variable names

Typically, PHP does not alter the names of variables when they are passed into a script. However, it should be noted that the dot (period, full stop) is not a valid character in a PHP variable name. For the reason, look at it:

```
<?php
$varname.ext;  /* invalid variable name */
?>
```

Now, what the parser sees is a variable named *$varname*, followed by the string concatenation operator, followed by the barestring (i.e. unquoted string which doesn't match any known key or reserved words) 'ext'. Obviously, this doesn't have the intended result.

For this reason, it is important to note that PHP will automatically replace any dots in incoming variable names with underscores.

Determining variable types

Because PHP determines the types of variables and converts them (generally) as needed, it is not always obvious what type a given variable is at any one time. PHP includes several functions which find out what type a variable is, such as: gettype(), is_array(), is_float(), is_int(), is_object(), and is_string(). See also the chapter on Types.

Constants

A constant is an identifier (name) for a simple value. As the name suggests, that value cannot change during the execution of the script (except for magic constants, which aren't actually constants). A constant is case-sensitive by default. By convention, constant identifiers are always uppercase.

The name of a constant follows the same rules as any label in PHP. A valid constant name starts with a letter or underscore, followed by any number of letters, numbers, or underscores. As a regular expression, it would be expressed thusly: *[a-zA-Z_\x7f-\xff][a-zA-Z0-9_\x7f-\xff]**

Tip

See also the Userland Naming Guide.

Example #30 - Valid and invalid constant names

```
<?php

// Valid constant names
define("FOO",     "something");
define("FOO2",    "something else");
define("FOO_BAR", "something more");

// Invalid constant names
define("2FOO",    "something");

// This is valid, but should be avoided:
// PHP may one day provide a magical constant
// that will break your script
define("__FOO__", "something");

?>
```

Note

For our purposes here, a letter is a-z, A-Z, and the ASCII characters from 127 through 255 (0x7f-0xff).

Like superglobals, the scope of a constant is global. You can access constants anywhere in your script without regard to scope. For more information on scope, read the manual section on variable scope.

Syntax

You can define a constant by using the define() -function. Once a constant is defined, it can never be changed or undefined.

Only scalar data (boolean, integer, float and string) can be contained in constants. Do not define resource constants.

You can get the value of a constant by simply specifying its name. Unlike with variables, you should *not* prepend a constant with a *$*. You can also use the function constant() to read a constant's value if you wish to obtain the constant's name dynamically. Use get_defined_constants() to get a list of all defined constants.

Note

Constants and (global) variables are in a different namespace. This implies that for example **TRUE** and *$TRUE* are generally different.

If you use an undefined constant, PHP assumes that you mean the name of the constant itself, just as if you called it as a string (CONSTANT vs "CONSTANT"). An error of level E_NOTICE will be issued when this happens. See also the manual entry on why $foo[bar] is wrong (unless you first define() *bar* as a constant). If you simply want to check if a constant is set, use the defined() function.

These are the differences between constants and variables:

- Constants do not have a dollar sign (*$*) before them;
- Constants may only be defined using the define() function, not by simple assignment;
- Constants may be defined and accessed anywhere without regard to variable scoping rules;
- Constants may not be redefined or undefined once they have been set; and
- Constants may only evaluate to scalar values.

Example #31 - Defining Constants

```
<?php
define("CONSTANT", "Hello world.");
echo CONSTANT; // outputs "Hello world."
echo Constant; // outputs "Constant" and issues a notice.
?>
```

See also Class Constants.

Magic constants

PHP provides a large number of predefined constants to any script which it runs. Many of these constants, however, are created by various extensions, and will only be present when those extensions are available, either via dynamic loading or because they have been compiled in.

There are seven magical constants that change depending on where they are used. For example, the value of __**LINE**__ depends on the line that it's used on in your script. These special constants are case-insensitive and are as follows:

A few "magical" PHP constants

Name	Description
__**LINE**__	The current line number of the file.
__**FILE**__	The full path and filename of the file. If used inside an include, the name of the included file is returned. Since PHP 4.0.2, __**FILE**__ always contains an absolute path with symlinks resolved whereas in older versions it contained relative path under some circumstances.
__**DIR**__	The directory of the file. If used inside an include, the directory of the included file is returned. This is equivalent to *dirname(__FILE__)*. This directory name does not have a trailing slash unless it is the root directory. (Added in PHP 5.3.0.)
__**FUNCTION**__	The function name. (Added in PHP 4.3.0) As of PHP 5 this constant returns the function name as it was declared (case-sensitive). In PHP 4 its value is always lowercased.
__**CLASS**__	The class name. (Added in PHP 4.3.0) As of PHP 5 this constant returns the class name as it was declared (case-sensitive). In PHP 4 its value is always lowercased.
__**METHOD**__	The class method name. (Added in PHP 5.0.0) The method name is returned as it was declared (case-sensitive).
__**NAMESPACE**__	The name of the current namespace (case-sensitive). This constant is defined in compile-time (Added in PHP 5.3.0).

See also get_class(), get_object_vars(), file_exists() and function_exists().

Expressions

Expressions are the most important building stones of PHP. In PHP, almost anything you write is an expression. The simplest yet most accurate way to define an expression is "anything that has a value".

The most basic forms of expressions are constants and variables. When you type "$a = 5", you're assigning '5' into $a. '5', obviously, has the value 5, or in other words '5' is an expression with the value of 5 (in this case, '5' is an integer constant).

After this assignment, you'd expect $a's value to be 5 as well, so if you wrote $b = $a, you'd expect it to behave just as if you wrote $b = 5. In other words, $a is an expression with the value of 5 as well. If everything works right, this is exactly what will happen.

Slightly more complex examples for expressions are functions. For instance, consider the following function:

```
<?php
function foo ()
{
   return 5;
}
?>
```

Assuming you're familiar with the concept of functions (if you're not, take a look at the chapter about functions), you'd assume that typing *$c = foo()* is essentially just like writing *$c = 5*, and you're right. Functions are expressions with the value of their return value. Since foo() returns 5, the value of the expression 'foo()' is 5. Usually functions don't just return a static value but compute something.

Of course, values in PHP don't have to be integers, and very often they aren't. PHP supports four scalar value types: integer values, floating point values (float), string values and boolean values (scalar values are values that you can't 'break' into smaller pieces, unlike arrays, for instance). PHP also supports two composite (non-scalar) types: arrays and objects. Each of these value types can be assigned into variables or returned from functions.

PHP takes expressions much further, in the same way many other languages do. PHP is an expression-oriented language, in the sense that almost everything is an expression. Consider the example we've already dealt with, '$a = 5'. It's easy to see that there are two values involved here, the value of the integer constant '5', and the value of $a which is being updated to 5 as well. But the truth is that there's one additional value involved here, and that's the value of the assignment itself. The assignment itself evaluates to the assigned value, in this case 5. In practice, it means that '$a = 5', regardless of what it does, is an expression with the value 5. Thus, writing something like '$b = ($a = 5)' is like writing '$a = 5; $b = 5;' (a semicolon marks the end of a statement). Since assignments are parsed in a right to left order, you can also write '$b = $a = 5'.

Another good example of expression orientation is pre- and post-increment and

decrement. Users of PHP and many other languages may be familiar with the notation of variable++ and variable--. These are increment and decrement operators. In PHP/FI 2, the statement '$a++' has no value (is not an expression), and thus you can't assign it or use it in any way. PHP enhances the increment/decrement capabilities by making these expressions as well, like in C. In PHP, like in C, there are two types of increment - pre-increment and post-increment. Both pre-increment and post-increment essentially increment the variable, and the effect on the variable is identical. The difference is with the value of the increment expression. Pre-increment, which is written '++$variable', evaluates to the incremented value (PHP increments the variable before reading its value, thus the name 'pre-increment'). Post-increment, which is written '$variable++' evaluates to the original value of $variable, before it was incremented (PHP increments the variable after reading its value, thus the name 'post-increment').

A very common type of expressions are comparison expressions. These expressions evaluate to either **FALSE** or **TRUE**. PHP supports > (bigger than), >= (bigger than or equal to), == (equal), != (not equal), < (smaller than) and <= (smaller than or equal to). The language also supports a set of strict equivalence operators: === (equal to and same type) and !== (not equal to or not same type). These expressions are most commonly used inside conditional execution, such as *if* statements.

The last example of expressions we'll deal with here is combined operator-assignment expressions. You already know that if you want to increment $a by 1, you can simply write '$a++' or '++$a'. But what if you want to add more than one to it, for instance 3? You could write '$a++' multiple times, but this is obviously not a very efficient or comfortable way. A much more common practice is to write '$a = $a + 3'. '$a + 3' evaluates to the value of $a plus 3, and is assigned back into $a, which results in incrementing $a by 3. In PHP, as in several other languages like C, you can write this in a shorter way, which with time would become clearer and quicker to understand as well. Adding 3 to the current value of $a can be written '$a += 3'. This means exactly "take the value of $a, add 3 to it, and assign it back into $a". In addition to being shorter and clearer, this also results in faster execution. The value of '$a += 3', like the value of a regular assignment, is the assigned value. Notice that it is NOT 3, but the combined value of $a plus 3 (this is the value that's assigned into $a). Any two-place operator can be used in this operator-assignment mode, for example '$a -= 5' (subtract 5 from the value of $a), '$b *= 7' (multiply the value of $b by 7), etc.

There is one more expression that may seem odd if you haven't seen it in other languages, the ternary conditional operator:

```
<?php
$first ? $second : $third
?>
```

If the value of the first subexpression is **TRUE** (non-zero), then the second subexpression is evaluated, and that is the result of the conditional expression. Otherwise, the third subexpression is evaluated, and that is the value.

The following example should help you understand pre- and post-increment and expressions in general a bit better:

```php
<?php
function double($i)
{
   return $i*2;
}
$b = $a = 5;        /* assign the value five into the variable $a and $b */
$c = $a++;          /* post-increment, assign original value of $a
                       (5) to $c */
$e = $d = ++$b;     /* pre-increment, assign the incremented value of
                       $b (6) to $d and $e */

/* at this point, both $d and $e are equal to 6 */

$f = double($d++);  /* assign twice the value of $d before
                       the increment, 2*6 = 12 to $f */
$g = double(++$e);  /* assign twice the value of $e after
                       the increment, 2*7 = 14 to $g */
$h = $g += 10;      /* first, $g is incremented by 10 and ends with the
                       value of 24. the value of the assignment (24) is
                       then assigned into $h, and $h ends with the value
                       of 24 as well. */
?>
```

Some expressions can be considered as statements. In this case, a statement has the form of 'expr' ';' that is, an expression followed by a semicolon. In '$b=$a=5;', $a=5 is a valid expression, but it's not a statement by itself. '$b=$a=5;' however is a valid statement.

One last thing worth mentioning is the truth value of expressions. In many events, mainly in conditional execution and loops, you're not interested in the specific value of the expression, but only care about whether it means **TRUE** or **FALSE**. The constants **TRUE** and **FALSE** (case-insensitive) are the two possible boolean values. When necessary, an expression is automatically converted to boolean. See the section about type-casting for details about how.

PHP provides a full and powerful implementation of expressions, and documenting it entirely goes beyond the scope of this manual. The above examples should give you a good idea about what expressions are and how you can construct useful expressions. Throughout the rest of this manual we'll write *expr* to indicate any valid PHP expression.

Operators

An operator is something that you feed with one or more values (or expressions, in programming jargon) which yields another value (so that the construction itself becomes an expression). So you can think of functions or constructions that return a value (like print) as operators and those that return nothing (like echo) as any other thing.

There are three types of operators. Firstly there is the unary operator which operates on only one value, for example ! (the negation operator) or ++ (the increment operator). The second group are termed binary operators; this group contains most of the operators that PHP supports, and a list follows below in the section Operator Precedence.

The third group is the ternary operator: ?:. It should be used to select between two expressions depending on a third one, rather than to select two sentences or paths of execution. Surrounding ternary expressions with parentheses is a very good idea.

Operator Precedence

The precedence of an operator specifies how "tightly" it binds two expressions together. For example, in the expression *1 + 5 * 3*, the answer is *16* and not *18* because the multiplication ("*") operator has a higher precedence than the addition ("+") operator. Parentheses may be used to force precedence, if necessary. For instance: *(1 + 5) * 3* evaluates to *18*. If operator precedence is equal, left to right associativity is used.

The following table lists the precedence of operators with the highest-precedence operators listed at the top of the table. Operators on the same line have equal precedence, in which case their associativity decides which order to evaluate them in.

Operator Precedence

Associativity	Operators	Additional Information
non-associative	new	new
left	[	array()
non-associative	++ --	increment/decrement
non-associative	~ - (int) (float) (string) (array) (object) (bool) @	types
non-associative	instanceof	types
right	!	logical
left	* / %	arithmetic
left	+ - .	arithmetic and string

left	<< >>	bitwise
non-associative	< <= > >= <>	comparison
non-associative	== != === !==	comparison
left	&	bitwise and references
left	^	bitwise
left	|	bitwise
left	&&	logical
left	||	logical
left	? :	ternary
right	= += -= *= /= .= %= &= |= ^= <<= >>=	assignment
left	and	logical
left	xor	logical
left	or	logical
left	,	many uses

Left associativity means that the expression is evaluated from left to right, right associativity means the opposite.

Example #32 - Associativity

```
<?php
$a = 3 * 3 % 5; // (3 * 3) % 5 = 4
$a = true ? 0 : true ? 1 : 2; // (true ? 0 : true) ? 1 : 2 = 2

$a = 1;
$b = 2;
$a = $b += 3; // $a = ($b += 3) -> $a = 5, $b = 5
?>
```

Use parentheses to increase readability of the code.

Note

Although = has a lower precedence than most other operators, PHP will still allow expressions similar to the following: *if (!$a = foo())*, in which case the return value of *foo()* is put into *$a*.

Arithmetic Operators

Remember basic arithmetic from school? These work just like those.

Arithmetic Operators

Example	Name	Result
-$a	Negation	Opposite of $a.
$a + $b	Addition	Sum of $a and $b.
$a - $b	Subtraction	Difference of $a and $b.
$a * $b	Multiplication	Product of $a and $b.
$a / $b	Division	Quotient of $a and $b.
$a % $b	Modulus	Remainder of $a divided by $b.

The division operator ("/") returns a float value unless the two operands are integers (or strings that get converted to integers) and the numbers are evenly divisible, in which case an integer value will be returned.

Operands of modulus are converted to integers (by stripping the decimal part) before processing.

Note
Remainder *$a % $b* is negative for negative *$a*.

See also the manual page on Math functions.

Assignment Operators

The basic assignment operator is "=". Your first inclination might be to think of this as "equal to". Don't. It really means that the left operand gets set to the value of the expression on the rights (that is, "gets set to").

The value of an assignment expression is the value assigned. That is, the value of "$a = 3" is 3. This allows you to do some tricky things:

```
<?php

$a = ($b = 4) + 5; // $a is equal to 9 now, and $b has been set to 4.
```

```
?>
```

In addition to the basic assignment operator, there are "combined operators" for all of the binary arithmetic, array union and string operators that allow you to use a value in an expression and then set its value to the result of that expression. For example:

```
<?php

$a = 3;
$a += 5; // sets $a to 8, as if we had said: $a = $a + 5;
$b = "Hello ";
$b .= "There!"; // sets $b to "Hello There!", just like $b = $b . "There!";

?>
```

Note that the assignment copies the original variable to the new one (assignment by value), so changes to one will not affect the other. This may also have relevance if you need to copy something like a large array inside a tight loop. Assignment by reference is also supported, using the $var = &$othervar; syntax. 'Assignment by reference' means that both variables end up pointing at the same data, and nothing is copied anywhere. To learn more about references, please read References explained. As of PHP 5, objects are assigned by reference unless explicitly told otherwise with the new clone keyword.

Bitwise Operators

Bitwise operators allow you to turn specific bits within an integer on or off. If both the left- and right-hand parameters are strings, the bitwise operator will operate on the characters' ASCII values.

```
<?php
echo 12 ^ 9; // Outputs '5'

echo "12" ^ "9"; // Outputs the Backspace character (ascii 8)
                 // ('1' (ascii 49)) ^ ('9' (ascii 57)) = #8

echo "hallo" ^ "hello"; // Outputs the ascii values #0 #4 #0 #0 #0
                        // 'a' ^ 'e' = #4

echo 2 ^ "3"; // Outputs 1
              // 2 ^ ((int)"3") == 1

echo "2" ^ 3; // Outputs 1
              // ((int)"2") ^ 3 == 1
?>
```

Bitwise Operators

Example	Name	Result
$a & $b	And	Bits that are set in both $a and $b are set.
$a | $b	Or	Bits that are set in either $a or $b are set.
$a ^ $b	Xor	Bits that are set in $a or $b but not both are set.
~ $a	Not	Bits that are set in $a are not set, and vice versa.
$a << $b	Shift left	Shift the bits of $a $b steps to the left (each step means "multiply by two")
$a >> $b	Shift right	Shift the bits of $a $b steps to the right (each step means "divide by two")

Warning
Don't right shift for more than 32 bits on 32 bits systems. Don't left shift in case it results to number longer than 32 bits.

Comparison Operators

Comparison operators, as their name implies, allow you to compare two values. You may also be interested in viewing the type comparison tables, as they show examples of various type related comparisons.

Comparison Operators

Example	Name	Result
$a == $b	Equal	**TRUE** if $a is equal to $b.
$a === $b	Identical	**TRUE** if $a is equal to $b, and they are of the same type. (introduced in PHP 4)
$a != $b	Not equal	**TRUE** if $a is not equal to

		$b.
$a <> $b	Not equal	**TRUE** if $a is not equal to $b.
$a !== $b	Not identical	**TRUE** if $a is not equal to $b, or they are not of the same type. (introduced in PHP 4)
$a < $b	Less than	**TRUE** if $a is strictly less than $b.
$a > $b	Greater than	**TRUE** if $a is strictly greater than $b.
$a <= $b	Less than or equal to	**TRUE** if $a is less than or equal to $b.
$a >= $b	Greater than or equal to	**TRUE** if $a is greater than or equal to $b.

If you compare an integer with a string, the string is converted to a number. If you compare two numerical strings, they are compared as integers. These rules also apply to the switch statement.

```
<?php
var_dump(0 == "a"); // 0 == 0 -> true
var_dump("1" == "01"); // 1 == 1 -> true
var_dump("1" == "1e0"); // 1 == 1 -> true

switch ("a") {
case 0:
   echo "0";
   break;
case "a": // never reached because "a" is already matched with 0
   echo "a";
   break;
}
?>
```

For various types, comparison is done according to the following table (in order).

Comparison with Various Types

Type of Operand 1	**Type of Operand 2**	**Result**
null or string	string	Convert **NULL** to "", numerical or lexical comparison

bool or null	anything	Convert to bool, **FALSE** < **TRUE**
object	object	Built-in classes can define its own comparison, different classes are uncomparable, same class - compare properties the same way as arrays (PHP 4), PHP 5 has its own explanation
string, resource or number	string, resource or number	Translate strings and resources to numbers, usual math
array	array	Array with fewer members is smaller, if key from operand 1 is not found in operand 2 then arrays are uncomparable, otherwise - compare value by value (see following example)
array	anything	array is always greater
object	anything	object is always greater

Example #33 - Transcription of standard array comparison

```
<?php
// Arrays are compared like this with standard comparison operators
function standard_array_compare($op1, $op2)
{
    if (count($op1) < count($op2)) {
        return -1; // $op1 < $op2
    } elseif (count($op1) > count($op2)) {
        return 1; // $op1 > $op2
    }
    foreach ($op1 as $key => $val) {
        if (!array_key_exists($key, $op2)) {
            return null; // uncomparable
        } elseif ($val < $op2[$key]) {
            return -1;
        } elseif ($val > $op2[$key]) {
            return 1;
        }
    }
    return 0; // $op1 == $op2
}
?>
```

See also strcasecmp(), strcmp(), Array operators, and the manual section on Types.

Ternary Operator

Another conditional operator is the "?:" (or ternary) operator.

Example #34 - Assigning a default value

```
<?php
// Example usage for: Ternary Operator
$action = (empty($_POST['action'])) ? 'default' : $_POST['action'];

// The above is identical to this if/else statement
if (empty($_POST['action'])) {
   $action = 'default';
} else {
   $action = $_POST['action'];
}

?>
```

The expression *(expr1) ? (expr2) : (expr3)* evaluates to *expr2* if *expr1* evaluates to **TRUE**, and *expr3* if *expr1* evaluates to **FALSE**.

Note

Please note that the ternary operator is a statement, and that it doesn't evaluate to a variable, but to the result of a statement. This is important to know if you want to return a variable by reference. The statement *return $var == 42 ? $a : $b;* in a return-by-reference function will therefore not work and a warning is issued in later PHP versions.

Note

Is is recommended that you avoid "stacking" ternary expressions. PHP's behaviour when using more than one ternary operator within a single statement is non-obvious:

Example #35 - Non-obvious Ternary Behaviour

```
<?php
// on first glance, the following appears to output 'true'
echo (true?'true':false?'t':'f');

// however, the actual output of the above is 't'
// this is because ternary expressions are evaluated from left to right

// the following is a more obvious version of the same code as above
echo ((true ? 'true' : 'false') ? 't' : 'f');

// here, you can see that the first expression is evaluated to 'true',
which
```

```
// in turn evaluates to (bool)true, thus returning the true branch of the
// second ternary expression.
?>
```

Error Control Operators

PHP supports one error control operator: the at sign (@). When prepended to an expression in PHP, any error messages that might be generated by that expression will be ignored.

If the *track_errors* feature is enabled, any error message generated by the expression will be saved in the variable $php_errormsg. This variable will be overwritten on each error, so check early if you want to use it.

```
<?php
/* Intentional file error */
$my_file = @file ('non_existent_file') or
   die ("Failed opening file: error was '$php_errormsg'");

// this works for any expression, not just functions:
$value = @$cache[$key];
// will not issue a notice if the index $key doesn't exist.

?>
```

Note

The @-operator works only on expressions. A simple rule of thumb is: if you can take the value of something, you can prepend the @ operator to it. For instance, you can prepend it to variables, function and **include()** calls, constants, and so forth. You cannot prepend it to function or class definitions, or conditional structures such as *if* and *foreach*, and so forth.

See also error_reporting() and the manual section for Error Handling and Logging functions.

Warning

Currently the "@" error-control operator prefix will even disable error reporting for critical errors that will terminate script execution. Among other things, this means that if you use "@" to suppress errors from a certain function and either it isn't available or has been mistyped, the script will die right there with no indication as to why.

Execution Operators

PHP supports one execution operator: backticks (``). Note that these are not single-quotes! PHP will attempt to execute the contents of the backticks as a shell command; the output will be returned (i.e., it won't simply be dumped to output; it can be assigned to a variable). Use of the backtick operator is identical to shell_exec().

```
<?php
$output = `ls -al`;
echo "<pre>$output</pre>";
?>
```

Note
The backtick operator is disabled when safe mode is enabled or shell_exec() is disabled.

See also the manual section on Program Execution functions, popen() proc_open(), and Using PHP from the commandline.

Incrementing/Decrementing Operators

PHP supports C-style pre- and post-increment and decrement operators.

Note
The increment/decrement operators do not affect boolean values. Decrementing **NULL** values has no effect too, but incrementing them results in *1*.

Increment/decrement Operators

Example	Name	Effect
++$a	Pre-increment	Increments $a by one, then returns $a.
$a++	Post-increment	Returns $a, then increments $a by one.
--$a	Pre-decrement	Decrements $a by one, then returns $a.
$a--	Post-decrement	Returns $a, then decrements $a by one.

Here's a simple example script:

```
<?php
echo "<h3>Postincrement</h3>";
$a = 5;
echo "Should be 5: " . $a++ . "<br />\n";
echo "Should be 6: " . $a . "<br />\n";

echo "<h3>Preincrement</h3>";
$a = 5;
echo "Should be 6: " . ++$a . "<br />\n";
echo "Should be 6: " . $a . "<br />\n";

echo "<h3>Postdecrement</h3>";
$a = 5;
echo "Should be 5: " . $a-- . "<br />\n";
echo "Should be 4: " . $a . "<br />\n";

echo "<h3>Predecrement</h3>";
$a = 5;
echo "Should be 4: " . --$a . "<br />\n";
echo "Should be 4: " . $a . "<br />\n";
?>
```

PHP follows Perl's convention when dealing with arithmetic operations on character variables and not C's. For example, in Perl 'Z'+1 turns into 'AA', while in C 'Z'+1 turns into '[' (ord('Z') == 90, ord('[') == 91). Note that character variables can be incremented but not decremented and even so only plain ASCII characters (a-z and A-Z) are supported.

Example #36 - Arithmetic Operations on Character Variables

```
<?php
$i = 'W';
for ($n=0; $n<6; $n++) {
   echo ++$i . "\n";
}
?>
```

The above example will output:

```
X
Y
Z
AA
AB
AC
```

Incrementing or decrementing booleans has no effect.

Logical Operators

Logical Operators

Example	Name	Result
$a and $b	And	**TRUE** if both $a and $b are **TRUE**.
$a or $b	Or	**TRUE** if either $a or $b is **TRUE**.
$a xor $b	Xor	**TRUE** if either $a or $b is **TRUE**, but not both.
! $a	Not	**TRUE** if $a is not **TRUE**.
$a && $b	And	**TRUE** if both $a and $b are **TRUE**.
$a || $b	Or	**TRUE** if either $a or $b is **TRUE**.

The reason for the two different variations of "and" and "or" operators is that they operate at different precedences. (See Operator Precedence.)

Example #37 - Logical operators illustrated

```
<?php

// foo() will never get called as those operators are short-circuit
$a = (false && foo());
$b = (true  || foo());
$c = (false and foo());
$d = (true  or  foo());

// "||" has a greater precedence than "or"
$e = false || true; // $e will be assigned to (false || true) which is true
$f = false or true; // $f will be assigned to false
var_dump($e, $f);

// "&&" has a greater precedence than "and"
$g = true && false; // $g will be assigned to (true && false) which is false
$h = true and false; // $h will be assigned to true
var_dump($g, $h);
?>
```

The above example will output something similar to:

```
bool(true)
bool(false)
bool(false)
bool(true)
```

String Operators

There are two string operators. The first is the concatenation operator ('.'), which returns the concatenation of its right and left arguments. The second is the concatenating assignment operator ('.='), which appends the argument on the right side to the argument on the left side. Please read Assignment Operators for more information.

```
<?php
$a = "Hello ";
$b = $a . "World!"; // now $b contains "Hello World!"

$a = "Hello ";
$a .= "World!";     // now $a contains "Hello World!"
?>
```

See also the manual sections on the String type and String functions.

Array Operators

Array Operators

Example	Name	Result
$a + $b	Union	Union of $a and $b.
$a == $b	Equality	**TRUE** if $a and $b have the same key/value pairs.
$a === $b	Identity	**TRUE** if $a and $b have the same key/value pairs in the same order and of the same types.
$a != $b	Inequality	**TRUE** if $a is not equal to $b.
$a <> $b	Inequality	**TRUE** if $a is not equal to $b.
$a !== $b	Non-identity	**TRUE** if $a is not identical to $b.

The + operator appends elements of remaining keys from the right handed array to the left handed, whereas duplicated keys are NOT overwritten.

```php
<?php
$a = array("a" => "apple", "b" => "banana");
$b = array("a" => "pear", "b" => "strawberry", "c" => "cherry");

$c = $a + $b; // Union of $a and $b
echo "Union of \$a and \$b: \n";
var_dump($c);

$c = $b + $a; // Union of $b and $a
echo "Union of \$b and \$a: \n";
var_dump($c);
?>
```

When executed, this script will print the following:

```
Union of $a and $b:
array(3) {
 ["a"]=>
 string(5) "apple"
 ["b"]=>
 string(6) "banana"
 ["c"]=>
 string(6) "cherry"
}
Union of $b and $a:
array(3) {
 ["a"]=>
 string(4) "pear"
 ["b"]=>
 string(10) "strawberry"
 ["c"]=>
 string(6) "cherry"
}
```

Elements of arrays are equal for the comparison if they have the same key and value.

Example #38 - Comparing arrays

```php
<?php
$a = array("apple", "banana");
$b = array(1 => "banana", "0" => "apple");

var_dump($a == $b); // bool(true)
var_dump($a === $b); // bool(false)
?>
```

See also the manual sections on the Array type and Array functions.

Type Operators

instanceof is used to determine whether a PHP variable is an instantiated object of a certain class:

Example #39 - Using instanceof with classes

```
<?php
class MyClass
{
}

class NotMyClass
{
}
$a = new MyClass;

var_dump($a instanceof MyClass);
var_dump($a instanceof NotMyClass);
?>
```

The above example will output:

```
bool(true)
bool(false)
```

instanceof can also be used to determine whether a variable is an instantiated object of a class that inherits from a parent class:

Example #40 - Using instanceof with inherited classes

```
<?php
class ParentClass
{
}

class MyClass extends ParentClass
{
}

$a = new MyClass;

var_dump($a instanceof MyClass);
var_dump($a instanceof ParentClass);
?>
```

The above example will output:

```
bool(true)
bool(true)
```

To check if an object is *not* an instanceof a class, the logical *not* operator can be used.

Example #41 - Using instanceof to check if object is *not* an instanceof a class

```
<?php
class MyClass
```

```
{
}

$a = new MyClass;
var_dump(!($a instanceof stdClass));
?>
```

The above example will output:

```
bool(true)
```

Lastly, *instanceof* can also be used to determine whether a variable is an instantiated object of a class that implements an interface:

Example #42 - Using instanceof for class

```
<?php
interface MyInterface
{
}

class MyClass implements MyInterface
{
}

$a = new MyClass;

var_dump($a instanceof MyClass);
var_dump($a instanceof MyInterface);
?>
```

The above example will output:

```
bool(true)
bool(true)
```

Although *instanceof* is usually used with a literal classname, it can also be used with another object or a string variable:

Example #43 - Using instanceof with other variables

```
<?php
interface MyInterface
{
}

class MyClass implements MyInterface
{
}

$a = new MyClass;
$b = new MyClass;
$c = 'MyClass';
```

```
$d = 'NotMyClass';

var_dump($a instanceof $b); // $b is an object of class MyClass
var_dump($a instanceof $c); // $c is a string 'MyClass'
var_dump($a instanceof $d); // $d is a string 'NotMyClass'
?>
```

The above example will output:

```
bool(true)
bool(true)
bool(false)
```

There are a few pitfalls to be aware of. Before PHP version 5.1.0, *instanceof* would call __autoload() if the class name did not exist. In addition, if the class was not loaded, a fatal error would occur. This can be worked around by using a *dynamic class reference*, or a string variable containing the class name:

Example #44 - Avoiding classname lookups and fatal errors with instanceof in PHP 5.0

```
<?php
$d = 'NotMyClass';
var_dump($a instanceof $d); // no fatal error here
?>
```

The above example will output:

```
bool(false)
```

The *instanceof* operator was introduced in PHP 5. Before this time is_a() was used but is_a() has since been deprecated in favor of *instanceof*.

See also get_class() and is_a().

Control Structures

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

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, *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' 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

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 bigger than b if *$a* is bigger than *$b*, and a is NOT bigger than b otherwise:

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

elseif **/** *else if*

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 *elseif* s 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 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 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;
?>
```

See also while, for, and if for further examples.

while

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

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* 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'.

for

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* 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;
```

foreach

PHP 4 introduced a *foreach* construct, much like Perl and some other languages. This simply gives an easy way to iterate over arrays. *foreach* works only on arrays, 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; the second is a minor but useful extension of the first:

```
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 loop, the value of the current element is assigned to *$value* and the internal array pointer is advanced by one (so on the next loop, you'll be looking at the next element).

The second form does the same thing, except that the current element's key will be assigned to the variable *$key* on each loop.

As of PHP 5, it is possible to iterate objects too.

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() before a *foreach* loop.

Note

Unless the array is referenced, *foreach* operates on a copy of the specified array and not the array itself. *foreach* has some side effects on the array pointer. Don't rely on the array pointer during or after the foreach without resetting it.

As of PHP 5, you can easily modify array's elements by preceding *$value* with *&*. This will assign reference instead of copying the value.

```
<?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
?>
```

This is possible only if iterated array can be referenced (i.e. is variable).

Warning

Reference of a *$value* and the last array element remain even after the *foreach* loop. It is recommended to destroy it by unset().

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 usages:

```
<?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";
}
?>
```

break

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;
   }
}
?>
```

continue

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 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.

```
<?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 return value of the print() call is *int(1)*, and it will look like the optional numeric argument mentioned above.

switch

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 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 comparision.

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 #45 - *switch* **structure**

```php
<?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 #46 - *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.

```
<?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;
?>
```

declare

The *declare* construct is used to set execution directives for a block of code. The syntax of *declare* is similar to the syntax of other flow control constructs:

```
declare (directive)
   statement
```

The *directive* section allows the behavior of the *declare* block to be set. Currently only one directive is recognized: the *ticks* directive. (See below for more information on the ticks directive)

The *statement* part of the *declare* block will be executed -- how it is executed and what side effects occur during execution may depend on the directive set in the *directive* block.

The *declare* construct can also be used in the global scope, affecting all code following it.

```
<?php
// these are the same:

// you can use this:
declare(ticks=1) {
   // entire script here
}

// or you can use this:
declare(ticks=1);
// entire script here
?>
```

Ticks

A tick is an event that occurs for every *N* low-level statements executed by the parser within the *declare* block. The value for *N* is specified using `ticks=`*N* within the *declare* blocks's *directive* section.

The event(s) that occur on each tick are specified using the register_tick_function(). See the example below for more details. Note that more than one event can occur for each tick.

Example #47 - Profile a section of PHP code

```
<?php
// A function that records the time when it is called
function profile($dump = FALSE)
{
   static $profile;

   // Return the times stored in profile, then erase it
   if ($dump) {
       $temp = $profile;
       unset($profile);
       return $temp;
   }

   $profile[] = microtime();
}

// Set up a tick handler
register_tick_function("profile");

// Initialize the function before the declare block
profile();

// Run a block of code, throw a tick every 2nd statement
declare(ticks=2) {
   for ($x = 1; $x < 50; ++$x) {
       echo similar_text(md5($x), md5($x*$x)), "<br />;";
   }
}

// Display the data stored in the profiler
print_r(profile(TRUE));
?>
```

The example profiles the PHP code within the 'declare' block, recording the time at which every second low-level statement in the block was executed. This information can then be used to find the slow areas within particular segments of code. This process can be performed using other methods: using ticks is more convenient and easier to implement.

Ticks are well suited for debugging, implementing simple multitasking, background I/O and many other tasks.

See also register_tick_function() and unregister_tick_function().

return

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()** will also end the execution of an eval() 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()** ed or **require()** ed, then control is passed back to the calling file. Furthermore, if the current script file was **include()** ed, then the value given to

return() will be returned as the value of the **include()** 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 or auto_append_file configuration options in *php.ini*, then that script file's execution is ended.

For more information, see Returning values.

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
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()

The **require()** statement includes and evaluates the specific file.

require() includes and evaluates a specific file. Detailed information on how this inclusion works is described in the documentation for **include()**.

require() and **include()** are identical in every way except how they handle failure. They both produce a Warning, but **require()** results in a Fatal Error. In other words, don't hesitate to use **require()** if you want a missing file to halt processing of the page. **include()** does not behave this way, the script will continue regardless. Be sure to have an appropriate include_path setting as well.

Example #48 - Basic require() examples

```
<?php

require 'prepend.php';

require $somefile;

require ('somefile.txt');

?>
```

See the **include()** documentation for more examples.

Note
Prior to PHP 4.0.2, the following applies: **require()** will always attempt to read the target file, even if the line it's on never executes. The conditional statement won't affect **require()**. However, if the line on which the **require()** occurs is not executed, neither will any of the code in the target file be executed. Similarly, looping structures do not affect the behaviour of **require()**. Although the code contained in the target file is still subject to the loop, the **require()** itself happens only once.

Note
Because this is a language construct and not a function, it cannot be called using variable functions

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 is enabled.

See also **include()**, **require_once()**, **include_once()**, get_included_files(), eval(), file(), readfile(), virtual() and include_path.

include()

The **include()** statement includes and evaluates the specified file.

The documentation below also applies to **require()**. The two constructs are identical in every way except how they handle failure. They both produce a Warning, but **require()** results in a Fatal Error. In other words, use **require()** if you want a missing file to halt processing of the page. **include()** does not behave this way, the script will continue regardless. Be sure to have an appropriate include_path setting as well. Be warned that parse error in included file doesn't cause processing halting in PHP versions prior to PHP 4.3.5. Since this version, it does.

Files for including are first looked for in each include_path entry relative to the current working directory, and then in the directory of current script. E.g. if your include_path is *libraries*, current working directory is */www/*, you included *include/a.php* and there is *include "b.php"* in that file, *b.php* is first looked in */www/libraries/* and then in */www/include/*. If filename begins with./ or../, it is looked only in the current working directory.

When a file is included, the code it contains inherits the variable scope 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 #49 - 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 which are evaluated by the parser before the include occurs.

Example #50 - 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.

If " URL fopen wrappers " are enabled in PHP (which they are in the default configuration), you can specify the file to be included using a URL (via HTTP or other supported wrapper - see List of Supported Protocols/Wrappers 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 is enabled.

Example #51 - 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() 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, fopen() and file() for related information.

Handling Returns: It is possible to execute a **return()** 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 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 (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 #52 - 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 #53 - include() and the return() 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()** 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()** 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()**. It is recommended to use **include_once()** 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 with **include()**. For example:

Example #54 - 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;
       $contents = ob_get_contents();
       ob_end_clean();
       return $contents;
   }
   return false;
}

?>
```

In order to automatically include files within scripts, see also the auto_prepend_file and 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

See also **require()**, **require_once()**, **include_once()**, get_included_files(), readfile(), virtual(), and include_path.

require_once()

The **require_once()** statement includes and evaluates the specified file during the execution of the script. This is a behavior similar to the **require()** statement, with the only difference being that if the code from a file has already been included, it will not be included again. See the documentation for **require()** for more information on how this statement works.

require_once() should be used in cases where the same file might be included and evaluated more than once during a particular execution of a script, and you want to be sure that it is included exactly once to avoid problems with function redefinitions, variable value reassignments, etc.

For examples on using **require_once()** and **include_once()**, look at the » PEAR code included in the latest PHP source code distributions.

Return values are the same as with **include()**. If the file was already included, this function returns **TRUE**

Note
require_once() was added in PHP 4.0.1

Note
Be aware, that the behaviour of **require_once()** and **include_once()** may not be what you expect on a non case sensitive operating system (such as Windows).

Example #55 - require_once() is case insensitive on Windows

```
<?php
require_once "a.php"; // this will include a.php
```

```
require_once "A.php"; // this will include a.php again on Windows! (PHP 4
only)
?>
```

This behaviour changed in PHP 5 - 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 required just once.

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 is enabled.

See also **require()**, **include()**, **include_once()**, get_required_files(), get_included_files(), readfile(), and virtual().

include_once()

The **include_once()** statement includes and evaluates the specified file during the execution of the script. This is a behavior similar to the **include()** 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() should be used in cases where the same file might be included and evaluated more than once during a particular execution of a script, and you want to be sure that it is included exactly once to avoid problems with function redefinitions, variable value reassignments, etc.

For more examples on using **require_once()** and **include_once()**, look at the » PEAR code included in the latest PHP source code distributions.

Return values are the same as with **include()**. If the file was already included, this function returns **TRUE**

Note

include_once() was added in PHP 4.0.1

Note

Be aware, that the behaviour of **include_once()** and **require_once()** may not be what

you expect on a non case sensitive operating system (such as Windows).

Example #56 - include_once() is case insensitive on Windows

```
<?php
include_once "a.php"; // this will include a.php
include_once "A.php"; // this will include a.php again on Windows! (PHP 4
only)
?>
```

This behaviour changed in PHP 5 - 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.

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 is enabled.

See also **include()**, **require()**, **require_once()**, get_required_files(), get_included_files(), readfile(), and virtual().

Functions

User-defined functions

A function may be defined using syntax such as the following:

Example #57 - Pseudo code to demonstrate function uses

```
<?php
function foo($arg_1, $arg_2, /* ..., */ $arg_n)
{
   echo "Example function.\n";
   return $retval;
}
?>
```

Any valid PHP code may appear inside a function, even other functions and class definitions.

Function names follow the same rules as other labels in PHP. A valid function name starts with a letter or underscore, followed by any number of letters, numbers, or underscores. As a regular expression, it would be expressed thus: *[a-zA-Z_\x7f-\xff][a-zA-Z0-9_\x7f-\xff]** .

Tip

See also the Userland Naming Guide.

Functions need not be defined before they are referenced, *except* when a function is conditionally defined as shown in the two examples below.

When a function is defined in a conditional manner such as the two examples shown. Its definition must be processed *prior* to being called.

Example #58 - Conditional functions

```
<?php

$makefoo = true;

/* We can't call foo() from here
  since it doesn't exist yet,
```

```
  but we can call bar() */

bar();

if ($makefoo) {
 function foo()
 {
   echo "I don't exist until program execution reaches me.\n";
 }
}

/* Now we can safely call foo()
  since $makefoo evaluated to true */

if ($makefoo) foo();

function bar()
{
 echo "I exist immediately upon program start.\n";
}

?>
```

Example #59 - Functions within functions

```
<?php
function foo()
{
 function bar()
 {
   echo "I don't exist until foo() is called.\n";
 }
}

/* We can't call bar() yet
  since it doesn't exist. */

foo();

/* Now we can call bar(),
  foo()'s processesing has
  made it accessible. */

bar();

?>
```

All functions and classes in PHP have the global scope - they can be called outside a function even if they were defined inside and vice versa.

PHP does not support function overloading, nor is it possible to undefine or redefine previously-declared functions.

Note

Function names are case-insensitive, though it is usually good form to call functions as they appear in their declaration.

Both variable number of arguments and default arguments are supported in functions. See also the function references for func_num_args(), func_get_arg(), and func_get_args() for more information.

It is possible to call recursive functions in PHP. However avoid recursive function/method calls with over 100-200 recursion levels as it can smash the stack and cause a termination of the current script.

Example #60 - Recursive functions

```
<?php
function recursion($a)
{
   if ($a < 20) {
       echo "$a\n";
       recursion($a + 1);
   }
}
?>
```

Function arguments

Information may be passed to functions via the argument list, which is a comma-delimited list of expressions.

PHP supports passing arguments by value (the default), passing by reference, and default argument values. Variable-length argument lists are also supported, see also the function references for func_num_args(), func_get_arg(), and func_get_args() for more information.

Example #61 - Passing arrays to functions

```
<?php
function takes_array($input)
{
   echo "$input[0] + $input[1] = ", $input[0]+$input[1];
}
?>
```

Making arguments be passed by reference

By default, function arguments are passed by value (so that if the value of the argument within the function is changed, it does not get changed outside of the function). To allow a function to modify its arguments, they must be passed by reference.

To have an argument to a function always passed by reference, prepend an ampersand (&) to the argument name in the function definition:

Example #62 - Passing function parameters by reference

```
<?php
function add_some_extra(&$string)
{
   $string .= 'and something extra.';
}
$str = 'This is a string, ';
add_some_extra($str);
echo $str;    // outputs 'This is a string, and something extra.'
?>
```

Default argument values

A function may define C++-style default values for scalar arguments as follows:

Example #63 - Use of default parameters in functions

```
<?php
function makecoffee($type = "cappuccino")
{
   return "Making a cup of $type.\n";
}
echo makecoffee();
echo makecoffee(null);
echo makecoffee("espresso");
?>
```

The output from the above snippet is:

```
Making a cup of cappuccino.
Making a cup of .
Making a cup of espresso.
```

PHP also allows the use of array s and the special type **NULL** as default values, for

example:

Example #64 - Using non-scalar types as default values

```
<?php
function makecoffee($types = array("cappuccino"), $coffeeMaker = NULL)
{
   $device = is_null($coffeeMaker) ? "hands" : $coffeeMaker;
   return "Making a cup of ".join(", ", $types)." with $device.\n";
}
echo makecoffee();
echo makecoffee(array("cappuccino", "lavazza"), "teapot");
?>
```

The default value must be a constant expression, not (for example) a variable, a class member or a function call.

Note that when using default arguments, any defaults should be on the right side of any non-default arguments; otherwise, things will not work as expected. Consider the following code snippet:

Example #65 - Incorrect usage of default function arguments

```
<?php
function makeyogurt($type = "acidophilus", $flavour)
{
   return "Making a bowl of $type $flavour.\n";
}

echo makeyogurt("raspberry");   // won't work as expected
?>
```

The output of the above example is:

```
Warning: Missing argument 2 in call to makeyogurt() in
/usr/local/etc/httpd/htdocs/phptest/functest.html on line 41
Making a bowl of raspberry .
```

Now, compare the above with this:

Example #66 - Correct usage of default function arguments

```
<?php
function makeyogurt($flavour, $type = "acidophilus")
{
   return "Making a bowl of $type $flavour.\n";
}

echo makeyogurt("raspberry");   // works as expected
?>
```

The output of this example is:

```
Making a bowl of acidophilus raspberry.
```

Note

As of PHP 5, default values may be passed by reference.

Variable-length argument lists

PHP 4 and above has support for variable-length argument lists in user-defined functions. This is really quite easy, using the func_num_args(), func_get_arg(), and func_get_args() functions.

No special syntax is required, and argument lists may still be explicitly provided with function definitions and will behave as normal.

Returning values

Values are returned by using the optional return statement. Any type may be returned, including arrays and objects. This causes the function to end its execution immediately and pass control back to the line from which it was called. See **return()** for more information.

Example #67 - Use of return()

```
<?php
function square($num)
{
   return $num * $num;
```

```
}
echo square(4);   // outputs '16'.
?>
```

A function can not return multiple values, but similar results can be obtained by returning an array.

Example #68 - Returning an array to get multiple values

```
<?php
function small_numbers()
{
   return array (0, 1, 2);
}
list ($zero, $one, $two) = small_numbers();
?>
```

To return a reference from a function, use the reference operator & in both the function declaration and when assigning the returned value to a variable:

Example #69 - Returning a reference from a function

```
<?php
function &returns_reference()
{
   return $someref;
}

$newref =& returns_reference();
?>
```

For more information on references, please check out References Explained.

Variable functions

PHP supports the concept of variable functions. This means that if a variable name has parentheses appended to it, PHP will look for a function with the same name as whatever the variable evaluates to, and will attempt to execute it. Among other things, this can be used to implement callbacks, function tables, and so forth.

Variable functions won't work with language constructs such as echo(), print(), unset(), isset(), empty(), **include()**, **require()** and the like. Utilize wrapper functions to make use of any of these constructs as variable functions.

Example #70 - Variable function example

```
<?php
function foo() {
   echo "In foo()<br />\n";
}

function bar($arg = '')
{
   echo "In bar(); argument was '$arg'.<br />\n";
}

// This is a wrapper function around echo
function echoit($string)
{
   echo $string;
}

$func = 'foo';
$func();        // This calls foo()

$func = 'bar';
$func('test');  // This calls bar()

$func = 'echoit';
$func('test');  // This calls echoit()
?>
```

An object method can also be called with the variable functions syntax.

Example #71 - Variable method example

```
<?php
class Foo
{
   function Variable()
   {
       $name = 'Bar';
       $this->$name(); // This calls the Bar() method
   }

   function Bar()
   {
       echo "This is Bar";
   }
}

$foo = new Foo();
$funcname = "Variable";
$foo->$funcname();  // This calls $foo->Variable()

?>
```

See also call_user_func(), variable variables and function_exists().

Internal (built-in) functions

PHP comes standard with many functions and constructs. There are also functions that require specific PHP extensions compiled in, otherwise fatal "undefined function" errors will appear. For example, to use image functions such as imagecreatetruecolor(), PHP must be compiled with GD support. Or, to use mysql_connect(), PHP must be compiled with MySQL support. There are many core functions that are included in every version of PHP, such as the string and variable functions. A call to phpinfo() or get_loaded_extensions() will show which extensions are loaded into PHP. Also note that many extensions are enabled by default and that the PHP manual is split up by extension. See the configuration, installation, and individual extension chapters, for information on how to set up PHP.

Reading and understanding a function's prototype is explained within the manual section titled how to read a function definition. It's important to realize what a function returns or if a function works directly on a passed in value. For example, str_replace() will return the modified string while usort() works on the actual passed in variable itself. Each manual page also has specific information for each function like information on function parameters, behavior changes, return values for both success and failure, and availability information. Knowing these important (yet often subtle) differences is crucial for writing correct PHP code.

Note
If the parameters given to a function are not what it expects, such as passing an array where a string is expected, the return value of the function is undefined. In this case it will likely return **NULL** but this is just a convention, and cannot be relied upon.

See also function_exists(), the function reference, get_extension_funcs(), and dl().

Classes and Objects (PHP 4)

class

A class is a collection of variables and functions working with these variables. Variables are defined by *var* and functions by *function*. A class is defined using the following syntax:

```
<?php
class Cart {
   var $items;  // Items in our shopping cart

   // Add $num articles of $artnr to the cart

   function add_item($artnr, $num) {
       $this->items[$artnr] += $num;
   }

   // Take $num articles of $artnr out of the cart

   function remove_item($artnr, $num) {
       if ($this->items[$artnr] > $num) {
           $this->items[$artnr] -= $num;
           return true;
       } elseif ($this->items[$artnr] == $num) {
           unset($this->items[$artnr]);
           return true;
       } else {
           return false;
       }
   }
}
?>
```

This defines a class named Cart that consists of an associative array of articles in the cart and two functions to add and remove items from this cart.

Warning

You can *NOT* break up a class definition into multiple files. You also can *NOT* break a class definition into multiple PHP blocks, unless the break is within a method declaration. The following will not work:
 	
`<?php`
`class test {`
`?>`
`<?php`

```
   function test() {
       print 'OK';
   }
}
?>
```

However, the following is allowed:

```
<?php
class test {
   function test() {
       ?>
       <?php
       print 'OK';
   }
}
?>
```

The following cautionary notes are valid for PHP 4.

Caution

The name *stdClass* is used internally by Zend and is reserved. You cannot have a class named *stdClass* in PHP.

Caution

The function names *__sleep* and *__wakeup* are magical in PHP classes. You cannot have functions with these names in any of your classes unless you want the magic functionality associated with them. See below for more information.

Caution

PHP reserves all function names starting with __ as magical. It is recommended that you do not use function names with __ in PHP unless you want some documented magic functionality.

In PHP 4, only constant initializers for *var* variables are allowed. To initialize variables with non-constant values, you need an initialization function which is called automatically when an object is being constructed from the class. Such a function is called a constructor (see below).

```
<?php
class Cart {
   /* None of these will work in PHP 4. */
   var $todays_date = date("Y-m-d");
   var $name = $firstname;
   var $owner = 'Fred ' . 'Jones';
   /* Arrays containing constant values will, though. */
   var $items = array("VCR", "TV");
}

/* This is how it should be done. */
class Cart {
   var $todays_date;
   var $name;
   var $owner;
   var $items = array("VCR", "TV");

   function Cart() {
       $this->todays_date = date("Y-m-d");
       $this->name = $GLOBALS['firstname'];
       /* etc. . . */
   }
}
?>
```

Classes are types, that is, they are blueprints for actual variables. You have to create a variable of the desired type with the *new* operator.

```
<?php
$cart = new Cart;
$cart->add_item("10", 1);

$another_cart = new Cart;
$another_cart->add_item("0815", 3);
?>
```

This creates the objects *$cart* and *$another_cart*, both of the class Cart. The function add_item() of the *$cart* object is being called to add 1 item of article number 10 to the *$cart* . 3 items of article number 0815 are being added to *$another_cart*.

Both, *$cart* and *$another_cart*, have functions add_item(), remove_item() and a variable items. These are distinct functions and variables. You can think of the objects as something similar to directories in a filesystem. In a filesystem you can have two different files *README.TXT*, as long as they are in different directories. Just like with directories where you'll have to type the full pathname in order to reach each file from the toplevel directory, you have to specify the complete name of the function you want to call: in PHP terms, the toplevel directory would be the global namespace, and the pathname separator would be ->. Thus, the names *$cart->items* and *$another_cart->items* name two different variables. Note that the variable is named *$cart->items*, not *$cart->$items*, that is, a variable name in PHP has only a single dollar sign.

```
<?php
// correct, single $
```

```
$cart->items = array("10" => 1);

// invalid, because $cart->$items becomes $cart->""
$cart->$items = array("10" => 1);

// correct, but may or may not be what was intended:
// $cart->$myvar becomes $cart->items
$myvar = 'items';
$cart->$myvar = array("10" => 1);
?>
```

Within a class definition, you do not know under which name the object will be accessible in your program: at the time the Cart class was written, it was unknown whether the object would be named *$cart*, *$another_cart*, or something else later. Thus, you cannot write *$cart->items* within the Cart class itself. Instead, in order to be able to access its own functions and variables from within a class, one can use the pseudo-variable *$this* which can be read as 'my own' or 'current object'. Thus, ' *$this->items[$artnr]* += *$num* ' can be read as 'add *$num* to the *$artnr* counter of my own items array' or 'add *$num* to the *$artnr* counter of the items array within the current object'.

Note

The *$this* pseudo-variable is not usually defined if the method in which it is hosted is called statically. This is not, however, a strict rule: *$this* is defined if a method is called statically from within another object. In this case, the value of *$this* is that of the calling object. This is illustrated in the following example:

```
<?php
class A
{
   function foo()
   {
       if (isset($this)) {
           echo '$this is defined (';
           echo get_class($this);
           echo ")\n";
       } else {
           echo "\$this is not defined.\n";
       }
   }
}

class B
{
   function bar()
   {
       A::foo();
   }
}

$a = new A();
$a->foo();
A::foo();
$b = new B();
$b->bar();
```

```
B::bar();
?>
```

The above example will output:

```
$this is defined (a)
$this is not defined.
$this is defined (b)
$this is not defined.
```

Note

There are some nice functions to handle classes and objects. You might want to take a look at the Class/Object Functions.

extends

Often you need classes with similar variables and functions to another existing class. In fact, it is good practice to define a generic class which can be used in all your projects and adapt this class for the needs of each of your specific projects. To facilitate this, classes can be extensions of other classes. The extended or derived class has all variables and functions of the base class (this is called 'inheritance' despite the fact that nobody died) and what you add in the extended definition. It is not possible to subtract from a class, that is, to undefine any existing functions or variables. An extended class is always dependent on a single base class, that is, multiple inheritance is not supported. Classes are extended using the keyword 'extends'.

```
<?php
class Named_Cart extends Cart {
   var $owner;

   function set_owner ($name) {
       $this->owner = $name;
   }
}
?>
```

This defines a class Named_Cart that has all variables and functions of Cart plus an additional variable *$owner* and an additional function set_owner(). You create a named cart the usual way and can now set and get the carts owner. You can still use normal cart functions on named carts:

```
<?php
$ncart = new Named_Cart;    // Create a named cart
$ncart->set_owner("kris");  // Name that cart
print $ncart->owner;        // print the cart owners name
```

```
$ncart->add_item("10", 1);  // (inherited functionality from cart)
?>
```

This is also called a "parent-child" relationship. You create a class, parent, and use *extends* to create a new class *based* on the parent class: the child class. You can even use this new child class and create another class based on this child class.

Note
Classes must be defined before they are used! If you want the class *Named_Cart* to extend the class *Cart*, you will have to define the class *Cart* first. If you want to create another class called *Yellow_named_cart* based on the class *Named_Cart* you have to define *Named_Cart* first. To make it short: the order in which the classes are defined is important.

Constructors

Constructors are functions in a class that are automatically called when you create a new instance of a class with *new*. A function becomes a constructor, when it has the same name as the class. If a class has no constructor, the constructor of the base class will be called, if it exists.

```
<?php
class Auto_Cart extends Cart {
   function Auto_Cart() {
       $this->add_item("10", 1);
   }
}
?>
```

This defines a class Auto_Cart that is a Cart plus a constructor which initializes the cart with one item of article number "10" each time a new Auto_Cart is being made with "new". Constructors can take arguments and these arguments can be optional, which makes them much more useful. To be able to still use the class without parameters, all parameters to constructors should be made optional by providing default values.

```
<?php
class Constructor_Cart extends Cart {
   function Constructor_Cart($item = "10", $num = 1) {
       $this->add_item ($item, $num);
   }
}

// Shop the same old boring stuff.
$default_cart = new Constructor_Cart;

// Shop for real...
$different_cart = new Constructor_Cart("20", 17);
```

```
?>
```

You also can use the @ operator to *mute* errors occurring in the constructor, e.g. *@new*.

```
<?php
class A
{
   function A()
   {
       echo "I am the constructor of A.<br />\n";
   }

   function B()
   {
       echo "I am a regular function named B in class A.<br />\n";
       echo "I am not a constructor in A.<br />\n";
   }
}

class B extends A
{
}

// This will call B() as a constructor
$b = new B;
?>
```

The function B() in class A will suddenly become a constructor in class B, although it was never intended to be. PHP 4 does not care if the function is being defined in class B, or if it has been inherited.

Caution
PHP 4 doesn't call constructors of the base class automatically from a constructor of a derived class. It is your responsibility to propagate the call to constructors upstream where appropriate.

Destructors are functions that are called automatically when an object is destroyed, either with unset() or by simply going out of scope. There are no destructors in PHP. You may use register_shutdown_function() instead to simulate most effects of destructors.

Scope Resolution Operator (*::*)

Caution
The following is valid for PHP 4 and later only.

Sometimes it is useful to refer to functions and variables in base classes or to refer to functions in classes that have not yet any instances. The :: operator is being used for this.

```
<?php
class A {
   function example() {
       echo "I am the original function A::example().<br />\n";
   }
}

class B extends A {
   function example() {
       echo "I am the redefined function B::example().<br />\n";
       A::example();
   }
}

// there is no object of class A.
// this will print
//   I am the original function A::example().<br />
A::example();

// create an object of class B.
$b = new B;

// this will print
//   I am the redefined function B::example().<br />
//   I am the original function A::example().<br />
$b->example();
?>
```

The above example calls the function example() in class A, but there is no object of class A, so that we cannot write $a->example() or similar. Instead we call example() as a 'class function', that is, as a function of the class itself, not any object of that class.

There are class functions, but there are no class variables. In fact, there is no object at all at the time of the call. Thus, a class function may not use any object variables (but it can use local and global variables), and it may not use *$this* at all.

In the above example, class B redefines the function example(). The original definition in class A is shadowed and no longer available, unless you are referring specifically to the implementation of example() in class A using the ::-operator. Write A::example() to do this (in fact, you should be writing parent::example(), as shown in the next section).

In this context, there is a current object and it may have object variables. Thus, when used from WITHIN an object function, you may use *$this* and object variables.

parent

You may find yourself writing code that refers to variables and functions in base classes. This is particularly true if your derived class is a refinement or specialisation of code in your base class.

Instead of using the literal name of the base class in your code, you should be using the special name *parent*, which refers to the name of your base class as given in the *extends* declaration of your class. By doing this, you avoid using the name of your base class in more than one place. Should your inheritance tree change during implementation, the change is easily made by simply changing the *extends* declaration of your class.

```
<?php
class A {
   function example() {
       echo "I am A::example() and provide basic functionality.<br />\n";
   }
}

class B extends A {
   function example() {
       echo "I am B::example() and provide additional functionality.<br />\n";
       parent::example();
   }
}

$b = new B;

// This will call B::example(), which will in turn call A::example().
$b->example();
?>
```

Serializing objects - objects in sessions

serialize() returns a string containing a byte-stream representation of any value that can be stored in PHP. unserialize() can use this string to recreate the original variable values. Using serialize to save an object will save all variables in an object. The functions in an object will not be saved, only the name of the class.

In order to be able to unserialize() an object, the class of that object needs to be defined. That is, if you have an object *$a* of class A on page1.php and serialize this, you'll get a string that refers to class A and contains all values of variabled contained in *$a*. If you want to be able to unserialize this on page2.php, recreating *$a* of class A, the definition of class A must be present in page2.php. This can be done for example by storing the class definition of class A in an include file and including this file in both page1.php and page2.php.

```
<?php
// classa.inc:

 class A {
     var $one = 1;

     function show_one() {
         echo $this->one;
     }
 }
```

```
// page1.php:

 include("classa.inc");

 $a = new A;
 $s = serialize($a);
 // store $s somewhere where page2.php can find it.
 $fp = fopen("store", "w");
 fwrite($fp, $s);
 fclose($fp);

// page2.php:

 // this is needed for the unserialize to work properly.
 include("classa.inc");

 $s = implode("", @file("store"));
 $a = unserialize($s);

 // now use the function show_one() of the $a object.
 $a->show_one();
?>
```

If you are using sessions and use session_register() to register objects, these objects are serialized automatically at the end of each PHP page, and are unserialized automatically on each of the following pages. This basically means that these objects can show up on any of your pages once they become part of your session.

It is strongly recommended that you include the class definitions of all such registered objects on all of your pages, even if you do not actually use these classes on all of your pages. If you don't and an object is being unserialized without its class definition being present, it will lose its class association and become an object of class *stdClass* without any functions available at all, that is, it will become quite useless.

So if in the example above *$a* became part of a session by running *session_register("a")*, you should include the file *classa.inc* on all of your pages, not only page1.php and page2.php.

The magic functions *__sleep* and *__wakeup*

serialize() checks if your class has a function with the magic name *__sleep*. If so, that function is being run prior to any serialization. It can clean up the object and is supposed to return an array with the names of all variables of that object that should be serialized. If the method doesn't return anything then **NULL** is serialized and E_NOTICE is issued.

The intended use of *__sleep* is to commit pending data or perform similar cleanup tasks. Also, the function is useful if you have very large objects which need not be saved completely.

Conversely, unserialize() checks for the presence of a function with the magic name *__wakeup*. If present, this function can reconstruct any resources that object may have.

The intended use of *__wakeup* is to reestablish any database connections that may have

been lost during serialization and perform other reinitialization tasks.

References inside the constructor

Creating references within the constructor can lead to confusing results. This tutorial-like section helps you to avoid problems.

```
<?php
class Foo {
   function Foo($name) {
       // create a reference inside the global array $globalref
       global $globalref;
       $globalref[] = &$this;
       // set name to passed value
       $this->setName($name);
       // and put it out
       $this->echoName();
   }

   function echoName() {
       echo "<br />", $this->name;
   }

   function setName($name) {
       $this->name = $name;
   }
}
?>
```

Let us check out if there is a difference between *$bar1* which has been created using the copy = operator and *$bar2* which has been created using the reference =&operator...

```
<?php
$bar1 = new Foo('set in constructor');
$bar1->echoName();
$globalref[0]->echoName();

/* output:
set in constructor
set in constructor
set in constructor */

$bar2 =& new Foo('set in constructor');
$bar2->echoName();
$globalref[1]->echoName();

/* output:
set in constructor
set in constructor
set in constructor */
?>
```

Apparently there is no difference, but in fact there is a very significant one: *$bar1* and *$globalref[0]* are _NOT_ referenced, they are NOT the same variable. This is because "new" does not return a reference by default, instead it returns a copy.

Note
There is no performance loss (since PHP 4 and up use reference counting) returning copies instead of references. On the contrary it is most often better to simply work with copies instead of references, because creating references takes some time where creating copies virtually takes no time (unless none of them is a large array or object and one of them gets changed and the other(s) one(s) subsequently, then it would be wise to use references to change them all concurrently).

To prove what is written above let us watch the code below.

```
<?php
// now we will change the name. what do you expect?
// you could expect that both $bar1 and $globalref[0] change their names...
$bar1->setName('set from outside');

// as mentioned before this is not the case.
$bar1->echoName();
$globalref[0]->echoName();

/* output:
set from outside
set in constructor */

// let us see what is different with $bar2 and $globalref[1]
$bar2->setName('set from outside');

// luckily they are not only equal, they are the same variable
// thus $bar2->name and $globalref[1]->name are the same too
$bar2->echoName();
$globalref[1]->echoName();

/* output:
set from outside
set from outside */
?>
```

Another final example, try to understand it.

```
<?php
class A {
   function A($i) {
       $this->value = $i;
       // try to figure out why we do not need a reference here
       $this->b = new B($this);
   }

   function createRef() {
       $this->c = new B($this);
   }
```

```
   function echoValue() {
       echo "<br />","class ",get_class($this),': ',$this->value;
   }
}


class B {
   function B(&$a) {
       $this->a = &$a;
   }

   function echoValue() {
       echo "<br />","class ",get_class($this),': ',$this->a->value;
   }
}

// try to understand why using a simple copy here would yield
// in an undesired result in the *-marked line
$a =& new A(10);
$a->createRef();

$a->echoValue();
$a->b->echoValue();
$a->c->echoValue();

$a->value = 11;

$a->echoValue();
$a->b->echoValue(); // *
$a->c->echoValue();

?>
```

The above example will output:

```
class A: 10
class B: 10
class B: 10
class A: 11
class B: 11
class B: 11
```

Comparing objects

In PHP 4, objects are compared in a very simple manner, namely: Two object instances are equal if they have the same attributes and values, and are instances of the same class. Similar rules are applied when comparing two objects using the identity operator (===).

If we were to execute the code in the example below:

Example #72 - Example of object comparison in PHP 4

```
<?php
```

```
function bool2str($bool) {
   if ($bool === false) {
           return 'FALSE';
   } else {
           return 'TRUE';
   }
}

function compareObjects(&$o1, &$o2) {
   echo 'o1 == o2 : '.bool2str($o1 == $o2)."\n";
   echo 'o1 != o2 : '.bool2str($o1 != $o2)."\n";
   echo 'o1 === o2 : '.bool2str($o1 === $o2)."\n";
   echo 'o1 !== o2 : '.bool2str($o1 !== $o2)."\n";
}

class Flag {
   var $flag;

   function Flag($flag=true) {
           $this->flag = $flag;
   }
}

class SwitchableFlag extends Flag {

   function turnOn() {
       $this->flag = true;
   }

   function turnOff() {
       $this->flag = false;
   }
}

$o = new Flag();
$p = new Flag(false);
$q = new Flag();

$r = new SwitchableFlag();

echo "Compare instances created with the same parameters\n";
compareObjects($o, $q);

echo "\nCompare instances created with different parameters\n";
compareObjects($o, $p);

echo "\nCompare an instance of a parent class with one from a subclass\n";
compareObjects($o, $r);
?>
```

The above example will output:

```
Compare instances created with the same parameters
o1 == o2 : TRUE
o1 != o2 : FALSE
o1 === o2 : TRUE
o1 !== o2 : FALSE

Compare instances created with different parameters
o1 == o2 : FALSE
```

```
o1 != o2 : TRUE
o1 === o2 : FALSE
o1 !== o2 : TRUE

Compare an instance of a parent class with one from a subclass
o1 == o2 : FALSE
o1 != o2 : TRUE
o1 === o2 : FALSE
o1 !== o2 : TRUE
```

Which is the output we will expect to obtain given the comparison rules above. Only instances with the same values for their attributes and from the same class are considered equal and identical.

Even in the cases where we have object composition, the same comparison rules apply. In the example below we create a container class that stores an associative array of Flag objects.

Example #73 - Compound object comparisons in PHP 4

```
<?php
class FlagSet {
   var $set;

   function FlagSet($flagArr = array()) {
       $this->set = $flagArr;
   }

   function addFlag($name, $flag) {
       $this->set[$name] = $flag;
   }

   function removeFlag($name) {
       if (array_key_exists($name, $this->set)) {
           unset($this->set[$name]);
       }
   }
}


$u = new FlagSet();
$u->addFlag('flag1', $o);
$u->addFlag('flag2', $p);
$v = new FlagSet(array('flag1'=>$q, 'flag2'=>$p));
$w = new FlagSet(array('flag1'=>$q));

echo "\nComposite objects u(o,p) and v(q,p)\n";
compareObjects($u, $v);

echo "\nu(o,p) and w(q)\n";
compareObjects($u, $w);
?>
```

The above example will output:

```
Composite objects u(o,p) and v(q,p)
o1 == o2 : TRUE
o1 != o2 : FALSE
```

```
o1 === o2 : TRUE
o1 !== o2 : FALSE

u(o,p) and w(q)
o1 == o2 : FALSE
o1 != o2 : TRUE
o1 === o2 : FALSE
o1 !== o2 : TRUE
```

Classes and Objects (PHP 5)

Introduction

In PHP 5 there is a new Object Model. PHP's handling of objects has been completely rewritten, allowing for better performance and more features.

Tip
See also the Userland Naming Guide.

The Basics

class

Every class definition begins with the keyword class, followed by a class name, which can be any name that isn't a reserved word in PHP. Followed by a pair of curly braces, which contains the definition of the classes members and methods. A pseudo-variable, *$this* is available when a method is called from within an object context. *$this* is a reference to the calling object (usually the object to which the method belongs, but can be another object, if the method is called statically from the context of a secondary object). This is illustrated in the following examples:

Example #74 - *$this* **variable in object-oriented language**

```
<?php
class A
{
   function foo()
   {
       if (isset($this)) {
           echo '$this is defined (';
           echo get_class($this);
           echo ")\n";
       } else {
           echo "\$this is not defined.\n";
       }
   }
}

class B
{
   function bar()
   {
       A::foo();
   }
}
```

```
$a = new A();
$a->foo();
A::foo();
$b = new B();
$b->bar();
B::bar();
?>
```

The above example will output:

```
$this is defined (a)
$this is not defined.
$this is defined (b)
$this is not defined.
```

Example #75 - Simple Class definition

```
<?php
class SimpleClass
{
   // member declaration
   public $var = 'a default value';

   // method declaration
   public function displayVar() {
       echo $this->var;
   }
}
?>
```

The default value must be a constant expression, not (for example) a variable, a class member or a function call.

Example #76 - Class members' default value

```
<?php
class SimpleClass
{
   // invalid member declarations:
   public $var1 = 'hello '.'world';
   public $var2 = <<<EOD
hello world
EOD;
   public $var3 = 1+2;
   public $var4 = self::myStaticMethod();
   public $var5 = $myVar;

   // valid member declarations:
   public $var6 = myConstant;
   public $var7 = self::classConstant;
   public $var8 = array(true, false);
```

```
}
?>
```

Note

There are some nice functions to handle classes and objects. You might want to take a look at the Class/Object Functions.

Unlike heredocs, nowdocs can be used in any static data context.

Example #77 - Static data example

```
<?php
class foo {
   // As of PHP 5.3.0
   public $bar = <<<'EOT'
bar
EOT;
}
?>
```

Note

Nowdoc support was added in PHP 5.3.0.

new

To create an instance of a class, a new object must be created and assigned to a variable. An object will always be assigned when creating a new object unless the object has a constructor defined that throws an exception on error. Classes should be defined before instantiation (and in some cases this is a requirement).

Example #78 - Creating an instance

```
<?php
$instance = new SimpleClass();
?>
```

In the class context, it is possible to create a new object by *new self* and *new parent*.

When assigning an already created instance of a class to a new variable, the new variable will access the same instance as the object that was assigned. This behaviour is the same when passing instances to a function. A copy of an already created object can be made by

cloning it.

Example #79 - Object Assignment

```
<?php
$assigned   =  $instance;
$reference  =& $instance;

$instance->var = '$assigned will have this value';

$instance = null; // $instance and $reference become null

var_dump($instance);
var_dump($reference);
var_dump($assigned);
?>
```

The above example will output:

```
NULL
NULL
object(SimpleClass)#1 (1) {
  ["var"]=>
    string(30) "$assigned will have this value"
}
```

extends

A class can inherit methods and members of another class by using the extends keyword in the declaration. It is not possible to extend multiple classes, a class can only inherit one base class.

The inherited methods and members can be overridden, unless the parent class has defined a method as final, by redeclaring them with the same name defined in the parent class. It is possible to access the overridden methods or static members by referencing them with parent::

Example #80 - Simple Class Inheritance

```
<?php
class ExtendClass extends SimpleClass
{
   // Redefine the parent method
   function displayVar()
   {
       echo "Extending class\n";
       parent::displayVar();
   }
}

$extended = new ExtendClass();
$extended->displayVar();
?>
```

The above example will output:

```
Extending class
a default value
```

Autoloading Objects

Many developers writing object-oriented applications create one PHP source file per-class definition. One of the biggest annoyances is having to write a long list of needed includes at the beginning of each script (one for each class).

In PHP 5, this is no longer necessary. You may define an __autoload function which is automatically called in case you are trying to use a class/interface which hasn't been defined yet. By calling this function the scripting engine is given a last chance to load the class before PHP fails with an error.

Note

Exceptions thrown in __autoload function cannot be caught in the catch block and results in a fatal error.

Note

Autoloading is not available if using PHP in CLI interactive mode.

Note

If the class name is used e.g. in call_user_func() then it can contain some dangerous characters such as *../*. It is recommended to not use the user-input in such functions or at least verify the input in **__autoload()**.

Example #81 - Autoload example

This example attempts to load the classes *MyClass1* and *MyClass2* from the files *MyClass1.php* and *MyClass2.php* respectively.

```
<?php
function __autoload($class_name) {
   require_once $class_name . '.php';
}

$obj  = new MyClass1();
$obj2 = new MyClass2();
```

```
?>
```

Example #82 - Autoload other example

This example attempts to load the interface *ITest*.

```
<?php

function __autoload($name) {
   var_dump($name);
}

class Foo implements ITest {
}

/*
string(5) "ITest"

Fatal error: Interface 'ITest' not found in ...
*/
?>
```

Constructors and Destructors

Constructor

void **__construct** ([mixed `$args` [, `$...`]])

PHP 5 allows developers to declare constructor methods for classes. Classes which have a constructor method call this method on each newly-created object, so it is suitable for any initialization that the object may need before it is used.

Note

Parent constructors are not called implicitly if the child class defines a constructor. In order to run a parent constructor, a call to **parent::__construct()** within the child constructor is required.

Example #83 - using new unified constructors

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

```
class SubClass extends BaseClass {
  function __construct() {
      parent::__construct();
      print "In SubClass constructor\n";
  }
}

$obj = new BaseClass();
$obj = new SubClass();
?>
```

For backwards compatibility, if PHP 5 cannot find a **__construct()** function for a given class, it will search for the old-style constructor function, by the name of the class. Effectively, it means that the only case that would have compatibility issues is if the class had a method named **__construct()** which was used for different semantics.

Destructor

void **__destruct** (void)

PHP 5 introduces a destructor concept similar to that of other object-oriented languages, such as C++. The destructor method will be called as soon as all references to a particular object are removed or when the object is explicitly destroyed or in any order in shutdown sequence.

Example #84 - Destructor Example

```
<?php
class MyDestructableClass {
  function __construct() {
      print "In constructor\n";
      $this->name = "MyDestructableClass";
  }

  function __destruct() {
      print "Destroying " . $this->name . "\n";
  }
}

$obj = new MyDestructableClass();
?>
```

Like constructors, parent destructors will not be called implicitly by the engine. In order to run a parent destructor, one would have to explicitly call **parent::__destruct()** in the destructor body.

Note

Destructors called during the script shutdown have HTTP headers already sent. The working directory in the script shutdown phase can be different with some SAPIs (e.g.

Apache).

Note

Attempting to throw an exception from a destructor (called in the time of script termination) causes a fatal error.

Visibility

The visibility of a property or method can be defined by prefixing the declaration with the keywords: public, protected or private. Public declared items can be accessed everywhere. Protected limits access to inherited and parent classes (and to the class that defines the item). Private limits visibility only to the class that defines the item.

Members Visibility

Class members must be defined with public, private, or protected.

Example #85 - Member declaration

```
<?php
/**
* Define MyClass
*/
class MyClass
{
   public $public = 'Public';
   protected $protected = 'Protected';
   private $private = 'Private';

   function printHello()
   {
       echo $this->public;
       echo $this->protected;
       echo $this->private;
   }
}

$obj = new MyClass();
echo $obj->public; // Works
echo $obj->protected; // Fatal Error
echo $obj->private; // Fatal Error
$obj->printHello(); // Shows Public, Protected and Private


/**
* Define MyClass2
*/
class MyClass2 extends MyClass
```

```
{
   // We can redeclare the public and protected method, but not private
   protected $protected = 'Protected2';

   function printHello()
   {
       echo $this->public;
       echo $this->protected;
       echo $this->private;
   }
}

$obj2 = new MyClass2();
echo $obj2->public; // Works
echo $obj2->private; // Undefined
echo $obj2->protected; // Fatal Error
$obj2->printHello(); // Shows Public, Protected2, Undefined

?>
```

Note

The PHP 4 method of declaring a variable with the *var* keyword is still supported for compatibility reasons (as a synonym for the public keyword). In PHP 5 before 5.1.3, its usage would generate an **E_STRICT** warning.

Method Visibility

Class methods must be defined with public, private, or protected. Methods without any declaration are defined as public.

Example #86 - Method Declaration

```
<?php
/**
* Define MyClass
*/
class MyClass
{
   // Declare a public constructor
   public function __construct() { }

   // Declare a public method
   public function MyPublic() { }

   // Declare a protected method
   protected function MyProtected() { }

   // Declare a private method
   private function MyPrivate() { }
```

```
    // This is public
    function Foo()
    {
        $this->MyPublic();
        $this->MyProtected();
        $this->MyPrivate();
    }
}

$myclass = new MyClass;
$myclass->MyPublic(); // Works
$myclass->MyProtected(); // Fatal Error
$myclass->MyPrivate(); // Fatal Error
$myclass->Foo(); // Public, Protected and Private work


/**
* Define MyClass2
*/
class MyClass2 extends MyClass
{
    // This is public
    function Foo2()
    {
        $this->MyPublic();
        $this->MyProtected();
        $this->MyPrivate(); // Fatal Error
    }
}

$myclass2 = new MyClass2;
$myclass2->MyPublic(); // Works
$myclass2->Foo2(); // Public and Protected work, not Private

class Bar
{
    public function test() {
        $this->testPrivate();
        $this->testPublic();
    }

    public function testPublic() {
        echo "Bar::testPublic\n";
    }

    private function testPrivate() {
        echo "Bar::testPrivate\n";
    }
}

class Foo extends Bar
{
    public function testPublic() {
        echo "Foo::testPublic\n";
    }

    private function testPrivate() {
        echo "Foo::testPrivate\n";
    }
```

```
}

$myFoo = new foo();
$myFoo->test(); // Bar::testPrivate
                // Foo::testPublic
?>
```

Scope Resolution Operator (::)

The Scope Resolution Operator (also called Paamayim Nekudotayim) or in simpler terms, the double colon, is a token that allows access to static, constant, and overridden members or methods of a class.

When referencing these items from outside the class definition, use the name of the class.

As of PHP 5.3.0, it's possible to reference the class using a variable. The variable's value can not be a keyword (e.g. *self*, *parent* and *static*).

Paamayim Nekudotayim would, at first, seem like a strange choice for naming a double-colon. However, while writing the Zend Engine 0.5 (which powers PHP 3), that's what the Zend team decided to call it. It actually does mean double-colon - in Hebrew!

Example #87 -:: from outside the class definition

```
<?php
class MyClass {
    const CONST_VALUE = 'A constant value';
}

$classname = 'MyClass';
echo $classname::CONST_VALUE; // As of PHP 5.3.0

echo MyClass::CONST_VALUE;
?>
```

Two special keywords *self* and *parent* are used to access members or methods from inside the class definition.

Example #88 -:: from inside the class definition

```
<?php
class OtherClass extends MyClass
{
    public static $my_static = 'static var';

    public static function doubleColon() {
        echo parent::CONST_VALUE . "\n";
        echo self::$my_static . "\n";
    }
```

```
}

$classname = 'OtherClass';
echo $classname::doubleColon(); // As of PHP 5.3.0

OtherClass::doubleColon();
?>
```

When an extending class overrides the parents definition of a method, PHP will not call the parent's method. It's up to the extended class on whether or not the parent's method is called. This also applies to Constructors and Destructors, Overloading, and Magic method definitions.

Example #89 - Calling a parent's method

```
<?php
class MyClass
{
    protected function myFunc() {
        echo "MyClass::myFunc()\n";
    }
}

class OtherClass extends MyClass
{
    // Override parent's definition
    public function myFunc()
    {
        // But still call the parent function
        parent::myFunc();
        echo "OtherClass::myFunc()\n";
    }
}

$class = new OtherClass();
$class->myFunc();
?>
```

Static Keyword

Declaring class members or methods as static makes them accessible without needing an instantiation of the class. A member declared as static can not be accessed with an instantiated class object (though a static method can).

For compatibility with PHP 4, if no visibility declaration is used, then the member or method will be treated as if it was declared as *public*.

Because static methods are callable without an instance of the object created, the pseudo variable *$this* is not available inside the method declared as static.

Static properties cannot be accessed through the object using the arrow operator ->.

Calling non-static methods statically generates an E_STRICT level warning.

As of PHP 5.3.0, it's possible to reference the class using a variable. The variable's value can not be a keyword (e.g. *self*, *parent* and *static*).

Example #90 - Static member example

```
<?php
class Foo
{
   public static $my_static = 'foo';

   public function staticValue() {
       return self::$my_static;
   }
}

class Bar extends Foo
{
   public function fooStatic() {
       return parent::$my_static;
   }
}


print Foo::$my_static . "\n";

$foo = new Foo();
print $foo->staticValue() . "\n";
print $foo->my_static . "\n";      // Undefined "Property" my_static

print $foo::$my_static . "\n";
$classname = 'Foo';
print $classname::$my_static . "\n"; // As of PHP 5.3.0

print Bar::$my_static . "\n";
$bar = new Bar();
print $bar->fooStatic() . "\n";
?>
```

Example #91 - Static method example

```
<?php
class Foo {
   public static function aStaticMethod() {
       // ...
   }
}

Foo::aStaticMethod();
$classname = 'Foo';
$classname::aStaticMethod(); // As of PHP 5.3.0
?>
```

Class Constants

It is possible to define constant values on a per-class basis remaining the same and unchangeable. Constants differ from normal variables in that you don't use the *$*symbol to declare or use them.

The value must be a constant expression, not (for example) a variable, a class member, result of a mathematical operation or a function call.

As of PHP 5.3.0, it's possible to reference the class using a variable. The variable's value can not be a keyword (e.g. *self*, *parent* and *static*).

Example #92 - Defining and using a constant

```
<?php
class MyClass
{
   const constant = 'constant value';

   function showConstant() {
       echo  self::constant . "\n";
   }
}

echo MyClass::constant . "\n";

$classname = "MyClass";
echo $classname::constant . "\n"; // As of PHP 5.3.0

$class = new MyClass();
$class->showConstant();

echo $class::constant."\n"; // As of PHP 5.3.0
?>
```

Example #93 - Static data example

```
<?php
class foo {
   // As of PHP 5.3.0
   const bar = <<<'EOT'
bar
EOT;
}
?>
```

Unlike heredocs, nowdocs can be used in any static data context.

Note

Nowdoc support was added in PHP 5.3.0.

Class Abstraction

PHP 5 introduces abstract classes and methods. It is not allowed to create an instance of a class that has been defined as abstract. Any class that contains at least one abstract method must also be abstract. Methods defined as abstract simply declare the method's signature they cannot define the implementation.

When inheriting from an abstract class, all methods marked abstract in the parent's class declaration must be defined by the child; additionally, these methods must be defined with the same (or a less restricted) visibility. For example, if the abstract method is defined as protected, the function implementation must be defined as either protected or public, but not private.

Example #94 - Abstract class example

```
<?php
abstract class AbstractClass
{
   // Force Extending class to define this method
   abstract protected function getValue();
   abstract protected function prefixValue($prefix);

   // Common method
   public function printOut() {
       print $this->getValue() . "\n";
   }
}

class ConcreteClass1 extends AbstractClass
{
   protected function getValue() {
       return "ConcreteClass1";
   }

   public function prefixValue($prefix) {
       return "{$prefix}ConcreteClass1";
   }
}

class ConcreteClass2 extends AbstractClass
{
   public function getValue() {
       return "ConcreteClass2";
   }

   public function prefixValue($prefix) {
       return "{$prefix}ConcreteClass2";
   }
}

$class1 = new ConcreteClass1;
$class1->printOut();
echo $class1->prefixValue('FOO_') ."\n";

$class2 = new ConcreteClass2;
$class2->printOut();
```

```
echo $class2->prefixValue('FOO_') ."\n";
?>
```

The above example will output:

```
ConcreteClass1
FOO_ConcreteClass1
ConcreteClass2
FOO_ConcreteClass2
```

Old code that has no user-defined classes or functions named 'abstract' should run without modifications.

Object Interfaces

Object interfaces allow you to create code which specifies which methods a class must implement, without having to define how these methods are handled.

Interfaces are defined using the interface keyword, in the same way as a standard class, but without any of the methods having their contents defined.

All methods declared in an interface must be public, this is the nature of an interface.

implements

To implement an interface, the *implements* operator is used. All methods in the interface must be implemented within a class; failure to do so will result in a fatal error. Classes may implement more than one interface if desired by separating each interface with a comma.

Note

A class cannot implement two interfaces that share function names, since it would cause ambiguity.

Examples

Example #95 - Interface example

```
<?php
// Declare the interface 'iTemplate'
interface iTemplate
{
   public function setVariable($name, $var);
   public function getHtml($template);
}

// Implement the interface
```

```
// This will work
class Template implements iTemplate
{
   private $vars = array();

   public function setVariable($name, $var)
   {
       $this->vars[$name] = $var;
   }

   public function getHtml($template)
   {
       foreach($this->vars as $name => $value) {
           $template = str_replace('{' . $name . '}', $value, $template);
       }

       return $template;
   }
}

// This will not work
// Fatal error: Class BadTemplate contains 1 abstract methods
// and must therefore be declared abstract (iTemplate::getHtml)
class BadTemplate implements iTemplate
{
   private $vars = array();

   public function setVariable($name, $var)
   {
       $this->vars[$name] = $var;
   }
}

?>
```

See also the instanceof operator.

Overloading

Overloading in PHP provides means to dynamically "create" members and methods. These dynamic entities are processed via magic methods one can establish in a class for various action types.

The overloading methods are invoked when interacting with members or methods that have not been declared or are not visible in the current scope. The rest of this section will use the terms "inaccessible members" and "inaccessible methods" to refer to this combination of declaration and visibility.

All overloading methods must be defined as *public*.

Note
None of the arguments of these magic methods can be passed by reference.

Note
PHP's interpretation of "overloading" is different than most object oriented languages. Overloading traditionally provides the ability to have multiple methods with the same name but different quantities and types of arguments.

ChangeLog

Version	Description
5.1.0	Added *__isset()* and *__unset()*.
5.3.0	Added *__callStatic()*.

Member overloading

void **__set** (string `$name`, mixed `$value`)

mixed **__get** (string `$name`)

bool **__isset** (string `$name`)

void **__unset** (string `$name`)

__set() is run when writing data to inaccessible members.

__get() is utilized for reading data from inaccessible members.

__isset() is triggered by calling isset() or empty() on inaccessible members.

__unset() is invoked when unset() is used on inaccessible members.

The *$name* argument is the name of the member being interacted with. The *__set()* method's *$value* argument specifies the value the *$name* 'ed member should be set to.

Member overloading only works in object context. These magic methods will not be triggered in static context. Therefore these methods can not be declared static.

Example #96 - overloading with __get, __set, __isset and __unset example

```
<?php
class MemberTest {
   /**  Location for overloaded data.  */
```

```
    private $data = array();

    /**  Overloading not used on declared members.  */
    public $declared = 1;

    /**  Overloading not triggered when accessed inside the class.  */
    private $hidden = 2;

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

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

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

    /**  As of PHP 5.1.0  */
    public function __isset($name) {
        echo "Is '$name' set?\n";
        return isset($this->data[$name]);
    }

    /**  As of PHP 5.1.0  */
    public function __unset($name) {
        echo "Unsetting '$name'\n";
        unset($this->data[$name]);
    }

    /**  Not a magic method, just here for example.  */
    public function getHidden() {
        echo "'hidden' visible here so __get() not used\n";
        return $this->hidden;
    }
}


echo "<pre>\n";

$obj = new MemberTest;

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

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

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

```
echo $obj->getHidden() . "\n";
echo $obj->hidden . "\n";
?>
```

The above example will output:

```
Setting 'a' to '1'
Getting 'a'
1
Is 'a' set?
bool(true)
Unsetting 'a'
Is 'a' set?
bool(false)
1
'hidden' visible here so __get() not used
2
Getting 'hidden'


Notice:  Undefined property: hidden in <file> on line 64 in <file> on line
28
```

Method overloading

mixed **__call** (string `$name`, array `$arguments`)

mixed **__callStatic** (string `$name`, array `$arguments`)

__call() is triggered when invoking inaccessible methods in an object context.

__callStatic() is triggered when invoking inaccessible methods in a static context.

The *$name* argument is the name of the method being called. The *$arguments* argument is an enumerated array containing the parameters passed to the *$name* 'ed method.

Example #97 - overloading instantiated methods with __call and ___callStatic

```
<?php
class MethodTest {
   public function __call($name, $arguments) {
       // Note: value of $name is case sensitive.
       echo "Calling object method '$name' "
            . implode(', ', $arguments). "\n";
   }

   /**  As of PHP 5.3.0  */
   public static function __callStatic($name, $arguments) {
       // Note: value of $name is case sensitive.
       echo "Calling static method '$name' "
            . implode(', ', $arguments). "\n";
   }
}
```

```
$obj = new MethodTest;
$obj->runTest('in object context');

MethodTest::runTest('in static context');  // As of PHP 5.3.0
?>
```

The above example will output:

```
Calling object method 'runTest' in object context
Calling static method 'runTest' in static context
```

Object Iteration

PHP 5 provides a way for objects to be defined so it is possible to iterate through a list of items, with, for example a foreach statement. By default, all visible properties will be used for the iteration.

Example #98 - 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();

?>
```

The above example will output:

```
var1 => value 1
var2 => value 2
var3 => value 3
```

```
MyClass::iterateVisible:
var1 => value 1
var2 => value 2
var3 => value 3
protected => protected var
private => private var
```

As the output shows, the foreach iterated through all visible variables that can be accessed. To take it a step further you can *implement* one of PHP 5's internal interface named *Iterator*. This allows the object to decide what and how the object will be iterated.

Example #99 - Object Iteration implementing Iterator

```
<?php
class MyIterator implements Iterator
{
   private $var = array();

   public function __construct($array)
   {
       if (is_array($array)) {
           $this->var = $array;
       }
   }

   public function rewind() {
       echo "rewinding\n";
       reset($this->var);
   }

   public function current() {
       $var = current($this->var);
       echo "current: $var\n";
       return $var;
   }

   public function key() {
       $var = key($this->var);
       echo "key: $var\n";
       return $var;
   }

   public function next() {
       $var = next($this->var);
       echo "next: $var\n";
       return $var;
   }

   public function valid() {
       $var = $this->current() !== false;
       echo "valid: {$var}\n";
       return $var;
   }
}

$values = array(1,2,3);
$it = new MyIterator($values);
```

```
foreach ($it as $a => $b) {
   print "$a: $b\n";
}
?>
```

The above example will output:

```
rewinding
current: 1
valid: 1
current: 1
key: 0
0: 1
next: 2
current: 2
valid: 1
current: 2
key: 1
1: 2
next: 3
current: 3
valid: 1
current: 3
key: 2
2: 3
next:
current:
valid:
```

You can also define your class so that it doesn't have to define all the *Iterator* functions by simply implementing the PHP 5 *IteratorAggregate* interface.

Example #100 - Object Iteration implementing IteratorAggregate

```
<?php
class MyCollection implements IteratorAggregate
{
   private $items = array();
   private $count = 0;

   // Required definition of interface IteratorAggregate
   public function getIterator() {
       return new MyIterator($this->items);
   }

   public function add($value) {
       $this->items[$this->count++] = $value;
   }
}

$coll = new MyCollection();
$coll->add('value 1');
$coll->add('value 2');
$coll->add('value 3');

foreach ($coll as $key => $val) {
```

```
   echo "key/value: [$key -> $val]\n\n";
}
?>
```

The above example will output:

```
rewinding
current: value 1
valid: 1
current: value 1
key: 0
key/value: [0 -> value 1]

next: value 2
current: value 2
valid: 1
current: value 2
key: 1
key/value: [1 -> value 2]

next: value 3
current: value 3
valid: 1
current: value 3
key: 2
key/value: [2 -> value 3]

next:
current:
valid:
```

Note

For more examples of iterators, see the SPL Extension.

Patterns

Patterns are ways to describe best practices and good designs. They show a flexible solution to common programming problems.

Factory

The Factory pattern allows for the instantiation of objects at runtime. It is called a Factory Pattern since it is responsible for "manufacturing" an object. A Parameterized Factory receives the name of the class to instantiate as argument.

Example #101 - Parameterized Factory Method

```
<?php
class Example
```

```
{
   // The parameterized factory method
   public static function factory($type)
   {
       if (include_once 'Drivers/' . $type . '.php') {
           $classname = 'Driver_' . $type;
           return new $classname;
       } else {
           throw new Exception ('Driver not found');
       }
   }
}
?>
```

Defining this method in a class allows drivers to be loaded on the fly. If the *Example* class was a database abstraction class, loading a *MySQL* and *SQLite* driver could be done as follows:

```
<?php
// Load a MySQL Driver
$mysql = Example::factory('MySQL');

// Load a SQLite Driver
$sqlite = Example::factory('SQLite');
?>
```

Singleton

The Singleton pattern applies to situations in which there needs to be a single instance of a class. The most common example of this is a database connection. Implementing this pattern allows a programmer to make this single instance easily accessible by many other objects.

Example #102 - Singleton Function

```
<?php
class Example
{
   // Hold an instance of the class
   private static $instance;

   // A private constructor; prevents direct creation of object
   private function __construct()
   {
       echo 'I am constructed';
   }

   // The singleton method
   public static function singleton()
   {
       if (!isset(self::$instance)) {
           $c = __CLASS__;
           self::$instance = new $c;
       }
```

```
        return self::$instance;
    }

    // Example method
    public function bark()
    {
        echo 'Woof!';
    }

    // Prevent users to clone the instance
    public function __clone()
    {
        trigger_error('Clone is not allowed.', E_USER_ERROR);
    }

}

?>
```

This allows a single instance of the *Example* class to be retrieved.

```
<?php
// This would fail because the constructor is private
$test = new Example;

// This will always retrieve a single instance of the class
$test = Example::singleton();
$test->bark();

// This will issue an E_USER_ERROR.
$test_clone = clone $test;

?>
```

Magic Methods

The function names *__construct*, *__destruct* (see Constructors and Destructors), *__call*, *__callStatic*, *__get*, *__set*, *__isset*, *__unset* (see Overloading), *__sleep*, *__wakeup*, *__toString*, *__set_state* and __clone are magical in PHP classes. You cannot have functions with these names in any of your classes unless you want the magic functionality associated with them.

Caution
PHP reserves all function names starting with __ as magical. It is recommended that you do not use function names with __ in PHP unless you want some documented magic functionality.

__sleep and *__wakeup*

serialize() checks if your class has a function with the magic name *__sleep*. If so, that

function is executed prior to any serialization. It can clean up the object and is supposed to return an array with the names of all variables of that object that should be serialized. If the method doesn't return anything then **NULL** is serialized and E_NOTICE is issued.

The intended use of *__sleep* is to commit pending data or perform similar cleanup tasks. Also, the function is useful if you have very large objects which do not need to be saved completely.

Conversely, unserialize() checks for the presence of a function with the magic name *__wakeup*. If present, this function can reconstruct any resources that the object may have.

The intended use of *__wakeup* is to reestablish any database connections that may have been lost during serialization and perform other reinitialization tasks.

Example #103 - Sleep and wakeup

```
<?php
class Connection {
   protected $link;
   private $server, $username, $password, $db;

   public function __construct($server, $username, $password, $db)
   {
       $this->server = $server;
       $this->username = $username;
       $this->password = $password;
       $this->db = $db;
       $this->connect();
   }

   private function connect()
   {
       $this->link = mysql_connect($this->server, $this->username,
$this->password);
       mysql_select_db($this->db, $this->link);
   }

   public function __sleep()
   {
       return array('server', 'username', 'password', 'db');
   }

   public function __wakeup()
   {
       $this->connect();
   }
}
?>
```

__toString

The *__toString* method allows a class to decide how it will react when it is converted to a

string.

Example #104 - Simple example

```
<?php
// Declare a simple class
class TestClass
{
   public $foo;

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

   public function __toString() {
       return $this->foo;
   }
}

$class = new TestClass('Hello');
echo $class;
?>
```

The above example will output:

```
Hello
```

It is worth noting that before PHP 5.2.0 the *__toString* method was only called when it was directly combined with echo() or print(). Since PHP 5.2.0, it is called in any string context (e.g. in printf() with *%s* modifier) but not in other types contexts (e.g. with *%d* modifier). Since PHP 5.2.0, converting objects without *__toString* method to string would cause **E_RECOVERABLE_ERROR**.

__set_state

This static method is called for classes exported by var_export() since PHP 5.1.0.

The only parameter of this method is an array containing exported properties in the form *array('property' => value, ...)*.

Example #105 - Using *__set_state* (since PHP 5.1.0)

```
<?php

class A
{
   public $var1;
   public $var2;

   public static function __set_state($an_array) // As of PHP 5.1.0
   {
       $obj = new A;
```

```
        $obj->var1 = $an_array['var1'];
        $obj->var2 = $an_array['var2'];
        return $obj;
    }
}

$a = new A;
$a->var1 = 5;
$a->var2 = 'foo';

eval('$b = ' . var_export($a, true) . ';'); // $b = A::__set_state(array(
                                            //    'var1' => 5,
                                            //    'var2' => 'foo',
                                            // ));
var_dump($b);

?>
```

The above example will output:

```
object(A)#2 (2) {
 ["var1"]=>
 int(5)
 ["var2"]=>
 string(3) "foo"
}
```

Final Keyword

PHP 5 introduces the final keyword, which prevents child classes from overriding a method by prefixing the definition with final. If the class itself is being defined final then it cannot be extended.

Example #106 - Final methods example

```
<?php
class BaseClass {
  public function test() {
      echo "BaseClass::test() called\n";
  }

  final public function moreTesting() {
      echo "BaseClass::moreTesting() called\n";
  }
}

class ChildClass extends BaseClass {
  public function moreTesting() {
      echo "ChildClass::moreTesting() called\n";
  }
}
// Results in Fatal error: Cannot override final method
BaseClass::moreTesting()
?>
```

Example #107 - Final class example

```
<?php
final class BaseClass {
  public function test() {
      echo "BaseClass::test() called\n";
  }

  // Here it doesn't matter if you specify the function as final or not
  final public function moreTesting() {
      echo "BaseClass::moreTesting() called\n";
  }
}

class ChildClass extends BaseClass {
}
// Results in Fatal error: Class ChildClass may not inherit from final class
(BaseClass)
?>
```

Object cloning

Creating a copy of an object with fully replicated properties is not always the wanted behavior. A good example of the need for copy constructors, is if you have an object which represents a GTK window and the object holds the resource of this GTK window, when you create a duplicate you might want to create a new window with the same properties and have the new object hold the resource of the new window. Another example is if your object holds a reference to another object which it uses and when you replicate the parent object you want to create a new instance of this other object so that the replica has its own separate copy.

An object copy is created by using the clone keyword (which calls the object's __clone() method if possible). An object's __clone() method cannot be called directly.

```
$copy_of_object = clone $object;
```

When an object is cloned, PHP 5 will perform a shallow copy of all of the object's properties. Any properties that are references to other variables, will remain references. If a __clone() method is defined, then the newly created object's __clone() method will be called, to allow any necessary properties that need to be changed.

Example #108 - Cloning an object

```
<?php
class SubObject
{
   static $instances = 0;
   public $instance;
```

```
    public function __construct() {
        $this->instance = ++self::$instances;
    }

    public function __clone() {
        $this->instance = ++self::$instances;
    }
}

class MyCloneable
{
    public $object1;
    public $object2;

    function __clone()
    {
        // Force a copy of this->object, otherwise
        // it will point to same object.
        $this->object1 = clone $this->object1;
    }
}

$obj = new MyCloneable();

$obj->object1 = new SubObject();
$obj->object2 = new SubObject();

$obj2 = clone $obj;


print("Original Object:\n");
print_r($obj);

print("Cloned Object:\n");
print_r($obj2);

?>
```

The above example will output:

```
Original Object:
MyCloneable Object
(
    [object1] => SubObject Object
        (
            [instance] => 1
        )

    [object2] => SubObject Object
        (
            [instance] => 2
        )

)
Cloned Object:
MyCloneable Object
(
    [object1] => SubObject Object
        (
            [instance] => 3
```

```
        )

    [object2] => SubObject Object
        (
            [instance] => 2
        )

)
```

Comparing objects

In PHP 5, object comparison is more complicated than in PHP 4 and more in accordance to what one will expect from an Object Oriented Language (not that PHP 5 is such a language).

When using the comparison operator (==), object variables are compared in a simple manner, namely: Two object instances are equal if they have the same attributes and values, and are instances of the same class.

On the other hand, when using the identity operator (===), object variables are identical if and only if they refer to the same instance of the same class.

An example will clarify these rules.

Example #109 - Example of object comparison in PHP 5

```php
<?php
function bool2str($bool)
{
   if ($bool === false) {
       return 'FALSE';
   } else {
       return 'TRUE';
   }
}

function compareObjects(&$o1, &$o2)
{
   echo 'o1 == o2 : ' . bool2str($o1 == $o2) . "\n";
   echo 'o1 != o2 : ' . bool2str($o1 != $o2) . "\n";
   echo 'o1 === o2 : ' . bool2str($o1 === $o2) . "\n";
   echo 'o1 !== o2 : ' . bool2str($o1 !== $o2) . "\n";
}

class Flag
{
   public $flag;

   function Flag($flag = true) {
       $this->flag = $flag;
   }
}

class OtherFlag
{
```

```
   public $flag;

   function OtherFlag($flag = true) {
       $this->flag = $flag;
   }
}

$o = new Flag();
$p = new Flag();
$q = $o;
$r = new OtherFlag();

echo "Two instances of the same class\n";
compareObjects($o, $p);

echo "\nTwo references to the same instance\n";
compareObjects($o, $q);

echo "\nInstances of two different classes\n";
compareObjects($o, $r);
?>
```

The above example will output:

```
Two instances of the same class
o1 == o2 : TRUE
o1 != o2 : FALSE
o1 === o2 : FALSE
o1 !== o2 : TRUE

Two references to the same instance
o1 == o2 : TRUE
o1 != o2 : FALSE
o1 === o2 : TRUE
o1 !== o2 : FALSE

Instances of two different classes
o1 == o2 : FALSE
o1 != o2 : TRUE
o1 === o2 : FALSE
o1 !== o2 : TRUE
```

Note
Extensions can define own rules for their objects comparison.

Reflection

Table of Contents

Introduction

PHP 5 comes with a complete reflection API that adds the ability to reverse-engineer classes, interfaces, functions and methods as well as extensions. Additionally, the reflection API also offers ways of retrieving doc comments for functions, classes and methods.

The reflection API is an object-oriented extension to the Zend Engine, consisting of the following classes:

```
<?php
class Reflection { }
interface Reflector { }
class ReflectionException extends Exception { }
class ReflectionFunction extends ReflectionFunctionAbstract implements Reflector
{ }
class ReflectionParameter implements Reflector { }
class ReflectionMethod extends ReflectionFunctionAbstract implements Reflector {
}
class ReflectionClass implements Reflector { }
class ReflectionObject extends ReflectionClass { }
class ReflectionProperty implements Reflector { }
class ReflectionExtension implements Reflector { }
?>
```

Note

For details on these classes, have a look at the next chapters.

If we were to execute the code in the example below:

Example #110 - Basic usage of the reflection API

```
<?php
Reflection::export(new ReflectionClass('Exception'));
?>
```

The above example will output:

```
Class [ <internal> class Exception ] {

 - Constants [0] {
 }

 - Static properties [0] {
 }

 - Static methods [0] {
 }

 - Properties [6] {
   Property [ <default> protected $message ]
   Property [ <default> private $string ]
   Property [ <default> protected $code ]
   Property [ <default> protected $file ]
   Property [ <default> protected $line ]
   Property [ <default> private $trace ]
 }

 - Methods [9] {
   Method [ <internal> final private method __clone ] {
   }

   Method [ <internal, ctor> public method __construct ] {

     - Parameters [2] {
       Parameter #0 [ <optional> $message ]
       Parameter #1 [ <optional> $code ]
     }
   }

   Method [ <internal> final public method getMessage ] {
   }

   Method [ <internal> final public method getCode ] {
   }

   Method [ <internal> final public method getFile ] {
   }

   Method [ <internal> final public method getLine ] {
   }

   Method [ <internal> final public method getTrace ] {
   }

   Method [ <internal> final public method getTraceAsString ] {
   }

   Method [ <internal> public method __toString ] {
   }
 }
}
```

Reflector

Reflector is an interface implemented by all exportable Reflection classes.

```
<?php
interface Reflector
{
   public string __toString()
   public static string export()
}
?>
```

ReflectionException

ReflectionException extends the standard Exception and is thrown by Reflection API. No specific methods or properties are introduced.

ReflectionFunction

The ReflectionFunction class lets you reverse-engineer functions.

```
<?php
class ReflectionFunction extends ReflectionFunctionAbstract implements Reflector
{
   final private __clone()
   public void __construct(string name)
   public string __toString()
   public static string export(string name, bool return)
   public string getName()
   public bool isInternal()
   public bool isDisabled()
   public bool isUserDefined()
   public string getFileName()
   public int getStartLine()
   public int getEndLine()
   public string getDocComment()
   public array getStaticVariables()
   public mixed invoke([mixed args [, ...]])
   public mixed invokeArgs(array args)
   public bool returnsReference()
   public ReflectionParameter[] getParameters()
   public int getNumberOfParameters()
   public int getNumberOfRequiredParameters()
}
?>
```

Parent class ReflectionFunctionAbstract has the same methods except **invoke()**, **invokeArgs()**, **export()** and **isDisabled()**.

Note

getNumberOfParameters() and **getNumberOfRequiredParameters()** were added in PHP 5.0.3, while **invokeArgs()** was added in PHP 5.1.0.

To introspect a function, you will first have to create an instance of the ReflectionFunction class. You can then call any of the above methods on this instance.

Example #111 - Using the ReflectionFunction class

```
<?php
/**
* A simple counter
*
* @return    int
*/
function counter()
{
   static $c = 0;
   return $c++;
}

// Create an instance of the ReflectionFunction class
$func = new ReflectionFunction('counter');

// Print out basic information
printf(
   "===> The %s function '%s'\n".
   "     declared in %s\n".
   "     lines %d to %d\n",
   $func->isInternal() ? 'internal' : 'user-defined',
   $func->getName(),
   $func->getFileName(),
   $func->getStartLine(),
   $func->getEndline()
);

// Print documentation comment
printf("---> Documentation:\n %s\n", var_export($func->getDocComment(), 1));

// Print static variables if existant
if ($statics = $func->getStaticVariables())
{
   printf("---> Static variables: %s\n", var_export($statics, 1));
}

// Invoke the function
printf("---> Invokation results in: ");
var_dump($func->invoke());


// you may prefer to use the export() method
echo "\nReflectionFunction::export() results:\n";
echo ReflectionFunction::export('counter');
?>
```

Note

The method **invoke()** accepts a variable number of arguments which are passed to the function just as in call_user_func().

ReflectionParameter

The ReflectionParameter class retrieves information about a function's or method's parameters.

```
<?php
class ReflectionParameter implements Reflector
{
   final private __clone()
   public void __construct(string function, string parameter)
   public string __toString()
   public static string export(mixed function, mixed parameter, bool return)
   public string getName()
   public bool isPassedByReference()
   public ReflectionClass getDeclaringClass()
   public ReflectionClass getClass()
   public bool isArray()
   public bool allowsNull()
   public bool isPassedByReference()
   public bool isOptional()
   public bool isDefaultValueAvailable()
   public mixed getDefaultValue()
   public int getPosition()
}
?>
```

Note

getDefaultValue(), **isDefaultValueAvailable()** and **isOptional()** were added in PHP 5.0.3, while **isArray()** was added in PHP 5.1.0. **getDeclaringFunction()** and **getPosition()** were added in PHP 5.2.3.

To introspect function parameters, you will first have to create an instance of the ReflectionFunction or ReflectionMethod classes and then use their **getParameters()** method to retrieve an array of parameters.

Example #112 - Using the ReflectionParameter class

```
<?php
function foo($a, $b, $c) { }
function bar(Exception $a, &$b, $c) { }
function baz(ReflectionFunction $a, $b = 1, $c = null) { }
```

```
function abc() { }

// Create an instance of ReflectionFunction with the
// parameter given from the command line.
$reflect = new ReflectionFunction($argv[1]);

echo $reflect;

foreach ($reflect->getParameters() as $i => $param) {
   printf(
       "-- Parameter #%d: %s {\n".
       "   Class: %s\n".
       "   Allows NULL: %s\n".
       "   Passed to by reference: %s\n".
       "   Is optional?: %s\n".
       "}\n",
       $i, // $param->getPosition() can be used from PHP 5.2.3
       $param->getName(),
       var_export($param->getClass(), 1),
       var_export($param->allowsNull(), 1),
       var_export($param->isPassedByReference(), 1),
       $param->isOptional() ? 'yes' : 'no'
   );
}
?>
```

ReflectionClass

The ReflectionClass class lets you reverse-engineer classes and interfaces.

```
<?php
class ReflectionClass implements Reflector
{
   final private __clone()
   public void __construct(string name)
   public string __toString()
   public static string export(mixed class, bool return)
   public string getName()
   public bool isInternal()
   public bool isUserDefined()
   public bool isInstantiable()
   public bool hasConstant(string name)
   public bool hasMethod(string name)
   public bool hasProperty(string name)
   public string getFileName()
   public int getStartLine()
   public int getEndLine()
   public string getDocComment()
   public ReflectionMethod getConstructor()
   public ReflectionMethod getMethod(string name)
   public ReflectionMethod[] getMethods()
   public ReflectionProperty getProperty(string name)
   public ReflectionProperty[] getProperties()
   public array getConstants()
   public mixed getConstant(string name)
   public ReflectionClass[] getInterfaces()
```

```
   public bool isInterface()
   public bool isAbstract()
   public bool isFinal()
   public int getModifiers()
   public bool isInstance(stdclass object)
   public stdclass newInstance(mixed args)
   public stdclass newInstanceArgs(array args)
   public ReflectionClass getParentClass()
   public bool isSubclassOf(ReflectionClass class)
   public array getStaticProperties()
   public mixed getStaticPropertyValue(string name [, mixed default])
   public void setStaticPropertyValue(string name, mixed value)
   public array getDefaultProperties()
   public bool isIterateable()
   public bool implementsInterface(string name)
   public ReflectionExtension getExtension()
   public string getExtensionName()
}
?>
```

Note

hasConstant(), **hasMethod()**, **hasProperty()**, **getStaticPropertyValue()** and **setStaticPropertyValue()** were added in PHP 5.1.0, while **newInstanceArgs()** was added in PHP 5.1.3.

To introspect a class, you will first have to create an instance of the ReflectionClass class. You can then call any of the above methods on this instance.

Example #113 - Using the ReflectionClass class

```
<?php
interface Serializable
{
   // ...
}

class Object
{
   // ...
}

/**
* A counter class
*/
class Counter extends Object implements Serializable
{
   const START = 0;
   private static $c = Counter::START;

   /**
    * Invoke counter
    *
    * @access  public
```

```
     * @return  int
     */
    public function count() {
        return self::$c++;
    }
}

// Create an instance of the ReflectionClass class
$class = new ReflectionClass('Counter');

// Print out basic information
printf(
    "===> The %s%s%s %s '%s' [extends %s]\n" .
    "     declared in %s\n" .
    "     lines %d to %d\n" .
    "     having the modifiers %d [%s]\n",
        $class->isInternal() ? 'internal' : 'user-defined',
        $class->isAbstract() ? ' abstract' : '',
        $class->isFinal() ? ' final' : '',
        $class->isInterface() ? 'interface' : 'class',
        $class->getName(),
        var_export($class->getParentClass(), 1),
        $class->getFileName(),
        $class->getStartLine(),
        $class->getEndline(),
        $class->getModifiers(),
        implode(' ', Reflection::getModifierNames($class->getModifiers()))
);

// Print documentation comment
printf("---> Documentation:\n %s\n", var_export($class->getDocComment(),
1));

// Print which interfaces are implemented by this class
printf("---> Implements:\n %s\n", var_export($class->getInterfaces(), 1));

// Print class constants
printf("---> Constants: %s\n", var_export($class->getConstants(), 1));

// Print class properties
printf("---> Properties: %s\n", var_export($class->getProperties(), 1));

// Print class methods
printf("---> Methods: %s\n", var_export($class->getMethods(), 1));

// If this class is instantiable, create an instance
if ($class->isInstantiable()) {
    $counter = $class->newInstance();

    echo '---> $counter is instance? ';
    echo $class->isInstance($counter) ? 'yes' : 'no';

    echo "\n---> new Object() is instance? ";
    echo $class->isInstance(new Object()) ? 'yes' : 'no';
}
?>
```

Note
The method **newInstance()** accepts a variable number of arguments which are passed to the function just as in call_user_func().

Note
$class = new ReflectionClass('Foo'); $class->isInstance($arg) is equivalent to *$arg instanceof Foo* or *is_a($arg, 'Foo')*.

ReflectionObject

The ReflectionObject class lets you reverse-engineer objects.

```
<?php
class ReflectionObject extends ReflectionClass
{
   final private __clone()
   public void __construct(mixed object)
   public string __toString()
   public static string export(mixed object, bool return)
}
?>
```

ReflectionMethod

The ReflectionMethod class lets you reverse-engineer class methods.

```
<?php
class ReflectionMethod extends ReflectionFunctionAbstract implements Reflector
{
   public void __construct(mixed class, string name)
   public string __toString()
   public static string export(mixed class, string name, bool return)
   public mixed invoke(stdclass object [, mixed args [, ...]])
   public mixed invokeArgs(stdclass object, array args)
   public bool isFinal()
   public bool isAbstract()
   public bool isPublic()
   public bool isPrivate()
   public bool isProtected()
   public bool isStatic()
   public bool isConstructor()
   public bool isDestructor()
   public int getModifiers()
   public ReflectionClass getDeclaringClass()
```

```
   // Inherited from ReflectionFunctionAbstract
   final private __clone()
   public string getName()
   public bool isInternal()
   public bool isUserDefined()
   public string getFileName()
   public int getStartLine()
   public int getEndLine()
   public string getDocComment()
   public array getStaticVariables()
   public bool returnsReference()
   public ReflectionParameter[] getParameters()
   public int getNumberOfParameters()
   public int getNumberOfRequiredParameters()
}
?>
```

To introspect a method, you will first have to create an instance of the ReflectionMethod class. You can then call any of the above methods on this instance.

Example #114 - Using the ReflectionMethod class

```
<?php
class Counter
{
   private static $c = 0;

   /**
    * Increment counter
    *
    * @final
    * @static
    * @access  public
    * @return  int
    */
   final public static function increment()
   {
       return ++self::$c;
   }
}

// Create an instance of the ReflectionMethod class
$method = new ReflectionMethod('Counter', 'increment');

// Print out basic information
printf(
   "===> The %s%s%s%s%s%s%s method '%s' (which is %s)\n" .
   "     declared in %s\n" .
   "     lines %d to %d\n" .
   "     having the modifiers %d[%s]\n",
       $method->isInternal() ? 'internal' : 'user-defined',
       $method->isAbstract() ? ' abstract' : '',
       $method->isFinal() ? ' final' : '',
       $method->isPublic() ? ' public' : '',
       $method->isPrivate() ? ' private' : '',
       $method->isProtected() ? ' protected' : '',
       $method->isStatic() ? ' static' : '',
```

```
        $method->getName(),
        $method->isConstructor() ? 'the constructor' : 'a regular method',
        $method->getFileName(),
        $method->getStartLine(),
        $method->getEndline(),
        $method->getModifiers(),
        implode(' ', Reflection::getModifierNames($method->getModifiers()))
);

// Print documentation comment
printf("---> Documentation:\n %s\n", var_export($method->getDocComment(),
1));

// Print static variables if existant
if ($statics= $method->getStaticVariables()) {
   printf("---> Static variables: %s\n", var_export($statics, 1));
}

// Invoke the method
printf("---> Invokation results in: ");
var_dump($method->invoke(NULL));
?>
```

Note
Trying to invoke private, protected or abstract methods will result in an exception being thrown from the **invoke()** method.

Note
For static methods as seen above, you should pass NULL as the first argument to **invoke()**. For non-static methods, pass an instance of the class.

ReflectionProperty

The ReflectionProperty class lets you reverse-engineer class properties.

```
<?php
class ReflectionProperty implements Reflector
{
   final private __clone()
   public void __construct(mixed class, string name)
   public string __toString()
   public static string export(mixed class, string name, bool return)
   public string getName()
   public bool isPublic()
   public bool isPrivate()
   public bool isProtected()
   public bool isStatic()
   public bool isDefault()
```

```
   public int getModifiers()
   public mixed getValue(stdclass object)
   public void setValue(stdclass object, mixed value)
   public ReflectionClass getDeclaringClass()
   public string getDocComment()
}
?>
```

Note

getDocComment() was added in PHP 5.1.0.

To introspect a property, you will first have to create an instance of the ReflectionProperty class. You can then call any of the above methods on this instance.

Example #115 - Using the ReflectionProperty class

```
<?php
class String
{
   public $length  = 5;
}

// Create an instance of the ReflectionProperty class
$prop = new ReflectionProperty('String', 'length');

// Print out basic information
printf(
   "===> The%s%s%s%s property '%s' (which was %s)\n" .
   "     having the modifiers %s\n",
       $prop->isPublic() ? ' public' : '',
       $prop->isPrivate() ? ' private' : '',
       $prop->isProtected() ? ' protected' : '',
       $prop->isStatic() ? ' static' : '',
       $prop->getName(),
       $prop->isDefault() ? 'declared at compile-time' : 'created at
run-time',
       var_export(Reflection::getModifierNames($prop->getModifiers()), 1)
);

// Create an instance of String
$obj= new String();

// Get current value
printf("---> Value is: ");
var_dump($prop->getValue($obj));

// Change value
$prop->setValue($obj, 10);
printf("---> Setting value to 10, new value is: ");
var_dump($prop->getValue($obj));

// Dump object
var_dump($obj);
```

```
?>
```

Note

Trying to get or set private or protected class property's values will result in an exception being thrown.

ReflectionExtension

The ReflectionExtension class lets you reverse-engineer extensions. You can retrieve all loaded extensions at runtime using the get_loaded_extensions().

```
<?php
class ReflectionExtension implements Reflector {
   final private __clone()
   public void __construct(string name)
   public string __toString()
   public static string export(string name, bool return)
   public string getName()
   public string getVersion()
   public ReflectionFunction[] getFunctions()
   public array getConstants()
   public array getINIEntries()
   public ReflectionClass[] getClasses()
   public array getClassNames()
   public string info()
}
?>
```

To introspect an extension, you will first have to create an instance of the ReflectionExtension class. You can then call any of the above methods on this instance.

Example #116 - Using the ReflectionExtension class

```
<?php
// Create an instance of the ReflectionProperty class
$ext = new ReflectionExtension('standard');

// Print out basic information
printf(
   "Name        : %s\n" .
   "Version     : %s\n" .
   "Functions   : [%d] %s\n" .
   "Constants   : [%d] %s\n" .
   "INI entries : [%d] %s\n" .
   "Classes     : [%d] %s\n",
       $ext->getName(),
       $ext->getVersion() ? $ext->getVersion() : 'NO_VERSION',
       sizeof($ext->getFunctions()),
       var_export($ext->getFunctions(), 1),
```

```
        sizeof($ext->getConstants()),
        var_export($ext->getConstants(), 1),

        sizeof($ext->getINIEntries()),
        var_export($ext->getINIEntries(), 1),

        sizeof($ext->getClassNames()),
        var_export($ext->getClassNames(), 1)
);
?>
```

Extending the reflection classes

In case you want to create specialized versions of the built-in classes (say, for creating colorized HTML when being exported, having easy-access member variables instead of methods or having utility methods), you may go ahead and extend them.

Example #117 - Extending the built-in classes

```
<?php
/**
* My Reflection_Method class
*/
class My_Reflection_Method extends ReflectionMethod
{
   public $visibility = array();

   public function __construct($o, $m)
   {
       parent::__construct($o, $m);
       $this->visibility =
Reflection::getModifierNames($this->getModifiers());
   }
}

/**
* Demo class #1
*
*/
class T {
   protected function x() {}
}

/**
* Demo class #2
*
*/
class U extends T {
   function x() {}
}

// Print out information
var_dump(new My_Reflection_Method('U', 'x'));
?>
```

Note

Caution: If you're overwriting the constructor, remember to call the parent's constructor _before_ any code you insert. Failing to do so will result in the following: *Fatal error: Internal error: Failed to retrieve the reflection object*

Type Hinting

PHP 5 introduces Type Hinting. Functions are now able to force parameters to be objects (by specifying the name of the class in the function prototype) or arrays (since PHP 5.1). However, if NULL is used as the default parameter value, it will be allowed as an argument for any later call.

Example #118 - Type Hinting examples

```
<?php
// An example class
class MyClass
{
   /**
    * A test function
    *
    * First parameter must be an object of type OtherClass
    */
   public function test(OtherClass $otherclass) {
       echo $otherclass->var;
   }


   /**
    * Another test function
    *
    * First parameter must be an array
    */
   public function test_array(array $input_array) {
       print_r($input_array);
   }
}

// Another example class
class OtherClass {
   public $var = 'Hello World';
}
?>
```

Failing to satisfy the type hint results in a catchable fatal error.

```
<?php
// An instance of each class
$myclass = new MyClass;
$otherclass = new OtherClass;

// Fatal Error: Argument 1 must be an object of class OtherClass
```

```
$myclass->test('hello');

// Fatal Error: Argument 1 must be an instance of OtherClass
$foo = new stdClass;
$myclass->test($foo);

// Fatal Error: Argument 1 must not be null
$myclass->test(null);

// Works: Prints Hello World
$myclass->test($otherclass);

// Fatal Error: Argument 1 must be an array
$myclass->test_array('a string');

// Works: Prints the array
$myclass->test_array(array('a', 'b', 'c'));
?>
```

Type hinting also works with functions:

```
<?php
// An example class
class MyClass {
   public $var = 'Hello World';
}

/**
* A test function
*
* First parameter must be an object of type MyClass
*/
function MyFunction (MyClass $foo) {
   echo $foo->var;
}

// Works
$myclass = new MyClass;
MyFunction($myclass);
?>
```

Type hinting allowing NULL value:

```
<?php

/* Accepting NULL value */
function test(stdClass $obj = NULL) {

}

test(NULL);
test(new stdClass);

?>
```

Type Hints can only be of the object and array (since PHP 5.1) type. Traditional type hinting with int and string isn't supported.

Late Static Bindings

As of PHP 5.3.0, PHP implements a feature called late static bindings which can be used to reference the called class in a context of static inheritance.

This feature was named "late static bindings" with an internal perspective in mind. "Late binding" comes from the fact that *static::* will no longer be resolved using the class where the method is defined but it will rather be computed using runtime information. It was also called a "static binding" as it can be used for (but is not limited to) static method calls.

Limitations of *self::*

Static references to the current class like *self::* or *__CLASS__* are resolved using the class in which the function belongs, as in where it was defined:

Example #119 - *self::* **usage**

```
<?php
class A {
   public static function who() {
       echo __CLASS__;
   }
   public static function test() {
       self::who();
   }
}

class B extends A {
   public static function who() {
        echo __CLASS__;
   }
}

B::test();
?>
```

The above example will output:

```
A
```

Late Static Bindings' usage

Late static bindings tries to solve that limitation by introducing a keyword that references the class that was initially called at runtime. Basically, a keyword that would allow you to reference *B* from *test()* in the previous example. It was decided not to introduce a new keyword but rather use *static* that was already reserved.

Example #120 - *static::* **simple usage**

```
<?php
class A {
   public static function who() {
       echo __CLASS__;
   }
   public static function test() {
       static::who(); // Here comes Late Static Bindings
   }
}

class B extends A {
   public static function who() {
        echo __CLASS__;
   }
}

B::test();
?>
```

The above example will output:

```
B
```

Note

static:: does not work like *$this* for static methods! *$this->* follows the rules of inheritance while *static::* doesn't. This difference is detailed later on this manual page.

Example #121 - *static::* **usage in a non-static context**

```
<?php
class TestChild extends TestParent {
   public function __construct() {
       static::who();
   }

   public function test() {
       $o = new TestParent();
   }

   public static function who() {
       echo __CLASS__."\n";
   }
}

class TestParent {
   public function __construct() {
       static::who();
   }

   public static function who() {
```

```
        echo __CLASS__."\n";
    }
}
$o = new TestChild;
$o->test();

?>
```

The above example will output:

```
TestChild
TestParent
```

Note

Late static bindings' resolution will stop at a fully resolved static call with no fallback.

Example #122 - Fully resolved static calls

```
<?php
class A {
    public static function foo() {
        static::who();
    }

    public static function who() {
        echo __CLASS__."\n";
    }
}

class B extends A {
    public static function test() {
        A::foo();
    }

    public static function who() {
        echo __CLASS__."\n";
    }
}

B::test();
?>
```

The above example will output:

```
A
```

Edge cases

There are lots of different ways to trigger a method call in PHP, like callbacks or magic methods. As late static bindings base their resolution on runtime information, it might give

unexpected results in so-called edge cases.

Example #123 - Late static bindings inside magic methods

```
<?php
class A {

  protected static function who() {
       echo __CLASS__."\n";
  }

  public function __get($var) {
      return static::who();
  }
}

class B extends A {

  protected static function who() {
       echo __CLASS__."\n";
  }
}

$b = new B;
$b->foo;
?>
```

The above example will output:

```
B
```

Namespaces

Namespaces overview

Namespaces in PHP are designed to solve scoping problem in large PHP libraries. In PHP, all class definitions are global. Thus, when a library author creates various utility or public API classes for the library, he must be aware of the possibility that other libraries with similar functionality would exist and thus choose unique names so that these libraries could be used together. Usually it is solved by prefixing the class names with an unique string - e.g., database classes would have prefix My_Library_DB, etc. As the library grows, prefixes add up, leading to the very long names.

The namespaces allow the developer to manage naming scopes without using the long names each time the class is referred to, and solve the problem of shared globals space without making code unreadable.

Namespaces are available in PHP as of PHP 5.3.0. This section is experimental and subject to changes.

Namespace definition

The namespace is declared using *namespace* keyword, which should be at the very beginning of the file. Example:

Example #124 - Defining namespace

```
<?php
   namespace MyProject::DB;

   const CONNECT_OK = 1;

   class Connection { /* ... */ }

   function connect() { /* ... */  }

?>
```

Same namespace name can be used in multiple files.

Namespace can contain class, constant and function definitions, but no free code.

Namespace definition does the following:

- Inside namespace, all class, function and constant names in definitions are automatically prefixed with namespace name. The class name is always the full name, i.e. in the example above the class is called MyProject::DB::Connection.
- Constant definitions create constant which is composed of namespace name and constant name. Like class constants, namespace constant can only contains static

values.

- Unqualified class name (i.e., name not containing*::*) is resolved at runtime following this procedure:
  - Class is looked up inside the current namespace (i.e. prefixing the name with the current namespace name) without attempting to autoload.
  - Class is looked up inside the global namespace without attempting to autoload.
  - Autoloading for name in current namespace is attempted.
  - If previous failed, lookup fails.

- Unqualified function name (i.e., name not containing*::*) is looked up at runtime first in the current namespace and then in the global space.
- Unqualified constant names are looked up first at current namespace and then among globally defined constants.

See also the full name resolution rules.

Using namespaces

Every class and function in a namespace can be referred to by the full name - e.g. MyProject::DB::Connection or MyProject::DB::connect - at any time.

Example #125 - Using namespaced name

```
<?php
   require 'MyProject/Db/Connection.php';
   $x = new MyProject::DB::Connection;
   MyProject::DB::connect();
?>
```

Namespaces can be imported into current context (global or namespace) using the *use* operator. The syntax for the operator is:

```
<?php
/* ... */
use Some::Name as Othername;

// The simplified form of use:
use Foo::Bar;
// which is the same as :
use Foo::Bar as Bar;
?>
```

The imported name works as follows: every time that the compiler encounters the local name *Othername* (as stand-alone name or as prefix to the longer name separated by*::*) the imported name *Some::Name* is substituted instead.

use can be used only in global scope, not inside function or class. Imported names have

effect from the point of import to the end of the current file. It is recommended to put imports at the beginning of the file to avoid confusion.

Example #126 - Importing and accessing namespace

```
<?php
   require 'MyProject/Db/Connection.php';
   use MyProject::DB;
   use MyProject::DB::Connection as DbConnection;

   $x = new MyProject::DB::Connection();
   $y = new DB::connection();
   $z = new DbConnection();
   DB::connect();
?>
```

Note

The import operation is compile-time only, all local names are converted to their full equivalents by the compiler. Note that it won't translate names in strings, so callbacks can't rely on import rules.

Global space

Without any namespace definition, all class and function definitions are placed into the global space - as it was in PHP before namespaces were supported. Prefixing a name with *::* will specify that the name is required from the global space even in the context of the namespace.

Example #127 - Using global space specification

```
<?php
   namespace A::B::C;

/* This function is A::B::C::fopen */
   function fopen() {
        /* ... */
        $f = ::fopen(...); // call global fopen
        return $f;
   }
?>
```

__NAMESPACE__

The compile-time constant __**NAMESPACE**__ is defined to the name of the current namespace. Outside namespace this constant has the value of empty string. This constant is useful when one needs to compose full name for local namespaced names.

Example #128 - Using __NAMESPACE__

```
<?php
namespace A::B::C;

function foo() {
// do stuff
}

set_error_handler(__NAMESPACE__ . "::foo");
?>
```

Name resolution rules

Names are resolved following these resolution rules:

- All qualified names are translated during compilation according to current import rules. In example, if the namespace A::B::C is imported, a call to `C::D::e()` is translated to `A::B::C::D::e()`.
- Unqualified class names are translated during compilation according to current import rules (full name substituted for short imported name). In example, if the namespace *A::B::C* is imported, `new C()` is translated to `new A::B::C()`.
- Inside namespace, calls to unqualified functions that are defined in the current namespace (and are known at the time the call is parsed) are interpreted as calls to these namespace functions, at compile time.
- Inside namespace (say A::B), calls to unqualified functions that are not defined in current namespace are resolved at run-time. Here is how a call to function *foo()* is resolved:
  - It looks for a function from the current namespace: *A::B::foo()*.
  - It tries to find and call the *internal* function *foo()*.

  To call a user defined function in the global namespace,*::foo()* has to be used.
- Inside namespace (say *A::B*), calls to unqualified class names are resolved at run-time. Here is how a call to `new C()` is resolved:
  - It looks for a class from the current namespace: *A::B::C*.
  - It tries to find and call the *internal* class *C*.
  - It attemts to autoload *A::B::C*.

  To reference a user defined class in the global namespace, `new ::C()` has to be used.

- Calls to qualified functions are resolved at run-time. Here is how a call to *A::B::foo()* is resolved:
  - It looks for a function *foo()* in the namespace *A::B*.
  - It looks for a class *A::B* and call its static method *foo()*. It will autoload the class if necessary.

- Qualified class names are resolved in compile-time as class from corresponding namespace. For example, `new A::B::C()` refers to class C from namespace *A::B*.

Example #129 - Name resolutions illustrated

```
<?php
namespace A;

// function calls

foo();      // first tries to call "foo" defined in namespace "A"
           // then calls internal function "foo"

::foo();    // calls function "foo" defined in global scope

// class references

new B();    // first tries to create object of class "B" defined in
namespace "A"
           // then creates object of internal class "B"

new ::B();  // creates object of class "B" defined in global scope

// static methods/namespace functions from another namespace

B::foo();   // first tries to call function "foo" from namespace "A::B"
           // then calls method "foo" of internal class "B"

::B::foo(); // first tries to call function "foo" from namespace "B"
           // then calls method "foo" of class "B" from global scope

// static methods/namespace functions of current namespace

A::foo();   // first tries to call function "foo" from namespace "A::A"
           // then tries to call method "foo" of class "A" from namespace
"A"
           // then tries to call function "foo" from namespace "A"
           // then calls method "foo" of internal class "A"

::A::foo(); // first tries to call function "foo" from namespace "A"
           // then calls method "foo" of class "A" from global scope
?>
```

Exceptions

PHP 5 has an exception model similar to that of other programming languages. An exception can be *throw* n, and caught (" *catch* ed") within PHP. Code may be surrounded in a *try* block, to facilitate the catching of potential exceptions. Each *try* must have at least one corresponding *catch* block. Multiple *catch* blocks can be used to catch different classes of exeptions. Normal execution (when no exception is thrown within the *try* block, or when a *catch* matching the thrown exception's class is not present) will continue after that last catch block defined in sequence. Exceptions can be *throw* n (or re-thrown) within a *catch* block.

When an exception is thrown, code following the statement will not be executed, and PHP will attempt to find the first matching *catch* block. If an exception is not caught, a PHP Fatal Error will be issued with an " *Uncaught Exception ...* " message, unless a handler has been defined with set_exception_handler().

Example #130 - Throwing an Exception

```
<?php
function inverse($x) {
   if (!$x) {
       throw new Exception('Division by zero.');
   }
   else return 1/$x;
}

try {
   echo inverse(5) . "\n";
   echo inverse(0) . "\n";
} catch (Exception $e) {
   echo 'Caught exception: ',  $e->getMessage(), "\n";
}

// Continue execution
echo 'Hello World';
?>
```

The above example will output:

```
0.2
Caught exception: Division by zero.
Hello World
```

Example #131 - Nested Exception

```
<?php

class MyException extends Exception { }

class Test {
   public function testing() {
```

```
        try {
            try {
                throw new MyException('foo!');
            } catch (MyException $e) {
                /* rethrow it */
                throw $e;
            }
        } catch (Exception $e) {
            var_dump($e->getMessage());
        }
    }
}

$foo = new Test;
$foo->testing();

?>
```

The above example will output:

```
string(4) "foo!"
```

Extending Exceptions

A User defined Exception class can be defined by extending the built-in Exception class. The members and properties below, show what is accessible within the child class that derives from the built-in Exception class.

Example #132 - The Built in Exception class

```
<?php
class Exception
{
   protected $message = 'Unknown exception';   // exception message
   protected $code = 0;                        // user defined exception
code
   protected $file;                            // source filename of
exception
   protected $line;                            // source line of exception

   function __construct($message = null, $code = 0);

   final function getMessage();                // message of exception
   final function getCode();                   // code of exception
   final function getFile();                   // source filename
   final function getLine();                   // source line
   final function getTrace();                  // an array of the
backtrace()
   final function getTraceAsString();          // formated string of trace

   /* Overrideable */
   function __toString();                       // formated string for
display
}
?>
```

If a class extends the built-in Exception class and re-defines the constructor, it is highly recomended that it also call parent::__construct() to ensure all available data has been properly assigned. The __toString() method can be overriden to provide a custom output when the object is presented as a string.

Example #133 - Extending the Exception class

```
<?php
/**
* Define a custom exception class
*/
class MyException extends Exception
{
   // Redefine the exception so message isn't optional
   public function __construct($message, $code = 0) {
       // some code

       // make sure everything is assigned properly
       parent::__construct($message, $code);
   }

   // custom string representation of object
   public function __toString() {
       return __CLASS__ . ": [{$this->code}]: {$this->message}\n";
   }

   public function customFunction() {
       echo "A Custom function for this type of exception\n";
   }
}


/**
* Create a class to test the exception
*/
class TestException
{
   public $var;

   const THROW_NONE    = 0;
   const THROW_CUSTOM  = 1;
   const THROW_DEFAULT = 2;

   function __construct($avalue = self::THROW_NONE) {

       switch ($avalue) {
           case self::THROW_CUSTOM:
               // throw custom exception
               throw new MyException('1 is an invalid parameter', 5);
               break;

           case self::THROW_DEFAULT:
               // throw default one.
               throw new Exception('2 isnt allowed as a parameter', 6);
               break;

           default:
               // No exception, object will be created.
```

```
                $this->var = $avalue;
                break;
        }
    }
}


// Example 1
try {
    $o = new TestException(TestException::THROW_CUSTOM);
} catch (MyException $e) {      // Will be caught
    echo "Caught my exception\n", $e;
    $e->customFunction();
} catch (Exception $e) {        // Skipped
    echo "Caught Default Exception\n", $e;
}

// Continue execution
var_dump($o);
echo "\n\n";


// Example 2
try {
    $o = new TestException(TestException::THROW_DEFAULT);
} catch (MyException $e) {      // Doesn't match this type
    echo "Caught my exception\n", $e;
    $e->customFunction();
} catch (Exception $e) {        // Will be caught
    echo "Caught Default Exception\n", $e;
}

// Continue execution
var_dump($o);
echo "\n\n";


// Example 3
try {
    $o = new TestException(TestException::THROW_CUSTOM);
} catch (Exception $e) {        // Will be caught
    echo "Default Exception caught\n", $e;
}

// Continue execution
var_dump($o);
echo "\n\n";


// Example 4
try {
    $o = new TestException();
} catch (Exception $e) {        // Skipped, no exception
    echo "Default Exception caught\n", $e;
}

// Continue execution
var_dump($o);
echo "\n\n";
?>
```

References Explained

What References Are

References in PHP are a means to access the same variable content by different names. They are not like C pointers; instead, they are symbol table aliases. Note that in PHP, variable name and variable content are different, so the same content can have different names. The most close analogy is with Unix filenames and files - variable names are directory entries, while variable contents is the file itself. References can be thought of as hardlinking in Unix filesystem.

What References Do

PHP references allow you to make two variables to refer to the same content. Meaning, when you do:

```
<?php
$a =& $b;
?>
```

it means that *$a* and *$b* point to the same content.

Note
$a and *$b* are completely equal here, that's not *$a* is pointing to *$b* or vice versa, that's *$a* and *$b* pointing to the same place.

Note
If array with references is copied, its values are not dereferenced. This is valid also for arrays passed by value to functions.

Note

If you assign, pass or return an undefined variable by reference, it will get created.

Example #134 - Using references with undefined variables

```
<?php
function foo(&$var) { }

foo($a); // $a is "created" and assigned to null
```

```
$b = array();
foo($b['b']);
var_dump(array_key_exists('b', $b)); // bool(true)

$c = new StdClass;
foo($c->d);
var_dump(property_exists($c, 'd')); // bool(true)
?>
```

The same syntax can be used with functions, that return references, and with *new* operator (in PHP 4.0.4 and later):

```
<?php
$bar =& new fooclass();
$foo =& find_var($bar);
?>
```

Since PHP 5, new return reference automatically so using =*&*in this context is deprecated and produces E_STRICT level message.

Note

Not using the *&*operator causes a copy of the object to be made. If you use *$this* in the class it will operate on the current instance of the class. The assignment without *&*will copy the instance (i.e. the object) and *$this* will operate on the copy, which is not always what is desired. Usually you want to have a single instance to work with, due to performance and memory consumption issues.

While you can use the @ operator to *mute* any errors in the constructor when using it as *@new*, this does not work when using the *&new* statement. This is a limitation of the Zend Engine and will therefore result in a parser error.

Warning

If you assign a reference to a variable declared *global* inside a function, the reference will be visible only inside the function. You can avoid this by using the *$GLOBALS* array.

Example #135 - Referencing global variables inside function

```
<?php
$var1 = "Example variable";
$var2 = "";

function global_references($use_globals)
{
   global $var1, $var2;
   if (!$use_globals) {
       $var2 =& $var1; // visible only inside the function
```

```
    } else {
        $GLOBALS["var2"] =& $var1; // visible also in global context
    }
}

global_references(false);
echo "var2 is set to '$var2'\n"; // var2 is set to ''
global_references(true);
echo "var2 is set to '$var2'\n"; // var2 is set to 'Example variable'
?>
```

Think about *global $var;* as a shortcut to *$var =& $GLOBALS['var'];*. Thus assigning other reference to *$var* only changes the local variable's reference.

Note

If you assign a value to a variable with references in a foreach statement, the references are modified too.

Example #136 - References and foreach statement

```
<?php
$ref = 0;
$row =& $ref;
foreach (array(1, 2, 3) as $row) {
   // do something
}
echo $ref; // 3 - last element of the iterated array
?>
```

The second thing references do is to pass variables by-reference. This is done by making a local variable in a function and a variable in the calling scope reference to the same content. Example:

```
<?php
function foo(&$var)
{
   $var++;
}

$a=5;
foo($a);
?>
```

will make *$a* to be 6. This happens because in the function *foo* the variable *$var* refers to the same content as *$a*. See also more detailed explanations about passing by reference.

The third thing reference can do is return by reference.

What References Are Not

As said before, references aren't pointers. That means, the following construct won't do what you expect:

```
<?php
function foo(&$var)
{
   $var =& $GLOBALS["baz"];
}
foo($bar);
?>
```

What happens is that *$var* in *foo* will be bound with *$bar* in caller, but then it will be re-bound with *$GLOBALS["baz"]*. There's no way to bind *$bar* in the calling scope to something else using the reference mechanism, since *$bar* is not available in the function *foo* (it is represented by *$var*, but *$var* has only variable contents and not name-to-value binding in the calling symbol table). You can use returning references to reference variables selected by the function.

Passing by Reference

You can pass variable to function by reference, so that function could modify its arguments. The syntax is as follows:

```
<?php
function foo(&$var)
{
   $var++;
}

$a=5;
foo($a);
// $a is 6 here
?>
```

Note that there's no reference sign on function call - only on function definition. Function definition alone is enough to correctly pass the argument by reference. In recent versions of PHP you will get a warning saying that "Call-time pass-by-reference" is deprecated when you use a & in *foo(&$a);*.

The following things can be passed by reference:

- Variable, i.e. *foo($a)*
- New statement, i.e. *foo(new foobar())*
- Reference, returned from a function, i.e.:

```
<?php
function &bar()
{
   $a = 5;
   return $a;
}
```

```
foo(bar());
?>
```

See also explanations about returning by reference.

Any other expression should not be passed by reference, as the result is undefined. For example, the following examples of passing by reference are invalid:

```
<?php
function bar() // Note the missing &
{
   $a = 5;
   return $a;
}
foo(bar()); // Produces fatal error since PHP 5.0.5

foo($a = 5); // Expression, not variable
foo(5); // Produces fatal error
?>
```

These requirements are for PHP 4.0.4 and later.

Returning References

Returning by-reference is useful when you want to use a function to find which variable a reference should be bound to. Do *not* use return-by-reference to increase performance, the engine is smart enough to optimize this on its own. Only return references when you have a valid technical reason to do it! To return references, use this syntax:

```
<?php
class foo {
   public $value = 42;

   public function &getValue() {
       return $this->value;
   }
}

$obj = new foo;
$myValue = &$obj->getValue(); // $myValue is a reference to $obj->value, which
is 42.
$obj->value = 2;
echo $myValue;                // prints the new value of $obj->value, i.e. 2.
?>
```

In this example, the property of the object returned by the *getValue* function would be set, not the copy, as it would be without using reference syntax.

Note
Unlike parameter passing, here you have to use *&*in both places - to indicate that you return by-reference, not a copy as usual, and to indicate that reference binding, rather

than usual assignment, should be done for *$myValue*.

Note

If you try to return a reference from a function with the syntax: *return ($this->value);* this will *not* work as you are attempting to return the result of an *expression*, and not a variable, by reference. You can only return variables by reference from a function - nothing else. **E_NOTICE** error is issued since PHP 4.4.0 and PHP 5.1.0 if the code tries to return a dynamic expression or a result of the *new* operator.

Unsetting References

When you unset the reference, you just break the binding between variable name and variable content. This does not mean that variable content will be destroyed. For example:

```
<?php
$a = 1;
$b =& $a;
unset($a);
?>
```

won't unset *$b*, just *$a*.

Again, it might be useful to think about this as analogous to Unix *unlink* call.

Spotting References

Many syntax constructs in PHP are implemented via referencing mechanisms, so everything told above about reference binding also apply to these constructs. Some constructs, like passing and returning by-reference, are mentioned above. Other constructs that use references are:

global References

When you declare variable as *global $var* you are in fact creating reference to a global variable. That means, this is the same as:

```
<?php
$var =& $GLOBALS["var"];
?>
```

That means, for example, that unsetting *$var* won't unset global variable.

$this

In an object method, *$this* is always a reference to the caller object.

Predefined variables

PHP provides a large number of predefined variables to all scripts. The variables represent everything from external variables to built-in environment variables, last error messages to last retrieved headers.

See also the FAQ titled " How does register_globals affect me? "

Superglobals

Superglobals -- Superglobals are built-in variables that are always available in all scopes

Description

Several predefined variables in PHP are "superglobals", which means they are available in all scopes throughout a script. There is no need to do *global $variable;* to access them within functions or methods.

These superglobal variables are:

- $GLOBALS
- $_SERVER
- $_GET
- $_POST
- $_FILES
- $_COOKIE
- $_SESSION
- $_REQUEST
- $_ENV

ChangeLog

Version	Description
4.1.0	Superglobals were introduced to PHP.

Notes

Note
Variable availability By default, all of the superglobals are available but there are directives that affect this availability. For further information, refer to the documentation for variables_order.

Note
Dealing with register_globals

If the deprecated register_globals directive is set to *on* then the variables within will also be made available in the global scope of the script. For example, *$_POST['foo']* would also exist as *$foo*.

For related information, see the FAQ titled " How does register_globals affect me? "

Note

Variable variables

Superglobals cannot be used as variable variables inside functions or class methods.

See Also

- variable scope
- The variables_order directive
- The filter extension

$GLOBALS

$GLOBALS -- References all variables available in global scope

Description

An associative array containing references to all variables which are currently defined in the global scope of the script. The variable names are the keys of the array.

Examples

Example #137 - *$GLOBALS* **example**

```
<?php
function test() {
   $foo = "local variable";

   echo '$foo in global scope: ' . $GLOBALS["foo"] . "\n";
   echo '$foo in current scope: ' . $foo . "\n";
}

$foo = "Example content";
test();
?>
```

The above example will output something similar to:

```
$foo in global scope: Example content
$foo in current scope: local variable
```

Notes

Note

This is a 'superglobal', or automatic global, variable. This simply means that it is available in all scopes throughout a script. There is no need to do *global $variable;* to access it within functions or methods.

Note

Variable availability

Unlike all of the other superglobals, *$GLOBALS* has essentially always been available in PHP.

$_SERVER

$HTTP_SERVER_VARS [deprecated]

$_SERVER -- $HTTP_SERVER_VARS [deprecated] -- Server and execution environment information

Description

$_SERVER is an array containing information such as headers, paths, and script locations. The entries in this array are created by the web server. There is no guarantee that every web server will provide any of these; servers may omit some, or provide others not listed here. That said, a large number of these variables are accounted for in the » CGI 1.1 specification, so you should be able to expect those.

$HTTP_SERVER_VARS contains the same initial information, but is not a superglobal. (Note that *$HTTP_SERVER_VARS* and *$_SERVER* are different variables and that PHP handles them as such)

You may or may not find any of the following elements in *$_SERVER*. Note that few, if any, of these will be available (or indeed have any meaning) if running PHP on the command line.

' *PHP_SELF* '
: The filename of the currently executing script, relative to the document root. For instance, *$_SERVER['PHP_SELF']* in a script at the address *http://example.com/test.php/foo.bar* would be */test.php/foo.bar*. The __FILE__ constant contains the full path and filename of the current (i.e. included) file. If PHP is running as a command-line processor this variable contains the script name since PHP 4.3.0. Previously it was not available.

' $argv '
: Array of arguments passed to the script. When the script is run on the command line, this gives C-style access to the command line parameters. When called via the GET method, this will contain the query string.

' $argc '
: Contains the number of command line parameters passed to the script (if run on the command line).

' *GATEWAY_INTERFACE* '
: What revision of the CGI specification the server is using; i.e. ' *CGI/1.1* '.

' *SERVER_ADDR* '
: The IP address of the server under which the current script is executing.

' *SERVER_NAME* '

The name of the server host under which the current script is executing. If the script is running on a virtual host, this will be the value defined for that virtual host.

' *SERVER_SOFTWARE* '
Server identification string, given in the headers when responding to requests.

' *SERVER_PROTOCOL* '
Name and revision of the information protocol via which the page was requested; i.e. ' *HTTP/1.0* ';

' *REQUEST_METHOD* '
Which request method was used to access the page; i.e. ' *GET* ', ' *HEAD* ', ' *POST* ', ' *PUT* '.

Note
PHP script is terminated after sending headers (it means after producing any output without output buffering) if the request method was *HEAD*.

' *REQUEST_TIME* '
The timestamp of the start of the request. Available since PHP 5.1.0.

' *QUERY_STRING* '
The query string, if any, via which the page was accessed.

' *DOCUMENT_ROOT* '
The document root directory under which the current script is executing, as defined in the server's configuration file.

' *HTTP_ACCEPT* '
Contents of the *Accept:* header from the current request, if there is one.

' *HTTP_ACCEPT_CHARSET* '
Contents of the *Accept-Charset:* header from the current request, if there is one. Example: ' *iso-8859-1,*,utf-8* '.

' *HTTP_ACCEPT_ENCODING* '
Contents of the *Accept-Encoding:* header from the current request, if there is one. Example: ' *gzip* '.

' *HTTP_ACCEPT_LANGUAGE* '
Contents of the *Accept-Language:* header from the current request, if there is one. Example: ' *en* '.

' *HTTP_CONNECTION* '
Contents of the *Connection:* header from the current request, if there is one. Example: ' *Keep-Alive* '.

' *HTTP_HOST* '
Contents of the *Host:* header from the current request, if there is one.

' *HTTP_REFERER* '
The address of the page (if any) which referred the user agent to the current page. This is set by the user agent. Not all user agents will set this, and some provide the ability to modify *HTTP_REFERER* as a feature. In short, it cannot really be trusted.

' *HTTP_USER_AGENT* '
Contents of the *User-Agent:* header from the current request, if there is one. This is a string denoting the user agent being which is accessing the page. A typical example is: Mozilla/4.5 [en] (X11; U; Linux 2.2.9 i586). Among other things, you can use this value with get_browser() to tailor your page's output to the capabilities of the user agent.

' *HTTPS* '
Set to a non-empty value if the script was queried through the HTTPS protocol. Note that when using ISAPI with IIS, the value will be *off* if the request was not made through the HTTPS protocol.

' *REMOTE_ADDR* '
The IP address from which the user is viewing the current page.

' *REMOTE_HOST* '
The Host name from which the user is viewing the current page. The reverse dns lookup is based off the *REMOTE_ADDR* of the user.

Note
Your web server must be configured to create this variable. For example in Apache you'll need *HostnameLookups On* inside *httpd.conf* for it to exist. See also gethostbyaddr().

' *REMOTE_PORT* '
The port being used on the user's machine to communicate with the web server.

' *SCRIPT_FILENAME* '
The absolute pathname of the currently executing script.

Note
If a script is executed with the CLI, as a relative path, such as *file.php* or*../file.php*, *$_SERVER['SCRIPT_FILENAME']* will contain the relative path specified by the user.

' *SERVER_ADMIN* '
The value given to the SERVER_ADMIN (for Apache) directive in the web server configuration file. If the script is running on a virtual host, this will be the value defined for that virtual host.

' *SERVER_PORT* '
The port on the server machine being used by the web server for communication. For

default setups, this will be ' *80* '; using SSL, for instance, will change this to whatever your defined secure HTTP port is.

' *SERVER_SIGNATURE* '
String containing the server version and virtual host name which are added to server-generated pages, if enabled.

' *PATH_TRANSLATED* '
Filesystem- (not document root-) based path to the current script, after the server has done any virtual-to-real mapping.

Note
As of PHP 4.3.2, PATH_TRANSLATED is no longer set implicitly under the Apache 2 SAPI in contrast to the situation in Apache 1, where it's set to the same value as the SCRIPT_FILENAME server variable when it's not populated by Apache. This change was made to comply with the CGI specification that PATH_TRANSLATED should only exist if PATH_INFO is defined. Apache 2 users may use *AcceptPathInfo = On* inside *httpd.conf* to define PATH_INFO.

' *SCRIPT_NAME* '
Contains the current script's path. This is useful for pages which need to point to themselves. The __FILE__ constant contains the full path and filename of the current (i.e. included) file.

' *REQUEST_URI* '
The URI which was given in order to access this page; for instance, ' */index.html* '.

' *PHP_AUTH_DIGEST* '
When running under Apache as module doing Digest HTTP authentication this variable is set to the 'Authorization' header sent by the client (which you should then use to make the appropriate validation).

' *PHP_AUTH_USER* '
When running under Apache or IIS (ISAPI on PHP 5) as module doing HTTP authentication this variable is set to the username provided by the user.

' *PHP_AUTH_PW* '
When running under Apache or IIS (ISAPI on PHP 5) as module doing HTTP authentication this variable is set to the password provided by the user.

' *AUTH_TYPE* '
When running under Apache as module doing HTTP authenticated this variable is set to the authentication type.

ChangeLog

Version	Description
4.1.0	Introduced *$_SERVER* that the deprecated *$HTTP_SERVER_VARS*.

Examples

Example #138 - *$_SERVER* example

```
<?php
echo $_SERVER['SERVER_NAME'];
?>
```

The above example will output something similar to:

```
www.example.com
```

Notes

Note

This is a 'superglobal', or automatic global, variable. This simply means that it is available in all scopes throughout a script. There is no need to do *global $variable;* to access it within functions or methods.

See Also

- The filter extension

$_GET

$HTTP_GET_VARS [deprecated]

$_GET -- $HTTP_GET_VARS [deprecated] -- HTTP GET variables

Description

An associative array of variables passed to the current script via the HTTP GET method.

$HTTP_GET_VARS contains the same initial information, but is not a superglobal. (Note that *$HTTP_GET_VARS* and *$_GET* are different variables and that PHP handles them as such)

ChangeLog

Version	Description
4.1.0	Introduced *$_GET* that deprecated *$HTTP_GET_VARS*.

Examples

Example #139 - *$_GET* example

```
<?php
echo 'Hello ' . htmlspecialchars($_GET["name"]) . '!';
?>
```

Assuming the user entered http://example.com/?name=Hannes

The above example will output something similar to:

```
Hello Hannes!
```

Notes

Note
This is a 'superglobal', or automatic global, variable. This simply means that it is available in all scopes throughout a script. There is no need to do *global $variable;* to access it within functions or methods.

See Also

- Handling external variables
- The filter extension

$_POST

$HTTP_POST_VARS [deprecated]

$_POST -- $HTTP_POST_VARS [deprecated] -- HTTP POST variables

Description

An associative array of variables passed to the current script via the HTTP POST method.

$HTTP_POST_VARS contains the same initial information, but is not a superglobal. (Note that *$HTTP_POST_VARS* and *$_POST* are different variables and that PHP handles them as such)

ChangeLog

Version	Description
4.1.0	Introduced *$_POST* that deprecated *$HTTP_POST_VARS*.

Examples

Example #140 - *$_POST* example

```
<?php
echo 'Hello ' . htmlspecialchars($_POST["name"]) . '!';
?>
```

Assuming the user POSTed name=Hannes

The above example will output something similar to:

```
Hello Hannes!
```

Notes

Note
This is a 'superglobal', or automatic global, variable. This simply means that it is available in all scopes throughout a script. There is no need to do *global $variable;* to access it within functions or methods.

See Also

- Handling external variables
- The filter extension

$_FILES

$HTTP_POST_FILES [deprecated]

$_FILES -- $HTTP_POST_FILES [deprecated] -- HTTP File Upload variables

Description

An associative array of items uploaded to the current script via the HTTP POST method.

$HTTP_POST_FILES contains the same initial information, but is not a superglobal. (Note that *$HTTP_POST_FILES* and *$_FILES* are different variables and that PHP handles them as such)

ChangeLog

Version	Description
4.1.0	Introduced *$_FILES* that deprecated *$HTTP_POST_FILES*.

Notes

Note
This is a 'superglobal', or automatic global, variable. This simply means that it is available in all scopes throughout a script. There is no need to do *global $variable;* to access it within functions or methods.

See Also

- move_uploaded_file()
- Handling File Uploads

$_REQUEST

$_REQUEST -- HTTP Request variables

Description

An associative array that by default contains the contents of $_GET, $_POST and $_COOKIE.

ChangeLog

Version	Description
5.3.0	Introduced request_order. This directive affects the contents of *$_REQUEST*.
4.3.0	$_FILES information was removed from *$_REQUEST*.
4.1.0	Introduced *$_REQUEST*.

Notes

Note
This is a 'superglobal', or automatic global, variable. This simply means that it is available in all scopes throughout a script. There is no need to do *global $variable;* to access it within functions or methods.

Note
When running on the command line, this will *not* include the argv and argc entries; these are present in the $_SERVER array.

Note
Variables provided to the script via the GET, POST, and COOKIE input mechanisms, and which therefore cannot be trusted. The presence and order of variable inclusion in this array is defined according to the PHP variables_order configuration directive.

See Also

- import_request_variables()
- Handling external variables
- The filter extension

$_SESSION

$HTTP_SESSION_VARS [deprecated]

$_SESSION -- $HTTP_SESSION_VARS [deprecated] -- Session variables

Description

An associative array containing session variables available to the current script. See the Session functions documentation for more information on how this is used.

$HTTP_SESSION_VARS contains the same initial information, but is not a superglobal. (Note that *$HTTP_SESSION_VARS* and *$_SESSION* are different variables and that PHP handles them as such)

ChangeLog

Version	Description
4.1.0	Introduced *$_SESSION* that the deprecated *$HTTP_SESSION_VARS*.

Notes

Note
This is a 'superglobal', or automatic global, variable. This simply means that it is available in all scopes throughout a script. There is no need to do *global $variable;* to access it within functions or methods.

See Also

- session_start()

$_ENV

$HTTP_ENV_VARS [deprecated]

$_ENV -- $HTTP_ENV_VARS [deprecated] -- Environment variables

Description

An associative array of variables passed to the current script via the environment method.

These variables are imported into PHP's global namespace from the environment under which the PHP parser is running. Many are provided by the shell under which PHP is running and different systems are likely running different kinds of shells, a definitive list is impossible. Please see your shell's documentation for a list of defined environment variables.

Other environment variables include the CGI variables, placed there regardless of whether PHP is running as a server module or CGI processor.

$HTTP_ENV_VARS contains the same initial information, but is not a superglobal. (Note that *$HTTP_ENV_VARS* and *$_ENV* are different variables and that PHP handles them as such)

ChangeLog

Version	Description
4.1.0	Introduced *$_ENV* that deprecated *$HTTP_ENV_VARS*.

Examples

Example #141 - *$_ENV* example

```
<?php
echo 'My username is ' .$_ENV["USER"] . '!';
?>
```

Assuming "bjori" executes this script

The above example will output something similar to:

```
My username is bjori!
```

Notes

Note
This is a 'superglobal', or automatic global, variable. This simply means that it is available in all scopes throughout a script. There is no need to do *global $variable;* to access it within functions or methods.

See Also

- getenv()
- The filter extension

$_COOKIE

$HTTP_COOKIE_VARS [deprecated]

$_COOKIE -- $HTTP_COOKIE_VARS [deprecated] -- HTTP Cookies

Description

An associative array of variables passed to the current script via HTTP Cookies.

$HTTP_COOKIE_VARS contains the same initial information, but is not a superglobal. (Note that *$HTTP_COOKIE_VARS* and *$_COOKIE* are different variables and that PHP handles them as such)

ChangeLog

Version	Description
4.1.0	Introduced *$_COOKIE* that deprecated *$HTTP_COOKIE_VARS*.

Examples

Example #142 - *$_COOKIE* example

```
<?php
echo 'Hello ' . htmlspecialchars($_COOKIE["name"]) . '!';
?>
```

Assuming the "name" cookie has been set earlier

The above example will output something similar to:

```
Hello Hannes!
```

Notes

Note
This is a 'superglobal', or automatic global, variable. This simply means that it is available in all scopes throughout a script. There is no need to do *global $variable;* to access it within functions or methods.

See Also

- setcookie()
- Handling external variables
- The filter extension

$php_errormsg

$php_errormsg -- The previous error message

Description

$php_errormsg is a variable containing the text of the last error message generated by PHP. This variable will only be available within the scope in which the error occurred, and only if the track_errors configuration option is turned on (it defaults to off).

Note
This variable is only available when *track_errors* is enabled in *php.ini*.

Warning
If a user defined error handler is set *$php_erromsg* is only set if the error handler returns **FALSE**

Examples

Example #143 - *$php_errormsg* **example**

```
<?php
@strpos();
echo $php_errormsg;
?>
```

The above example will output something similar to:

```
Wrong parameter count for strpos()
```

$HTTP_RAW_POST_DATA

$HTTP_RAW_POST_DATA -- Raw POST data

Description

$HTTP_RAW_POST_DATA contains the raw POST data. See always_populate_raw_post_data

$http_response_header

$http_response_header -- HTTP response headers

Description

The *$http_response_header* array is similar to the get_headers() function. When using the HTTP wrapper, *$http_response_header* will be populated with the HTTP response headers.

Examples

Example #144 - *$http_response_header* **example**

```
<?php
file_get_contents("http://example.com");
var_dump($http_response_header);
?>
```

The above example will output something similar to:

```
array(9) {
 [0]=>
 string(15) "HTTP/1.1 200 OK"
 [1]=>
 string(35) "Date: Sat, 12 Apr 2008 17:30:38 GMT"
 [2]=>
 string(29) "Server: Apache/2.2.3 (CentOS)"
 [3]=>
 string(44) "Last-Modified: Tue, 15 Nov 2005 13:24:10 GMT"
 [4]=>
 string(27) "ETag: "280100-1b6-80bfd280""
 [5]=>
 string(20) "Accept-Ranges: bytes"
 [6]=>
 string(19) "Content-Length: 438"
 [7]=>
 string(17) "Connection: close"
 [8]=>
 string(38) "Content-Type: text/html; charset=UTF-8"
}
```

$argc

$argc -- The number of arguments passed to script

Description

Contains the number of arguments passed to the current script when running from the command line.

Note
The script's filename is always passed as an argument to the script, therefore the minimum value of *$argc* is *1*.

Note
This variable is only available when register_argc_argv is enabled.

Examples

Example #145 - *$argc* **example**

```
<?php
var_dump($argc);
?>
```

When executing the example with: php script.php arg1 arg2 arg3

The above example will output something similar to:

```
int(4)
```

$argv

$argv -- Array of arguments passed to script

Description

Contains an array of all the arguments passed to the script when running from the command line.

Note
The first argument is always the current script's filename, therefore *$argv[0]* is the script's name.

Note
This variable is only available when register_argc_argv is enabled.

Examples

Example #146 - *$argv* **example**

```
<?php
var_dump($argv);
?>
```

When executing the example with: php script.php arg1 arg2 arg3

The above example will output something similar to:

```
array(4) {
 [0]=>
 string(10) "script.php"
 [1]=>
 string(4) "arg1"
 [2]=>
 string(4) "arg2"
 [3]=>
 string(4) "arg3"
}
```

Predefined Exceptions

Exception

Introduction

Exception is the base class for all Exceptions.

Class synopsis

Exception

```
Exception {
    /*  Properties */
    protected string message;
    private string string;
    protected int code;
    protected string file;
    protected int line;
    private array trace;
    /*  Methods */
    public Exception::__construct ( [ string $message [, int $code ] ] )
    final public string Exception::getMessage ( void )
    final public int Exception::getCode ( void )
    final public string Exception::getFile ( void )
    final public string Exception::getLine ( void )
    final public array Exception::getTrace ( void )
    final public string Exception::getTraceAsString ( void )
    public string Exception::__toString ( void )
```

```
    final private string Exception::__clone ( void )
}
```

Properties

message
: The exception message

string
: Internal Exception name

code
: The Exception code

file
: The filename where the exception was thrown

line
: The line where the exception was thrown

trace
: The stack trace

Exception::__construct

Exception::__construct -- Construct the exception

Description

public **Exception::__construct** ([string `$message` [, int `$code`]])

Constructs the Exception.

Parameters

`message`
The Exception message to throw.

`code`
The Exception code.

Exception::getMessage

Exception::getMessage -- Gets the Exception message

Description

final public string **Exception::getMessage** (void)

Returns the Exception message.

Parameters

This function has no parameters.

Return Values

Returns the Exception message as a string.

Examples

Example #147 - Exception::getMessage() example

```
<?php
try {
   throw new Exception("Some error message");
} catch(Exception $e) {
   echo $e->getMessage();
}
?>
```

The above example will output something similar to:

```
Some error message
```

Exception::getCode

Exception::getCode -- Gets the Exception code

Description

final public int **Exception::getCode** (void)

Returns the Exception code.

Parameters

This function has no parameters.

Return Values

Returns the Exception code as a integer.

Examples

Example #148 - Exception::getCode() example

```
<?php
try {
   throw new Exception("Some error message", 30);
} catch(Exception $e) {
   echo "The exception code is: " . $e->getCode();
}
?>
```

The above example will output something similar to:

```
The exception code is: 30
```

Exception::getFile

Exception::getFile -- Gets the file in which the exception occurred

Description

final public string **Exception::getFile** (void)

Get the name of the file the exception was thrown from.

Parameters

This function has no parameters.

Return Values

Returns the filename in which the exception was thrown.

Examples

Example #149 - Exception::getFile() example

```
<?php
try {
   throw new Exception;
} catch(Exception $e) {
   echo $e->getFile();
}
?>
```

The above example will output something similar to:

```
/home/bjori/tmp/ex.php
```

Exception::getLine

Exception::getLine -- Gets the line in which the exception occurred

Description

final public string **Exception::getLine** (void)

Returns line number where the exception was thrown.

Parameters

This function has no parameters.

Return Values

Returns the line number where the exception was thrown.

Examples

Example #150 - Exception::getLine() example

```
<?php
try {
   throw new Exception("Some error message");
} catch(Exception $e) {
   echo "The exception was thrown on line: " . $e->getLine();
}
?>
```

The above example will output something similar to:

```
The exception was thrown on line: 3
```

Exception::getTrace

Exception::getTrace -- Gets the stack trace

Description

final public array **Exception::getTrace** (void)

Returns the Exception stack trace.

Parameters

This function has no parameters.

Return Values

Returns the Exception stack trace as an array.

Examples

Example #151 - Exception::getTrace() example

```
<?php
function test() {
throw new Exception;
}

try {
test();
} catch(Exception $e) {
var_dump($e->getTrace());
}
?>
```

The above example will output something similar to:

```
array(1) {
 [0]=>
 array(4) {
   ["file"]=>
   string(22) "/home/bjori/tmp/ex.php"
   ["line"]=>
   int(7)
   ["function"]=>
   string(4) "test"
   ["args"]=>
   array(0) {
   }
 }
}
```

Exception::getTraceAsString

Exception::getTraceAsString -- Gets the stack trace as a string

Description

final public string **Exception::getTraceAsString** (void)

Returns the Exception stack trace as a string.

Parameters

This function has no parameters.

Return Values

Returns the Exception stack trace as a string.

Examples

Example #152 - Exception::getTraceAsString() **example**

```
<?php
function test() {
   throw new Exception;
}

try {
   test();
} catch(Exception $e) {
   echo $e->getTraceAsString();
}
?>
```

The above example will output something similar to:

```
#0 /home/bjori/tmp/ex.php(7): test()
#1 {main}
```

Exception::__toString

Exception::__toString -- String representation of the exception

Description

public string **Exception::__toString** (void)

Returns the string representation of the exception.

Parameters

This function has no parameters.

Return Values

Returns the string representation of the exception.

Examples

Example #153 - Exception::__toString() example

```
<?php
try {
   throw new Exception("Some error message");
} catch(Exception $e) {
   echo $e;
}
?>
```

The above example will output something similar to:

```
exception 'Exception' with message 'Some error message' in
/home/bjori/tmp/ex.php:3
Stack trace:
#0 {main}
```

Exception::__clone

Exception::__clone -- Clone the exception

Description

final private string **Exception::__clone** (void)

Tries to clone the Exception, which results in Fatal error.

Parameters

This function has no parameters.

Return Values

No value is returned.

Errors/Exceptions

Exceptions are *not* clonable.

ErrorException

Introduction

An Error Exception.

Class synopsis

ErrorException

ErrorException extends Exception {

/* Properties */

protected int *severity*;

/* Methods */

public **ErrorException::__construct** ([string `$message` [, int `$code` [, int `$severity` [, string `$filename` [, int `$lineno`]]]]])

final public int **ErrorException::getSeverity** (void)

/* Inherited methods */

final public string **Exception::getMessage** (void)

final public int **Exception::getCode** (void)

final public string **Exception::getFile** (void)

final public string **Exception::getLine** (void)

final public array **Exception::getTrace** (void)

final public string **Exception::getTraceAsString** (void)

public string **Exception::__toString** (void)

final private string **Exception::__clone** (void)

}

Properties

severity
The severity of the exception

Examples

Example #154 - Turn all error messages into ErrorException.

```
<?php
function exception_error_handler($errno, $errstr, $errfile, $errline ) {
throw new ErrorException($errstr, 0, $errno, $errfile, $errline);
}
set_error_handler("exception_error_handler");

/* Trigger exception */
strpos();
?>
```

The above example will output something similar to:

```
Fatal error: Uncaught exception 'ErrorException' with message 'Wrong
parameter count for strpos()' in /home/bjori/tmp/ex.php:8
Stack trace:
#0 [internal function]: exception_error_handler(2, 'Wrong parameter...',
'/home/bjori/php...', 8, Array)
#1 /home/bjori/php/cleandocs/test.php(8): strpos()
#2 {main}
 thrown in /home/bjori/tmp/ex.php on line 8
```

ErrorException::__construct

ErrorException::__construct -- Construct the exception

Description

public **ErrorException::__construct** ([string `$message` [, int `$code` [, int `$severity` [, string `$filename` [, int `$lineno`]]]]])

Constructs the Exception.

Parameters

`message`
: The Exception message to throw.

`code`
: The Exception code.

`severity`
: The severity level of the exception.

`filename`
: The filename where the exception is thrown.

`lineno`
: The line number where the exception is thrown.

ErrorException::getSeverity

ErrorException::getSeverity -- Gets the exception severity

Description

final public int **ErrorException::getSeverity** (void)

Returns the severity of the exception.

Parameters

This function has no parameters.

Return Values

Returns the severity level of the exception.

Examples

Example #155 - ErrorException() example

```
<?php
try {
   throw new ErrorException("Exception message", 0, 75);
} catch(ErrorException $e) {
   echo "This exception severity is: " . $e->getSeverity();
}
?>
```

The above example will output something similar to:

```
This exception severity is: 75
```

Context options and parameters

PHP offers various context options and parameters which can be used with all filesystem and stream wrappers. The context is created with stream_context_create(). Options are set with stream_context_set_option() and parameters with stream_context_set_params().

Socket context options

Socket context options -- Socket context option listing

Description

Socket context options are available for all wrappers that work over sockets, like *tcp*, *http* and *ftp*.

Options

`bindto`
: Used to specify the IP address (either IPv4 or IPv6) and/or the port number that PHP will use to access the network. The syntax is *ip:port*. Setting the IP or the port to *0* will let the system choose the IP and/or port.

Note
As FTP creates two socket connections during normal operation, the port number cannot be specified using this option.

ChangeLog

Version	Description
5.1.0	Added *bindto*.

Examples

Example #156 - Basic *`bindto`* usage example

```
<?php
// connect to the internet using the '192.168.0.100' IP
$opts = array(
   'socket' => array(
       'bindto' => '192.168.0.100:0',
   ),
);
```

```
// connect to the internet using the '192.168.0.100' IP and port '7000'
$opts = array(
   'socket' => array(
       'bindto' => '192.168.0.100:7000',
   ),
);


// connect to the internet using port '7000'
$opts = array(
   'socket' => array(
       'bindto' => '0:7000',
   ),
);


// create the context...
$context = stream_context_create($opts);

// ...and use it to fetch the data
echo file_get_contents('http://www.example.com', false, $context);

?>
```

HTTP context options

HTTP context options -- HTTP context option listing

Description

Context options for *http://* and *https://* transports.

Options

`method` string
: **GET**, **POST**, or any other HTTP method supported by the remote server. Defaults to **GET**.

`header` string
: Additional headers to be sent during request. Values in this option will override other values (such as *User-agent:*, *Host:*, and *Authentication:*).

`user_agent` string
: Value to send with User-Agent: header. This value will only be used if user-agent is *not* specified in the *header* context option above. By default the user_agent *php.ini* setting is used.

`content` string
: Additional data to be sent after the headers. Typically used with POST or PUT requests.

`proxy` string
: URI specifying address of proxy server. (e.g. *tcp://proxy.example.com:5100*).

`request_fulluri` boolean
: When set to **TRUE**, the entire URI will be used when constructing the request. (i.e. *GET http://www.example.com/path/to/file.html HTTP/1.0*). While this is a non-standard request format, some proxy servers require it. Defaults to **FALSE**.

`max_redirects` integer
: The max number of redirects to follow. Value *1* or less means that no redirects are followed. Defaults to *20*.

`protocol_version` float
: HTTP protocol version. Defaults to *1.0*.

`timeout` float
: Read timeout in seconds, specified by a float (e.g. *10.5*). By default the default_socket_timeout *php.ini* setting is used.

`ignore_errors` boolean
: Fetch the content even on failure status codes. Defaults to **FALSE**

ChangeLog

Version	Description
5.3.0	Added *`ignore_errors`*.
5.2.1	Added *`timeout`*.
5.1.0	Added HTTPS proxying through HTTP proxies.
5.1.0	Added *`protocol_version`*.

5.1.0

Examples

Example #157 - Fetch a page and send POST data

```
<?php

$postdata = http_build_query(
   array(
       'var1' => 'some content',
       'var2' => 'doh'
   )
);

$opts = array('http' =>
   array(
       'method'  => 'POST',
       'header'  => 'Content-type: application/x-www-form-urlencoded',
       'content' => $postdata
   )
);

$context  = stream_context_create($opts);

$result = file_get_contents('http://example.com/submit.php', false,
$context);

?>
```

Notes

Note
Underlying socket stream context options Additional context options may be supported by the underlying transport For *http://* streams, refer to context options for the *tcp://* transport. For *https://* streams, refer to context options for the *ssl://* transport.

See Also

- Socket context options
- SSL context options

FTP context options

FTP context options -- FTP context option listing

Description

Context options for *ftp://* and *ftps://* transports.

Options

`overwrite` boolean
: Allow overwriting of already existing files on remote server. Applies to write mode (uploading) only. Defaults to **FALSE**.

`resume_pos` integer
: File offset at which to begin transfer. Applies to read mode (downloading) only. Defaults to *0* (Beginning of File).

`proxy` string
: Proxy FTP request via http proxy server. Applies to file read operations only. Ex: *tcp://squid.example.com:8000*.

ChangeLog

Version	Description
5.1.0	Added `proxy`.
5.0.0	Added `overwrite` and `resume_pos`.

Notes

Note
Underlying socket stream context options Additional context options may be supported by the underlying transport For *ftp://* streams, refer to context options for the *tcp://* transport. For *ftps://* streams, refer to context options for the *ssl://* transport.

See Also

- Socket context options
- SSL context options

SSL context options

SSL context options -- SSL context option listing

Description

Context options for *ssl://* and *tls://* transports.

Options

`verify_peer` boolean
: Require verification of SSL certificate used. Defaults to **FALSE**.

`allow_self_signed` boolean
: Allow self-signed certificates. Defaults to **FALSE**

`cafile` string
: Location of Certificate Authority file on local filesystem which should be used with the *verify_peer* context option to authenticate the identity of the remote peer.

`capath` string
: If *cafile* is not specified or if the certificate is not found there, the directory pointed to by *capath* is searched for a suitable certificate. *capath* must be a correctly hashed certificate directory.

`local_cert` string
: Path to local certificate file on filesystem. It must be a PEM encoded file which contains your certificate and private key. It can optionally contain the certificate chain of issuers.

`passphrase` string
: Passphrase with which your *local_cert* file was encoded.

`CN_match` string
: Common Name we are expecting. PHP will perform limited wildcard matching. If the Common Name does not match this, the connection attempt will fail.

`verify_depth` integer
: Abort if the certificate chain is too deep. Defaults to no verification.

`ciphers` string
: Sets the list of available ciphers. The format of the string is described in » ciphers(1). Defaults to *DEFAULT*.

`capture_peer_cert` boolean
: If set to **TRUE** a *peer_certificate* context option will be created containing the peer certificate.

`capture_peer_chain` boolean

If set to **TRUE** a *peer_certificate_chain* context option will be created containing the certificate chain.

ChangeLog

Version	Description
5.0.0	Added `capture_peer_cert`, `capture_peer_chain` and `ciphers`.

Notes

Note
Because *ssl://* is the underlying transport for the *https://* and *ftps://* wrappers, any context options which apply to *ssl://* also apply to *https://* and *ftps://*.

See Also

- Socket context options

CURL context options

CURL context options -- CURL context option listing

Description

CURL context options are available when the CURL extension was compiled using the *--with-curlwrappers* configure option.

Options

`method` string
: **GET**, **POST**, or any other HTTP method supported by the remote server. Defaults to **GET**.

`header` string
: Additional headers to be sent during request. Values in this option will override other values (such as *User-agent:*, *Host:*, and *Authentication:*).

`user_agent` string
: Value to send with User-Agent: header. By default the user_agent *php.ini* setting is used.

`content` string
: Additional data to be sent after the headers. This option is not used for **GET** or **HEAD** requests.

`proxy` string
: URI specifying address of proxy server. (e.g. *tcp://proxy.example.com:5100*).

`max_redirects` integer
: The max number of redirects to follow. Value *1* or less means that no redirects are followed. Defaults to *20*.

`curl_verify_ssl_host` boolean
: Verify the host. Defaults to **FALSE**

Note
This option is available for both the http and ftp protocol wrappers.

`curl_verify_ssl_peer` boolean
: Require verification of SSL certificate used. Defaults to **FALSE**

Note
This option is available for both the http and ftp protocol wrappers.

Examples

Example #158 - Fetch a page and send POST data

```
<?php

$postdata = http_build_query(
   array(
       'var1' => 'some content',
       'var2' => 'doh'
   )
);

$opts = array('http' =>
   array(
       'method'  => 'POST',
       'header'  => 'Content-type: application/x-www-form-urlencoded',
       'content' => $postdata
   )
);

$context  = stream_context_create($opts);

$result = file_get_contents('http://example.com/submit.php', false,
$context);

?>
```

See Also

- Socket context options

Context parameters

Context parameters -- Context parameter listing

Description

These *parameters* can be set on a *context* using the stream_context_set_params() function.

Options

`notification` callback
: A callback to be called when an event occurs on a stream. See stream_notification_callback() for more details.