# Types

* [Booleans](http://www.php.net/manual/en/language.types.boolean.php)
* [Integers](http://www.php.net/manual/en/language.types.integer.php)
* [Floating point numbers](http://www.php.net/manual/en/language.types.float.php)
* [Strings](http://www.php.net/manual/en/language.types.string.php)
* [Arrays](http://www.php.net/manual/en/language.types.array.php)
* [Objects](http://www.php.net/manual/en/language.types.object.php)
* [Resources](http://www.php.net/manual/en/language.types.resource.php)
* [NULL](http://www.php.net/manual/en/language.types.null.php)
* [Callbacks](http://www.php.net/manual/en/language.types.callable.php)
* [Pseudo-types and variables used in this documentation](http://www.php.net/manual/en/language.pseudo-types.php)
* [Type Juggling](http://www.php.net/manual/en/language.types.type-juggling.php)

Four scalar types:

* [boolean](http://www.php.net/manual/en/language.types.boolean.php)
* [integer](http://www.php.net/manual/en/language.types.integer.php)
* [float](http://www.php.net/manual/en/language.types.float.php) (floating-point number, aka [double](http://www.php.net/manual/en/language.types.float.php))
* [string](http://www.php.net/manual/en/language.types.string.php)

Two compound types:

* [array](http://www.php.net/manual/en/language.types.array.php)
* [object](http://www.php.net/manual/en/language.types.object.php)

And finally two special types:

* [resource](http://www.php.net/manual/en/language.types.resource.php)
* [NULL](http://www.php.net/manual/en/language.types.null.php)

This manual also introduces some [pseudo-types](http://www.php.net/manual/en/language.pseudo-types.php) for readability reasons:

* [mixed](http://www.php.net/manual/en/language.pseudo-types.php#language.types.mixed)
* [number](http://www.php.net/manual/en/language.pseudo-types.php#language.types.number)
* [callback](http://www.php.net/manual/en/language.pseudo-types.php#language.types.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](http://www.php.net/manual/en/language.expressions.php), use the [var\_dump()](http://www.php.net/manual/en/function.var-dump.php) function.

To get a human-readable representation of a type for debugging, use the [gettype()](http://www.php.net/manual/en/function.gettype.php) function. To check for a certain type, do not use [gettype()](http://www.php.net/manual/en/function.gettype.php), 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 $a\_bool is a string, print it out  
// (does not print out anything)  
if (is\_string($a\_bool)) {  
    echo "String: $a\_bool";  
}  
?>

To forcibly convert a variable to a certain type, either [cast](http://www.php.net/manual/en/language.types.type-juggling.php#language.types.typecasting) the variable or use the [settype()](http://www.php.net/manual/en/function.settype.php) 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](http://www.php.net/manual/en/language.types.type-juggling.php). [The type comparison tables](http://www.php.net/manual/en/types.comparisons.php) may also be useful, as they show examples of various type-related comparisons.

## Booleans

This is the simplest type. A [boolean](http://www.php.net/manual/en/language.types.boolean.php) expresses a truth value. It can be either **TRUE** or **FALSE**.

### Syntax

To specify a [boolean](http://www.php.net/manual/en/language.types.boolean.php) literal, use the constants **TRUE** or **FALSE**. Both are case-insensitive.

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

Typically, the result of an [operator](http://www.php.net/manual/en/language.operators.php) which returns a [boolean](http://www.php.net/manual/en/language.types.boolean.php) value is passed on to a [control structure](http://www.php.net/manual/en/language.control-structures.php).

<?php  
// == is an operator which tests  
// 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 this can be used with exactly the same meaning:  
if ($show\_separators) {  
    echo "<hr>\n";  
}  
?>

### Converting to boolean

To explicitly convert a value to [boolean](http://www.php.net/manual/en/language.types.boolean.php), use the (bool) or (boolean) casts. However, in most cases the cast is unnecessary, since a value will be automatically converted if an operator, function or control structure requires a [boolean](http://www.php.net/manual/en/language.types.boolean.php) argument.

See also [Type Juggling](http://www.php.net/manual/en/language.types.type-juggling.php).

When converting to [boolean](http://www.php.net/manual/en/language.types.boolean.php), the following values are considered **FALSE**:

* the [boolean](http://www.php.net/manual/en/language.types.boolean.php) **FALSE** itself
* the [integer](http://www.php.net/manual/en/language.types.integer.php) 0 (zero)
* the [float](http://www.php.net/manual/en/language.types.float.php) 0.0 (zero)
* the empty [string](http://www.php.net/manual/en/language.types.string.php), and the [string](http://www.php.net/manual/en/language.types.string.php) "0"
* an [array](http://www.php.net/manual/en/language.types.array.php) with zero elements
* an [object](http://www.php.net/manual/en/language.types.object.php) with zero member variables (PHP 4 only)
* the special type [NULL](http://www.php.net/manual/en/language.types.null.php) (including unset variables)
* [SimpleXML](http://www.php.net/manual/en/ref.simplexml.php) objects created from empty tags

Every other value is considered **TRUE** (including any [resource](http://www.php.net/manual/en/language.types.resource.php)).

**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](http://www.php.net/manual/en/language.types.integer.php) is a number of the set ℤ = {..., -2, -1, 0, 1, 2, ...}.

See also:

* [Arbitrary length integer / GMP](http://www.php.net/manual/en/ref.gmp.php)
* [Floating point numbers](http://www.php.net/manual/en/language.types.float.php)
* [Arbitrary precision / BCMath](http://www.php.net/manual/en/ref.bc.php)

### Syntax

[Integer](http://www.php.net/manual/en/language.types.integer.php)s can be specified in decimal (base 10), hexadecimal (base 16), octal (base 8) or binary (base 2) notation, optionally preceded by a sign (- or +).

Binary [integer](http://www.php.net/manual/en/language.types.integer.php) literals are available since PHP 5.4.0.

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

**Example #1 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)  
$a = 0b11111111; // binary number (equivalent to 255 decimal)  
?>

Formally, the structure for [integer](http://www.php.net/manual/en/language.types.integer.php) literals is:

decimal : [1-9][0-9]\*

| 0

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

octal : 0[0-7]+

binary : 0b[01]+

integer : [+-]?decimal

| [+-]?hexadecimal

| [+-]?octal

| [+-]?binary

The size of an [integer](http://www.php.net/manual/en/language.types.integer.php) is platform-dependent, although a maximum value of about two billion is the usual value (that's 32 bits signed). 64-bit platforms usually have a maximum value of about 9E18, except for Windows, which is always 32 bit. PHP does not support unsigned [integer](http://www.php.net/manual/en/language.types.integer.php)s. [Integer](http://www.php.net/manual/en/language.types.integer.php) 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](http://www.php.net/manual/en/language.types.integer.php) (i.e. 8 or 9), the rest of the number is ignored.

**Example #2 Octal weirdness**

<?php  
var\_dump(01090); // 010 octal = 8 decimal  
?>

### Integer overflow

If PHP encounters a number beyond the bounds of the [integer](http://www.php.net/manual/en/language.types.integer.php) type, it will be interpreted as a [float](http://www.php.net/manual/en/language.types.float.php) instead. Also, an operation which results in a number beyond the bounds of the [integer](http://www.php.net/manual/en/language.types.integer.php) type will return a [float](http://www.php.net/manual/en/language.types.float.php) instead.

**Example #3 Integer overflow on a 32-bit system**

<?php  
$large\_number = 2147483647;  
var\_dump($large\_number);                     // int(2147483647)  
  
$large\_number = 2147483648;  
var\_dump($large\_number);                     // float(2147483648)  
  
$million = 1000000;  
$large\_number =  50000 \* $million;  
var\_dump($large\_number);                     // float(50000000000)  
?>

**Example #4 Integer overflow on a 64-bit system**

<?php  
$large\_number = 9223372036854775807;  
var\_dump($large\_number);                     // int(9223372036854775807)  
  
$large\_number = 9223372036854775808;  
var\_dump($large\_number);                     // float(9.2233720368548E+18)  
  
$million = 1000000;  
$large\_number =  50000000000000 \* $million;  
var\_dump($large\_number);                     // float(5.0E+19)  
?>

There is no [integer](http://www.php.net/manual/en/language.types.integer.php) division operator in PHP. 1/2 yields the [float](http://www.php.net/manual/en/language.types.float.php) 0.5. The value can be casted to an [integer](http://www.php.net/manual/en/language.types.integer.php) to round it downwards, or the [round()](http://www.php.net/manual/en/function.round.php) 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](http://www.php.net/manual/en/language.types.integer.php), 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](http://www.php.net/manual/en/language.types.integer.php) argument. A value can also be converted to [integer](http://www.php.net/manual/en/language.types.integer.php) with the [intval()](http://www.php.net/manual/en/function.intval.php) function.

If a [resource](http://www.php.net/manual/en/language.types.resource.php) is converted to an [integer](http://www.php.net/manual/en/language.types.integer.php), then the result will be the unique resource number assigned to the [resource](http://www.php.net/manual/en/language.types.resource.php) by PHP at runtime.

See also [Type Juggling](http://www.php.net/manual/en/language.types.type-juggling.php).

#### From [booleans](http://www.php.net/manual/en/language.types.boolean.php)

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

#### From [floating point numbers](http://www.php.net/manual/en/language.types.float.php) [¶](http://www.php.net/manual/en/language.types.integer.php#language.types.integer.casting.from-float)

When converting from [float](http://www.php.net/manual/en/language.types.float.php) to [integer](http://www.php.net/manual/en/language.types.integer.php), the number will be rounded towards zero.

If the float is beyond the boundaries of [integer](http://www.php.net/manual/en/language.types.integer.php) (usually +/- 2.15e+9 = 2^31 on 32-bit platforms and +/- 9.22e+18 = 2^63 on 64-bit platforms other than Windows), the result is undefined, since the [float](http://www.php.net/manual/en/language.types.float.php) doesn't have enough precision to give an exact [integer](http://www.php.net/manual/en/language.types.integer.php) result. No warning, not even a notice will be issued when this happens!

**Warning**

Never cast an unknown fraction to [integer](http://www.php.net/manual/en/language.types.integer.php), 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](http://www.php.net/manual/en/language.types.float.php#warn.float-precision).

#### From strings

See [String conversion to numbers](http://www.php.net/manual/en/language.types.string.php#language.types.string.conversion)

#### From other types

**Caution**

The behaviour of converting to [integer](http://www.php.net/manual/en/language.types.integer.php) 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

Floating point numbers have limited precision. Although it depends on the system, PHP typically uses the IEEE 754 double precision format, which will give a maximum relative error due to rounding in the order of 1.11e-16. Non elementary arithmetic operations may give larger errors, and, of course, error propagation must be considered when several operations are compounded.

Additionally, rational numbers that are exactly representable as floating point numbers in base 10, like 0.1 or 0.7, do not have an exact representation as floating point numbers in base 2, which is used internally, no matter the size of the mantissa. Hence, they 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 7.9999999999999991118....

So never trust floating number results to the last digit, and do not compare floating point numbers directly for equality. If higher precision is necessary, the [arbitrary precision math functions](http://www.php.net/manual/en/ref.bc.php) and [gmp](http://www.php.net/manual/en/ref.gmp.php) functions are available.

For a "simple" explanation, see the [» floating point guide](http://floating-point-gui.de/) that's also titled "Why don’t my numbers add up?"

### Converting to float

For information on converting [string](http://www.php.net/manual/en/language.types.string.php)s to [float](http://www.php.net/manual/en/language.types.float.php), see [String conversion to numbers](http://www.php.net/manual/en/language.types.string.php#language.types.string.conversion). For values of other types, the conversion is performed by converting the value to [integer](http://www.php.net/manual/en/language.types.integer.php) first and then to [float](http://www.php.net/manual/en/language.types.float.php). See [Converting to integer](http://www.php.net/manual/en/language.types.integer.php#language.types.integer.casting) for more information. As of PHP 5, a notice is thrown if an [object](http://www.php.net/manual/en/language.types.object.php) is converted to [float](http://www.php.net/manual/en/language.types.float.php).

### Comparing floats

As noted in the warning above, testing floating point values for equality is problematic, due to the way that they are represented internally. However, there are ways to make comparisons of floating point values that work around these limitations.

To test floating point values for equality, an upper bound on the relative error due to rounding is used. This value is known as the machine epsilon, or unit roundoff, and is the smallest acceptable difference in calculations.

$a and $b are equal to 5 digits of precision.

<?php  
$a = 1.23456789;  
$b = 1.23456780;  
$epsilon = 0.00001;  
  
if(abs($a-$b) < $epsilon) {  
    echo "true";  
}  
?>

### NaN

Some numeric operations can result in a value represented by the constant **NAN**. This result represents an undefined or unrepresentable value in floating-point calculations. Any loose or strict comparisons of this value against any other value, including itself, will have a result of **FALSE**.

Because **NAN** represents any number of different values, **NAN** should not be compared to other values, including itself, and instead should be checked for using [is\_nan()](http://www.php.net/manual/en/function.is-nan.php).

## Strings

A [string](http://www.php.net/manual/en/language.types.string.php) is series of characters, where a character is the same as a byte. This means that PHP only supports a 256-character set, and hence does not offer native Unicode support. See [details of the string type](http://www.php.net/manual/en/language.types.string.php#language.types.string.details).

**Note**: [string](http://www.php.net/manual/en/language.types.string.php) can be as large as up to 2GB (2147483647 bytes maximum)

### Syntax

A [string](http://www.php.net/manual/en/language.types.string.php) literal can be specified in four different ways:

* [single quoted](http://www.php.net/manual/en/language.types.string.php#language.types.string.syntax.single)
* [double quoted](http://www.php.net/manual/en/language.types.string.php#language.types.string.syntax.double)
* [heredoc syntax](http://www.php.net/manual/en/language.types.string.php#language.types.string.syntax.heredoc)
* [nowdoc syntax](http://www.php.net/manual/en/language.types.string.php#language.types.string.syntax.nowdoc) (since PHP 5.3.0)

#### Single quoted

The simplest way to specify a [string](http://www.php.net/manual/en/language.types.string.php) 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, double it (\\). All other instances of backslash will be treated as a literal backslash: this means that the other escape sequences you might be used to, such as \r or \n, will be output literally as specified rather than having any special meaning.

**Note**: Unlike the [double-quoted](http://www.php.net/manual/en/language.types.string.php#language.types.string.syntax.double) and [heredoc](http://www.php.net/manual/en/language.types.string.php#language.types.string.syntax.heredoc) syntaxes, [variables](http://www.php.net/manual/en/language.variables.php) and escape sequences for special characters will not be expanded when they occur in single quoted [string](http://www.php.net/manual/en/language.types.string.php)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](http://www.php.net/manual/en/language.types.string.php) 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) |
| \e | escape (ESC or 0x1B (27) in ASCII) (since PHP 5.4.0) |
| \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](http://www.php.net/manual/en/language.types.string.php)s, escaping any other character will result in the backslash being printed too. Before PHP 5.1.1, the backslash in \{$var} had not been printed.

The most important feature of double-quoted [string](http://www.php.net/manual/en/language.types.string.php)s is the fact that variable names will be expanded. See [string parsing](http://www.php.net/manual/en/language.types.string.php#language.types.string.parsing) for details.

#### Heredoc

A third way to delimit [string](http://www.php.net/manual/en/language.types.string.php)s is the heredoc syntax: <<<. After this operator, an identifier is provided, then a newline. The [string](http://www.php.net/manual/en/language.types.string.php) 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 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 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 properties. Since PHP 5.3, this limitation is valid only for heredocs containing variables.

**Example #1 Invalid example**

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

Heredoc text behaves just like a double-quoted [string](http://www.php.net/manual/en/language.types.string.php), 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](http://www.php.net/manual/en/language.types.string.php)s.

**Example #2 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

It is also possible to use the Heredoc syntax to pass data to function arguments:

**Example #3 Heredoc in arguments example**

<?php  
var\_dump(array(<<<EOD  
foobar!  
EOD  
));  
?>

As of PHP 5.3.0, it's possible to initialize static variables and class properties/constants using the Heredoc syntax:

**Example #4 Using Heredoc to initialize static values**

<?php  
// Static variables  
function foo()  
{  
    static $bar = <<<LABEL  
Nothing in here...  
LABEL;  
}  
  
// Class properties/constants  
class foo  
{  
    const BAR = <<<FOOBAR  
Constant example  
FOOBAR;  
  
    public $baz = <<<FOOBAR  
Property example  
FOOBAR;  
}  
?>

Starting with PHP 5.3.0, the opening Heredoc identifier may optionally be enclosed in double quotes:

**Example #5 Using double quotes in Heredoc**

<?php  
echo <<<"FOOBAR"  
Hello World!  
FOOBAR;  
?>

#### Nowdoc [¶](http://www.php.net/manual/en/language.types.string.php#language.types.string.syntax.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 <<< sequence 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 #6 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 properties or constants:

**Example #7 Static data example**

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

**Note**:

Nowdoc support was added in PHP 5.3.0.

#### Variable parsing [¶](http://www.php.net/manual/en/language.types.string.php#language.types.string.parsing)

When a [string](http://www.php.net/manual/en/language.types.string.php) is specified in double quotes or with heredoc, [variables](http://www.php.net/manual/en/language.variables.php) are parsed within it.

There are two types of syntax: a [simple](http://www.php.net/manual/en/language.types.string.php#language.types.string.parsing.simple) one and a [complex](http://www.php.net/manual/en/language.types.string.php#language.types.string.parsing.complex) one. The simple syntax is the most common and convenient. It provides a way to embed a variable, an [array](http://www.php.net/manual/en/language.types.array.php) value, or an [object](http://www.php.net/manual/en/language.types.object.php) property in a [string](http://www.php.net/manual/en/language.types.string.php) with a minimum of effort.

The complex syntax 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  
$juice = "apple";  
  
echo "He drank some $juice juice.".PHP\_EOL;  
// Invalid. "s" is a valid character for a variable name, but the variable is $juice.  
echo "He drank some juice made of $juices.";  
?>

The above example will output:

He drank some apple juice.

He drank some juice made of .

Similarly, an [array](http://www.php.net/manual/en/language.types.array.php) index or an [object](http://www.php.net/manual/en/language.types.object.php) 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.

**Example #8 Simple syntax example**

<?php  
$juices = array("apple", "orange", "koolaid1" => "purple");  
  
echo "He drank some $juices[0] juice.".PHP\_EOL;  
echo "He drank some $juices[1] juice.".PHP\_EOL;  
echo "He drank some $juices[koolaid1] juice.".PHP\_EOL;  
  
class people {  
    public $john = "John Smith";  
    public $jane = "Jane Smith";  
    public $robert = "Robert Paulsen";  
      
    public $smith = "Smith";  
}  
  
$people = new people();  
  
echo "$people->john drank some $juices[0] juice.".PHP\_EOL;  
echo "$people->john then said hello to $people->jane.".PHP\_EOL;  
echo "$people->john's wife greeted $people->robert.".PHP\_EOL;  
echo "$people->robert greeted the two $people->smiths."; // Won't work  
?>

The above example will output:

He drank some apple juice.

He drank some orange juice.

He drank some purple juice.

John Smith drank some apple juice.

John Smith then said hello to Jane Smith.

John Smith's wife greeted Robert Paulsen.

Robert Paulsen greeted the two .

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.

Any scalar variable, array element or object property with a [string](http://www.php.net/manual/en/language.types.string.php) representation can be included via this syntax. Simply write the expression the same way as it would appear outside the [string](http://www.php.net/manual/en/language.types.string.php), 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, quoted keys only work using the curly brace syntax  
echo "This works: {$arr['key']}";  
  
  
// 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()}}";  
  
// Won't work, outputs: This is the return value of getName(): {getName()}  
echo "This is the return value of getName(): {getName()}";  
?>

It is also possible to access class properties using variables within strings using this syntax.

<?php  
class foo {  
    var $bar = 'I am bar.';  
}  
  
$foo = new foo();  
$bar = 'bar';  
$baz = array('foo', 'bar', 'baz', 'quux');  
echo "{$foo->$bar}\n";  
echo "{$foo->$baz[1]}\n";  
?>

The above example will output:

I am bar.

I am bar.

**Note**:

Functions, method calls, static class variables, and class constants inside {$} work since PHP 5. However, the value accessed will be interpreted as the name of a variable in the scope in which the string is defined. Using single curly braces ({}) will not work for accessing the return values of functions or methods or the values of class constants or static class variables.

<?php  
// Show all errors.  
error\_reporting(E\_ALL);  
  
class beers {  
    const softdrink = 'rootbeer';  
    public static $ale = 'ipa';  
}  
  
$rootbeer = 'A & W';  
$ipa = 'Alexander Keith\'s';  
  
// This works; outputs: I'd like an A & W  
echo "I'd like an {${beers::softdrink}}\n";  
  
// This works too; outputs: I'd like an Alexander Keith's  
echo "I'd like an {${beers::$ale}}\n";  
?>

#### String access and modification by character [¶](http://www.php.net/manual/en/language.types.string.php#language.types.string.substr)

Characters within [string](http://www.php.net/manual/en/language.types.string.php)s may be accessed and modified by specifying the zero-based offset of the desired character after the [string](http://www.php.net/manual/en/language.types.string.php) using square [array](http://www.php.net/manual/en/language.types.array.php) brackets, as in $str[42]. Think of a [string](http://www.php.net/manual/en/language.types.string.php) as an [array](http://www.php.net/manual/en/language.types.array.php) of characters for this purpose. The functions [substr()](http://www.php.net/manual/en/function.substr.php) and [substr\_replace()](http://www.php.net/manual/en/function.substr-replace.php) can be used when you want to extract or replace more than 1 character.

**Note**: [String](http://www.php.net/manual/en/language.types.string.php)s may also be accessed using braces, as in $str{42}, for the same purpose.

**Warning**

Writing to an out of range offset pads the string with spaces. Non-integer types are converted to integer. Illegal offset type emits **E\_NOTICE**. Negative offset emits **E\_NOTICE** in write but reads empty string. Only the first character of an assigned string is used. Assigning empty string assigns NULL byte.

**Warning**

Internally, PHP strings are byte arrays. As a result, accessing or modifying a string using array brackets is not multi-byte safe, and should only be done with strings that are in a single-byte encoding such as ISO-8859-1.

**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';  
  
?>

As of PHP 5.4 string offsets have to either be integers or integer-like strings, otherwise a warning will be thrown. Previously an offset like "foo" was silently cast to 0.

**Example #10 Differences between PHP 5.3 and PHP 5.4**

<?php  
$str = 'abc';  
  
var\_dump($str['1']);  
var\_dump(isset($str['1']));  
  
var\_dump($str['1.0']);  
var\_dump(isset($str['1.0']));  
  
var\_dump($str['x']);  
var\_dump(isset($str['x']));  
  
var\_dump($str['1x']);  
var\_dump(isset($str['1x']));  
?>

Output of the above example in PHP 5.3:

string(1) "b"

bool(true)

string(1) "b"

bool(true)

string(1) "a"

bool(true)

string(1) "b"

bool(true)

Output of the above example in PHP 5.4:

string(1) "b"

bool(true)

Warning: Illegal string offset '1.0' in /tmp/t.php on line 7

string(1) "b"

bool(false)

Warning: Illegal string offset 'x' in /tmp/t.php on line 9

string(1) "a"

bool(false)

string(1) "b"

bool(false)

**Note**:

Accessing variables of other types (not including arrays or objects implementing the appropriate interfaces) using [] or {} silently returns **NULL**.

**Note**:

PHP 5.5 added support for accessing characters within string literals using [] or {}.

### Useful functions and operators [¶](http://www.php.net/manual/en/language.types.string.php#language.types.string.useful-funcs)

[String](http://www.php.net/manual/en/language.types.string.php)s may be concatenated using the '.' (dot) operator. Note that the '+' (addition) operator will not work for this. See [String operators](http://www.php.net/manual/en/language.operators.string.php) for more information.

There are a number of useful functions for [string](http://www.php.net/manual/en/language.types.string.php) manipulation.

See the [string functions section](http://www.php.net/manual/en/ref.strings.php) for general functions, and the [regular expression functions](http://www.php.net/manual/en/ref.regex.php) or the [Perl-compatible regular expression functions](http://www.php.net/manual/en/ref.pcre.php) for advanced find & replace functionality.

There are also [functions for URL strings](http://www.php.net/manual/en/ref.url.php), and functions to encrypt/decrypt strings ([mcrypt](http://www.php.net/manual/en/ref.mcrypt.php) and [mhash](http://www.php.net/manual/en/ref.mhash.php)).

Finally, see also the [character type functions](http://www.php.net/manual/en/ref.ctype.php).

### Converting to string

A value can be converted to a [string](http://www.php.net/manual/en/language.types.string.php) using the (string) cast or the [strval()](http://www.php.net/manual/en/function.strval.php) function. [String](http://www.php.net/manual/en/language.types.string.php) conversion is automatically done in the scope of an expression where a [string](http://www.php.net/manual/en/language.types.string.php) is needed. This happens when using the [echo](http://www.php.net/manual/en/function.echo.php) or [print](http://www.php.net/manual/en/function.print.php) functions, or when a variable is compared to a [string](http://www.php.net/manual/en/language.types.string.php). The sections on [Types](http://www.php.net/manual/en/language.types.php) and [Type Juggling](http://www.php.net/manual/en/language.types.type-juggling.php) will make the following clearer. See also the [settype()](http://www.php.net/manual/en/function.settype.php) function.

A [boolean](http://www.php.net/manual/en/language.types.boolean.php) **TRUE** value is converted to the [string](http://www.php.net/manual/en/language.types.string.php) "1". [Boolean](http://www.php.net/manual/en/language.types.boolean.php) **FALSE** is converted to "" (the empty string). This allows conversion back and forth between [boolean](http://www.php.net/manual/en/language.types.boolean.php) and [string](http://www.php.net/manual/en/language.types.string.php) values.

An [integer](http://www.php.net/manual/en/language.types.integer.php) or [float](http://www.php.net/manual/en/language.types.float.php) is converted to a [string](http://www.php.net/manual/en/language.types.string.php) representing the number textually (including the exponent part for [float](http://www.php.net/manual/en/language.types.float.php)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()](http://www.php.net/manual/en/function.setlocale.php) function.

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

[Object](http://www.php.net/manual/en/language.types.object.php)s in PHP 4 are always converted to the [string](http://www.php.net/manual/en/language.types.string.php) "Object". To print the values of object properties for debugging reasons, read the paragraphs below. To get an object's class name, use the [get\_class()](http://www.php.net/manual/en/function.get-class.php) function. As of PHP 5, the [\_\_toString](http://www.php.net/manual/en/language.oop5.magic.php) method is used when applicable.

[Resource](http://www.php.net/manual/en/language.types.resource.php)s are always converted to [string](http://www.php.net/manual/en/language.types.string.php)s with the structure "Resource id #1", where 1 is the resource number assigned to the [resource](http://www.php.net/manual/en/language.types.resource.php) by PHP at runtime. While the exact structure of this string should not be relied on and is subject to change, it will always be unique for a given resource within the lifetime of a script being executed (ie a Web request or CLI process) and won't be reused. To get a [resource](http://www.php.net/manual/en/language.types.resource.php)'s type, use the [get\_resource\_type()](http://www.php.net/manual/en/function.get-resource-type.php) function.

**NULL** is always converted to an empty string.

As stated above, directly converting an [array](http://www.php.net/manual/en/language.types.array.php), [object](http://www.php.net/manual/en/language.types.object.php), or [resource](http://www.php.net/manual/en/language.types.resource.php) to a [string](http://www.php.net/manual/en/language.types.string.php) does not provide any useful information about the value beyond its type. See the functions [print\_r()](http://www.php.net/manual/en/function.print-r.php) and [var\_dump()](http://www.php.net/manual/en/function.var-dump.php) for more effective means of inspecting the contents of these types.

Most PHP values can also be converted to [string](http://www.php.net/manual/en/language.types.string.php)s for permanent storage. This method is called serialization, and is performed by the [serialize()](http://www.php.net/manual/en/function.serialize.php) function. If the PHP engine was built with [WDDX](http://www.php.net/manual/en/ref.wddx.php) support, PHP values can also be serialized as well-formed XML text.

### String conversion to numbers

When a [string](http://www.php.net/manual/en/language.types.string.php) is evaluated in a numeric context, the resulting value and type are determined as follows.

If the [string](http://www.php.net/manual/en/language.types.string.php) does not contain any of the characters '.', 'e', or 'E' and the numeric value fits into integer type limits (as defined by **PHP\_INT\_MAX**), the [string](http://www.php.net/manual/en/language.types.string.php) will be evaluated as an [integer](http://www.php.net/manual/en/language.types.integer.php). In all other cases it will be evaluated as a [float](http://www.php.net/manual/en/language.types.float.php).

The value is given by the initial portion of the [string](http://www.php.net/manual/en/language.types.string.php). If the [string](http://www.php.net/manual/en/language.types.string.php) 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()](http://www.php.net/manual/en/function.ord.php) and [chr()](http://www.php.net/manual/en/function.chr.php) functions to convert between ASCII codes and characters.

### Details of the String Type

The [string](http://www.php.net/manual/en/language.types.string.php) in PHP is implemented as an array of bytes and an integer indicating the length of the buffer. It has no information about how those bytes translate to characters, leaving that task to the programmer. There are no limitations on the values the string can be composed of; in particular, bytes with value 0 (“NUL bytes”) are allowed anywhere in the string (however, a few functions, said in this manual not to be “binary safe”, may hand off the strings to libraries that ignore data after a NUL byte.)

This nature of the string type explains why there is no separate “byte” type in PHP – strings take this role. Functions that return no textual data – for instance, arbitrary data read from a network socket – will still return strings.

Given that PHP does not dictate a specific encoding for strings, one might wonder how string literals are encoded. For instance, is the string "á" equivalent to "\xE1" (ISO-8859-1), "\xC3\xA1" (UTF-8, C form), "\x61\xCC\x81" (UTF-8, D form) or any other possible representation? The answer is that string will be encoded in whatever fashion it is encoded in the script file. Thus, if the script is written in ISO-8859-1, the string will be encoded in ISO-8859-1 and so on. However, this does not apply if Zend Multibyte is enabled; in that case, the script may be written in an arbitrary encoding (which is explicity declared or is detected) and then converted to a certain internal encoding, which is then the encoding that will be used for the string literals. Note that there are some constraints on the encoding of the script (or on the internal encoding, should Zend Multibyte be enabled) – this almost always means that this encoding should be a compatible superset of ASCII, such as UTF-8 or ISO-8859-1. Note, however, that state-dependent encodings where the same byte values can be used in initial and non-initial shift states may be problematic.

Of course, in order to be useful, functions that operate on text may have to make some assumptions about how the string is encoded. Unfortunately, there is much variation on this matter throughout PHP’s functions:

* Some functions assume that the string is encoded in some (any) single-byte encoding, but they do not need to interpret those bytes as specific characters. This is case of, for instance, [substr()](http://www.php.net/manual/en/function.substr.php), [strpos()](http://www.php.net/manual/en/function.strpos.php), [strlen()](http://www.php.net/manual/en/function.strlen.php) or [strcmp()](http://www.php.net/manual/en/function.strcmp.php). Another way to think of these functions is that operate on memory buffers, i.e., they work with bytes and byte offsets.
* Other functions are passed the encoding of the string, possibly they also assume a default if no such information is given. This is the case of [htmlentities()](http://www.php.net/manual/en/function.htmlentities.php) and the majority of the functions in the [mbstring](http://www.php.net/manual/en/book.mbstring.php) extension.
* Others use the current locale (see [setlocale()](http://www.php.net/manual/en/function.setlocale.php)), but operate byte-by-byte. This is the case of [strcasecmp()](http://www.php.net/manual/en/function.strcasecmp.php), [strtoupper()](http://www.php.net/manual/en/function.strtoupper.php) and [ucfirst()](http://www.php.net/manual/en/function.ucfirst.php). This means they can be used only with single-byte encodings, as long as the encoding is matched by the locale. For instance strtoupper("á") may return "Á" if the locale is correctly set and á is encoded with a single byte. If it is encoded in UTF-8, the correct result will not be returned and the resulting string may or may not be returned corrupted, depending on the current locale.
* Finally, they may just assume the string is using a specific encoding, usually UTF-8. This is the case of most functions in the [intl](http://www.php.net/manual/en/book.intl.php) extension and in the [PCRE](http://www.php.net/manual/en/book.pcre.php) extension (in the last case, only when the u modifier is used). Although this is due to their special purpose, the function [utf8\_decode()](http://www.php.net/manual/en/function.utf8-decode.php) assumes a UTF-8 encoding and the function [utf8\_encode()](http://www.php.net/manual/en/function.utf8-encode.php) assumes an ISO-8859-1 encoding.

Ultimately, this means writing correct programs using Unicode depends on carefully avoiding functions that will not work and that most likely will corrupt the data and using instead the functions that do behave correctly, generally from the [intl](http://www.php.net/manual/en/book.intl.php) and [mbstring](http://www.php.net/manual/en/book.mbstring.php) extensions. However, using functions that can handle Unicode encodings is just the beginning. No matter the functions the language provides, it is essential to know the Unicode specification. For instance, a program that assumes there is only uppercase and lowercase is making a wrong assumption.

## Arrays

An [array](http://www.php.net/manual/en/language.types.array.php) 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](http://www.php.net/manual/en/language.types.array.php) values can be other [array](http://www.php.net/manual/en/language.types.array.php)s, trees and multidimensional [array](http://www.php.net/manual/en/language.types.array.php)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()](http://www.php.net/manual/en/function.array.php)

An [array](http://www.php.net/manual/en/language.types.array.php) can be created using the [array()](http://www.php.net/manual/en/function.array.php) language construct. It takes any number of comma-separated *key* => *value* pairs as arguments.

array(

key => value,

key2 => value2,

key3 => value3,

...

)

The comma after the last array element is optional and can be omitted. This is usually done for single-line arrays, i.e. array(1, 2) is preferred over array(1, 2, ). For multi-line arrays on the other hand the trailing comma is commonly used, as it allows easier addition of new elements at the end.

As of PHP 5.4 you can also use the short array syntax, which replaces array() with [].

**Example #1 A simple array**

<?php  
$array = array(  
    "foo" => "bar",  
    "bar" => "foo",  
);  
  
// as of PHP 5.4  
$array = [  
    "foo" => "bar",  
    "bar" => "foo",  
];  
?>

The key can either be an [integer](http://www.php.net/manual/en/language.types.integer.php) or a [string](http://www.php.net/manual/en/language.types.string.php). The value can be of any type.

Additionally the following key casts will occur:

* [String](http://www.php.net/manual/en/language.types.string.php)s containing valid [integer](http://www.php.net/manual/en/language.types.integer.php)s will be cast to the [integer](http://www.php.net/manual/en/language.types.integer.php) type. E.g. the key "8" will actually be stored under 8. On the other hand "08" will not be cast, as it isn't a valid decimal integer.
* [Float](http://www.php.net/manual/en/language.types.float.php)s are also cast to [integer](http://www.php.net/manual/en/language.types.integer.php)s, which means that the fractional part will be truncated. E.g. the key 8.7 will actually be stored under 8.
* [Bool](http://www.php.net/manual/en/language.types.boolean.php)s are cast to [integer](http://www.php.net/manual/en/language.types.integer.php)s, too, i.e. the key true will actually be stored under 1 and the key false under 0.
* [Null](http://www.php.net/manual/en/language.types.null.php) will be cast to the empty string, i.e. the key null will actually be stored under "".
* [Array](http://www.php.net/manual/en/language.types.array.php)s and [object](http://www.php.net/manual/en/language.types.object.php)s can not be used as keys. Doing so will result in a warning: Illegal offset type.

If multiple elements in the array declaration use the same key, only the last one will be used as all others are overwritten.

**Example #2 Type Casting and Overwriting example**

<?php  
$array = array(  
    1    => "a",  
    "1"  => "b",  
    1.5  => "c",  
    true => "d",  
);  
var\_dump($array);  
?>

The above example will output:

array(1) {

[1]=>

string(1) "d"

}

As all the keys in the above example are cast to 1, the value will be overwritten on every new element and the last assigned value "d" is the only one left over.

PHP arrays can contain [integer](http://www.php.net/manual/en/language.types.integer.php) and [string](http://www.php.net/manual/en/language.types.string.php) keys at the same time as PHP does not distinguish between indexed and associative arrays.

**Example #3 Mixed** [**integer**](http://www.php.net/manual/en/language.types.integer.php) **and** [**string**](http://www.php.net/manual/en/language.types.string.php) **keys**

<?php  
$array = array(  
    "foo" => "bar",  
    "bar" => "foo",  
    100   => -100,  
    -100  => 100,  
);  
var\_dump($array);  
?>

The above example will output:

array(4) {

["foo"]=>

string(3) "bar"

["bar"]=>

string(3) "foo"

[100]=>

int(-100)

[-100]=>

int(100)

}

The key is optional. If it is not specified, PHP will use the increment of the largest previously used [integer](http://www.php.net/manual/en/language.types.integer.php) key.

**Example #4 Indexed arrays without key**

<?php  
$array = array("foo", "bar", "hello", "world");  
var\_dump($array);  
?>

The above example will output:

array(4) {

[0]=>

string(3) "foo"

[1]=>

string(3) "bar"

[2]=>

string(5) "hello"

[3]=>

string(5) "world"

}

It is possible to specify the key only for some elements and leave it out for others:

**Example #5 Keys not on all elements**

<?php  
$array = array(  
         "a",  
         "b",  
    6 => "c",  
         "d",  
);  
var\_dump($array);  
?>

The above example will output:

array(4) {

[0]=>

string(1) "a"

[1]=>

string(1) "b"

[6]=>

string(1) "c"

[7]=>

string(1) "d"

}

As you can see the last value "d" was assigned the key 7. This is because the largest integer key before that was 6.

#### Accessing array elements with square bracket syntax

Array elements can be accessed using the array[key] syntax.

**Example #6 Accessing array elements**

<?php  
$array = array(  
    "foo" => "bar",  
    42    => 24,  
    "multi" => array(  
         "dimensional" => array(  
             "array" => "foo"  
         )  
    )  
);  
  
var\_dump($array["foo"]);  
var\_dump($array[42]);  
var\_dump($array["multi"]["dimensional"]["array"]);  
?>

The above example will output:

string(3) "bar"

int(24)

string(3) "foo"

**Note**:

Both square brackets and curly braces can be used interchangeably for accessing array elements (e.g. $array[42] and $array{42} will both do the same thing in the example above).

As of PHP 5.4 it is possible to array dereference the result of a function or method call directly. Before it was only possible using a temporary variable.

As of PHP 5.5 it is possible to array dereference an array literal.

**Example #7 Array dereferencing**

<?php  
function getArray() {  
    return array(1, 2, 3);  
}  
  
// on PHP 5.4  
$secondElement = getArray()[1];  
  
// previously  
$tmp = getArray();  
$secondElement = $tmp[1];  
  
// or  
list(, $secondElement) = getArray();  
?>

**Note**:

Attempting to access an array key which has not been defined is the same as accessing any other undefined variable: an **E\_NOTICE**-level error message will be issued, and the result will be **NULL**.

#### Creating/modifying with square bracket syntax

An existing [array](http://www.php.net/manual/en/language.types.array.php) can be modified by explicitly setting values in it.

This is done by assigning values to the [array](http://www.php.net/manual/en/language.types.array.php), 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](http://www.php.net/manual/en/language.types.integer.php) or [string](http://www.php.net/manual/en/language.types.string.php)

// 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](http://www.php.net/manual/en/language.types.array.php). This practice is however discouraged because if $arr already contains some value (e.g. [string](http://www.php.net/manual/en/language.types.string.php) from request variable) then this value will stay in the place and [] may actually stand for [string access operator](http://www.php.net/manual/en/language.types.string.php#language.types.string.substr). It is always better to initialize variable by a direct assignment.

To change a certain value, assign a new value to that element using its key. To remove a key/value pair, call the [unset()](http://www.php.net/manual/en/function.unset.php) function on it.

<?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](http://www.php.net/manual/en/language.types.integer.php) indices is taken, and the new key will be that maximum value plus 1 (but at least 0). If no [integer](http://www.php.net/manual/en/language.types.integer.php) indices exist yet, the key will be 0 (zero).

Note that the maximum integer key used for this need not currently exist in the [*array*](http://www.php.net/manual/en/language.types.array.php). It need only have existed in the [array](http://www.php.net/manual/en/language.types.array.php) at some time since the last time the [array](http://www.php.net/manual/en/language.types.array.php) 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](http://www.php.net/manual/en/ref.array.php) section.

**Note**:

The [unset()](http://www.php.net/manual/en/function.unset.php) function allows removing keys from an [array](http://www.php.net/manual/en/language.types.array.php). Be aware that the array will not be reindexed. If a true "remove and shift" behavior is desired, the [array](http://www.php.net/manual/en/language.types.array.php) can be reindexed using the [array\_values()](http://www.php.net/manual/en/function.array-values.php) 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](http://www.php.net/manual/en/control-structures.foreach.php) control structure exists specifically for [array](http://www.php.net/manual/en/language.types.array.php)s. It provides an easy way to traverse an [array](http://www.php.net/manual/en/language.types.array.php).

### 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](http://www.php.net/manual/en/language.types.string.php) ('bar' - notice the quotes). PHP may in the future define constants which, unfortunately for such code, have the same name. It works because PHP automatically converts a bare string (an unquoted [string](http://www.php.net/manual/en/language.types.string.php) which does not correspond to any known symbol) into a [string](http://www.php.net/manual/en/language.types.string.php) which contains the bare [string](http://www.php.net/manual/en/language.types.string.php). For instance, if there is no defined constant named **bar**, then PHP will substitute in the [string](http://www.php.net/manual/en/language.types.string.php) 'bar' and use that.

**Note**: This does not mean to always quote the key. Do not quote keys which are [constants](http://www.php.net/manual/en/language.constants.php) or [variables](http://www.php.net/manual/en/language.variables.php), 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](http://www.php.net/manual/en/errorfunc.configuration.php#ini.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](http://www.php.net/manual/en/errorfunc.configuration.php#ini.error-reporting) is set not to show notices.

As stated in the [syntax](http://www.php.net/manual/en/language.types.array.php#language.types.array.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 occurred";  
$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 occurred";  
$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](http://www.php.net/manual/en/reserved.php).

**Note**: To reiterate, inside a double-quoted [string](http://www.php.net/manual/en/language.types.string.php), 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](http://www.php.net/manual/en/language.types.string.php#language.types.string.parsing).

### Converting to array [¶](http://www.php.net/manual/en/language.types.array.php#language.types.array.casting)

For any of the types: [integer](http://www.php.net/manual/en/language.types.integer.php), [float](http://www.php.net/manual/en/language.types.float.php), [string](http://www.php.net/manual/en/language.types.string.php), [boolean](http://www.php.net/manual/en/language.types.boolean.php) and [resource](http://www.php.net/manual/en/language.types.resource.php), converting a value to an [array](http://www.php.net/manual/en/language.types.array.php) 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](http://www.php.net/manual/en/language.types.object.php) is converted to an [array](http://www.php.net/manual/en/language.types.array.php), the result is an [array](http://www.php.net/manual/en/language.types.array.php) whose elements are the [object](http://www.php.net/manual/en/language.types.object.php)'s properties. The keys are the member variable names, with a few notable exceptions: integer properties are unaccessible; 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](http://www.php.net/manual/en/language.types.array.php) results in an empty [array](http://www.php.net/manual/en/language.types.array.php).

### Comparing

It is possible to compare arrays with the [array\_diff()](http://www.php.net/manual/en/function.array-diff.php) function and with [array operators](http://www.php.net/manual/en/language.operators.array.php).

### 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  
          );  
  
$b = array('a', 'b', 'c');  
  
// . . .is completely equivalent with this:  
$a = array();  
$a['color'] = 'red';  
$a['taste'] = 'sweet';  
$a['shape'] = 'round';  
$a['name']  = 'apple';  
$a[]        = 4;        // key will be 0  
  
$b = array();  
$b[] = 'a';  
$b[] = 'b';  
$b[] = 'c';  
  
// After the above code is executed, $a will be the array  
// array('color' => 'red', 'taste' => 'sweet', 'shape' => 'round',   
// 'name' => 'apple', 0 => 4), and $b will be the array   
// array(0 => 'a', 1 => 'b', 2 => 'c'), or simply array('a', 'b', 'c').  
?>

**Example #8 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 #9 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](http://www.php.net/manual/en/language.types.array.php) directly is possible since PHP 5 by passing them by reference. Before that, a workaround is necessary:

**Example #10 Changing element in the loop**

<?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 #11 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 #12 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](http://www.php.net/manual/en/language.types.array.php)s are ordered. The order can be changed using various sorting functions. See the [array functions](http://www.php.net/manual/en/ref.array.php) section for more information. The [count()](http://www.php.net/manual/en/function.count.php) function can be used to count the number of items in an [array](http://www.php.net/manual/en/language.types.array.php).

**Example #13 Sorting an array**

<?php  
sort($files);  
print\_r($files);  
?>

Because the value of an [array](http://www.php.net/manual/en/language.types.array.php) can be anything, it can also be another [array](http://www.php.net/manual/en/language.types.array.php). This enables the creation of recursive and multi-dimensional [array](http://www.php.net/manual/en/language.types.array.php)s.

**Example #14 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](http://www.php.net/manual/en/language.types.array.php) assignment always involves value copying. Use the [reference operator](http://www.php.net/manual/en/language.operators.php) to copy an [array](http://www.php.net/manual/en/language.types.array.php) 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](http://www.php.net/manual/en/language.types.object.php), 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](http://www.php.net/manual/en/language.oop5.php) chapter.

### Converting to object

If an [object](http://www.php.net/manual/en/language.types.object.php) is converted to an [object](http://www.php.net/manual/en/language.types.object.php), it is not modified. If a value of any other type is converted to an [object](http://www.php.net/manual/en/language.types.object.php), a new instance of the stdClass built-in class is created. If the value was **NULL**, the new instance will be empty. [Array](http://www.php.net/manual/en/language.types.array.php)s convert to an [object](http://www.php.net/manual/en/language.types.object.php) 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](http://www.php.net/manual/en/language.types.resource.php) is a special variable, holding a reference to an external resource. Resources are created and used by special functions. See the [appendix](http://www.php.net/manual/en/resource.php) for a listing of all these functions and the corresponding [resource](http://www.php.net/manual/en/language.types.resource.php) types.

See also the [get\_resource\_type()](http://www.php.net/manual/en/function.get-resource-type.php) function.

### Converting to resource

As [resource](http://www.php.net/manual/en/language.types.resource.php) variables hold special handlers to opened files, database connections, image canvas areas and the like, converting to a [resource](http://www.php.net/manual/en/language.types.resource.php) makes no sense.

### Freeing resources

Thanks to the reference-counting system introduced with PHP 4's Zend Engine, a [resource](http://www.php.net/manual/en/language.types.resource.php) 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](http://www.php.net/manual/en/features.persistent-connections.php) section for more information.

## NULL

The special **NULL** value represents a variable with no value. **NULL** is the only possible value of type [null](http://www.php.net/manual/en/language.types.null.php).

A variable is considered to be [null](http://www.php.net/manual/en/language.types.null.php) if:

* it has been assigned the constant **NULL**.
* it has not been set to any value yet.
* it has been [unset()](http://www.php.net/manual/en/function.unset.php).

### Syntax

There is only one value of type [null](http://www.php.net/manual/en/language.types.null.php), and that is the case-insensitive constant **NULL**.

<?php  
$var = NULL;         
?>

See also the functions [is\_null()](http://www.php.net/manual/en/function.is-null.php) and [unset()](http://www.php.net/manual/en/function.unset.php).

### Casting to NULL

Casting a variable to [null](http://www.php.net/manual/en/language.types.null.php) using (unset) $var will not remove the variable or unset its value. It will only return a **NULL** value.

## Callbacks

Callbacks can be denoted by [callable](http://www.php.net/manual/en/language.types.callable.php) type hint as of PHP 5.4. This documentation used [callback](http://www.php.net/manual/en/language.pseudo-types.php#language.types.callback) type information for the same purpose.

Some functions like [call\_user\_func()](http://www.php.net/manual/en/function.call-user-func.php) or [usort()](http://www.php.net/manual/en/function.usort.php) accept user-defined callback functions as a parameter. Callback functions can not only be simple functions, but also [object](http://www.php.net/manual/en/language.types.object.php) methods, including static class methods.

### Passing

A PHP function is passed by its name as a [string](http://www.php.net/manual/en/language.types.string.php). Any built-in or user-defined function can be used, except language constructs such as: [array()](http://www.php.net/manual/en/function.array.php), [echo](http://www.php.net/manual/en/function.echo.php), [empty()](http://www.php.net/manual/en/function.empty.php), [eval()](http://www.php.net/manual/en/function.eval.php), [exit()](http://www.php.net/manual/en/function.exit.php), [isset()](http://www.php.net/manual/en/function.isset.php), [list()](http://www.php.net/manual/en/function.list.php), [print](http://www.php.net/manual/en/function.print.php) or [unset()](http://www.php.net/manual/en/function.unset.php).

A method of an instantiated [object](http://www.php.net/manual/en/language.types.object.php) is passed as an [array](http://www.php.net/manual/en/language.types.array.php) containing an [object](http://www.php.net/manual/en/language.types.object.php) at index 0 and the method name at index 1.

Static class methods can also be passed without instantiating an [object](http://www.php.net/manual/en/language.types.object.php) of that class by passing the class name instead of an [object](http://www.php.net/manual/en/language.types.object.php) at index 0. As of PHP 5.2.3, it is also possible to pass 'ClassName::methodName'.

Apart from common user-defined function, [anonymous functions](http://www.php.net/manual/en/functions.anonymous.php) can also be passed to a callback parameter.

**Example #1 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  
?>

**Example #2 Callback example using a Closure**

<?php  
// Our closure  
$double = function($a) {  
    return $a \* 2;  
};  
  
// This is our range of numbers  
$numbers = range(1, 5);  
  
// Use the closure as a callback here to   
// double the size of each element in our   
// range  
$new\_numbers = array\_map($double, $numbers);  
  
print implode(' ', $new\_numbers);  
?>

The above example will output:

2 4 6 8 10

**Note**: In PHP 4, it was necessary to use a reference to create a callback that points to the actual [object](http://www.php.net/manual/en/language.types.object.php), and not a copy of it. For more details, see [References Explained](http://www.php.net/manual/en/language.references.php).

**Note**:

Callbacks registered with functions such as [call\_user\_func()](http://www.php.net/manual/en/function.call-user-func.php) and [call\_user\_func\_array()](http://www.php.net/manual/en/function.call-user-func-array.php) will not be called if there is an uncaught exception thrown in a previous callback.

## Pseudo-types and variables used in this documentation

### mixed

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

[gettype()](http://www.php.net/manual/en/function.gettype.php) for example will accept all PHP types, while [str\_replace()](http://www.php.net/manual/en/function.str-replace.php) will accept [string](http://www.php.net/manual/en/language.types.string.php)s and [array](http://www.php.net/manual/en/language.types.array.php)s.

### number

number indicates that a parameter can be either [integer](http://www.php.net/manual/en/language.types.integer.php) or [float](http://www.php.net/manual/en/language.types.float.php).

### callback

[callback](http://www.php.net/manual/en/language.pseudo-types.php#language.types.callback) pseudo-types was used in this documentation before [callable](http://www.php.net/manual/en/language.types.callable.php) type hint was introduced by PHP 5.4. It means exactly the same.

### 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](http://www.php.net/manual/en/language.types.string.php) value is assigned to variable $var, $var becomes a [string](http://www.php.net/manual/en/language.types.string.php). If an [integer](http://www.php.net/manual/en/language.types.integer.php) value is then assigned to $var, it becomes an [integer](http://www.php.net/manual/en/language.types.integer.php).

An example of PHP's automatic type conversion is the addition operator '+'. If either operand is a [float](http://www.php.net/manual/en/language.types.float.php), then both operands are evaluated as [float](http://www.php.net/manual/en/language.types.float.php)s, and the result will be a [float](http://www.php.net/manual/en/language.types.float.php). Otherwise, the operands will be interpreted as [integer](http://www.php.net/manual/en/language.types.integer.php)s, and the result will also be an [integer](http://www.php.net/manual/en/language.types.integer.php). 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](http://www.php.net/manual/en/language.types.string.php#language.types.string.conversion).

To force a variable to be evaluated as a certain type, see the section on [Type casting](http://www.php.net/manual/en/language.types.type-juggling.php#language.types.typecasting). To change the type of a variable, see the [settype()](http://www.php.net/manual/en/function.settype.php) function.

To test any of the examples in this section, use the [var\_dump()](http://www.php.net/manual/en/function.var-dump.php) function.

**Note**:

The behaviour of an automatic conversion to [array](http://www.php.net/manual/en/language.types.array.php) is currently undefined.

Also, because PHP supports indexing into [string](http://www.php.net/manual/en/language.types.string.php)s via offsets using the same syntax as [array](http://www.php.net/manual/en/language.types.array.php) 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](http://www.php.net/manual/en/language.types.string.php#language.types.string.substr) 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](http://www.php.net/manual/en/language.types.integer.php)
* (bool), (boolean) - cast to [boolean](http://www.php.net/manual/en/language.types.boolean.php)
* (float), (double), (real) - cast to [float](http://www.php.net/manual/en/language.types.float.php)
* (string) - cast to [string](http://www.php.net/manual/en/language.types.string.php)
* (array) - cast to [array](http://www.php.net/manual/en/language.types.array.php)
* (object) - cast to [object](http://www.php.net/manual/en/language.types.object.php)
* (unset) - cast to [NULL](http://www.php.net/manual/en/language.types.null.php) (PHP 5)

(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](http://www.php.net/manual/en/language.types.string.php)s and variables to binary [string](http://www.php.net/manual/en/language.types.string.php)s:

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

**Note**:

Instead of casting a variable to a [string](http://www.php.net/manual/en/language.types.string.php), 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";  
}  
?>

**Next Topics**

## Variables

## Operators

## Control Structures

## Language Constructs and Functions

## Namespaces

## Extensions

## Config

## Performance/bytecode caching \*

There is a type trap in PHP: false is less than any negative figure.  
<?php  
    var\_dump(-1>false);  
?>

if we use gettype() before initializinf any variable it give NULL   
for eg.   
  
<?php   
$foo;   
echo gettype($foo);   
?>   
  
it will show   
  
NULL

Note that you can chain type castng:  
  
var\_dump((string)(int)false); //string(1) "0"

<?php  
  
/\*  delcare few variable to check their data type \*/  
$intVar = 10;   //Integar variable  
  
$floatVar = 2.49;   //Float type aka Double  
  
$stringVar = "There are 8 data types in PHP";   // String  
  
$arrayVar = array(  
    'Boolean',   
    'Integar',   
    'Float aka Double',  
    'String', 'Array',  
    'Object',  
    'Resource',  
    'Null'  
);  // Array type  
  
$booleanVar = true; // Boolean type  
  
// Make an empty class to check object type  
class obj{  
  
}  
  
$objectVar = new obj();  
  
// To check resource type  
$resource = mysql\_connect();  
  
// Null value  
$nullVar = null;  
  
// output the data type of each variable  
echo gettype($intVar);  
echo "\n";  
echo gettype($floatVar);  
echo "\n";  
echo  gettype($stringVar);  
echo "\n";  
echo gettype($arrayVar);  
echo "\n";  
echo gettype($booleanVar);  
echo "\n";  
echo gettype($objectVar);  
echo "\n";  
echo gettype($resource);  
echo "\n";  
echo gettype($nullVar);  
echo "\n";

The Object (compound) Type  
  
Like every programming language, PHP offers the usual basic primitive types which can hold only one piece of data at a time (scalar). I am particularly fond of the "object" type (compound) because that allows me to group many basic PHP types together, and I can name it anything I want.  
  
<?php  
class Person  
{  
  $firstName;                   // a PHP String  
  $middleName;                  // a PHP String  
  $lastName;                    // a PHP String  
  $age;                         // a PHP Integer  
  $hasDriversLicense;           // a PHP Boolean   
}  
?>  
  
Here, I have grouped several basic PHP types together, (3) Strings, (1) Integer, and (1) Boolean... then I named that group "Person". Since I used the proper syntax to do so, this code is pure PHP, which means that if you run this code, you would have an extra PHP "type" available to you in your scripts, like so:  
  
<?php  
$myAge = 16;                    // a PHP Integer - always available  
$yourAge = 15.5;                // a PHP Float   - always available  
$hasHair = true;                // a PHP Boolean - always available  
$greeting = "Hello World!"      // a PHP String  - always available  
  
$person = new Person();         // a PHP Person  - available NOW!  
?>  
  
You can make your own object types and have PHP execute it as if it were part of the PHP language itself. See more on classes and objects in this manual at: <http://www.php.net/manual/en/language.oop5.php>

<?php   
      $a=23;  
      $b='php';  
  
      if(is\_int($a))  
     {  
           echo ++$b;     //Out put phq    
     }  
?>  
  
So, String also incrementing , cooll

One small observation about setting type :  
  
<?php  
        $string = "Hello World!";  
  
        // wrong way  
  
       $stringToInt = settype($string, "integer");     
  
      // correct way  
  
     echo  settype($string, "integer");  // integer  
     echo gettype($stringToFloat) ; // boolean  
?>

In languages like JAVA, we usually have to define the variable data types as well while declaring them but in PHP, we don't have to define the type of the variable while declaring it. Just give a variable any value. For example, we have variable $myFolks. You can set its value as under, say to a string:  
  
$myFolks = "Hossayn opposed Tyranny of Yazid."  
  
And then you can call it or print it and while doing so, you don't have to mention its data type. Afterwards, you can change data type as well:  
  
$myFolks = 5+12+14  
Or just $myFolks = TRUE

Ah, yes, booleans - bit values that are either set (TRUE) or not set (FALSE).  Now that we have 64 bit compilers using an int variable for booleans, there is \*one\* value which is FALSE (zero) and 2\*\*64-1 values that are TRUE (everything else).  It appears there's a lot more truth in this universe, but false can trump anything that's true...  
  
PHP's handling of strings as booleans is \*almost\* correct - an empty string is FALSE, and a non-empty string is TRUE - with one exception:  A string containing a single zero is considered FALSE.  Why?  If \*any\* non-empty strings are going to be considered FALSE, why \*only\* a single zero?  Why not "FALSE" (preferably case insensitive), or "0.0" (with how many decimal places), or "NO" (again, case insensitive), or ... ?  
  
The \*correct\* design would have been that \*any\* non-empty string is TRUE - period, end of story.  Instead, there's another GOTCHA for the less-than-completely-experienced programmer to watch out for, and fixing the language's design error at this late date would undoubtedly break so many things that the correction is completely out of the question.  
  
Speaking of GOTCHAs, consider this code sequence:  
<?php  
$x=TRUE;  
$y=FALSE;  
$z=$y OR $x;  
?>  
  
Is $z TRUE or FALSE?  
  
In this case, $z will be FALSE because the above code is equivalent to <?php ($z=$y) OR $x ?> rather than <?php $z=($y OR $x) ?> as might be expected - because the OR operator has lower precedence than assignment operators.  
  
On the other hand, after this code sequence:  
<?php  
$x=TRUE;  
$y=FALSE;  
$z=$y || $x;  
?>  
  
$z will be TRUE, as expected, because the || operator has higher precedence than assignment:  The code is equivalent to $z=($y OR $x).  
  
This is why you should NEVER use the OR operator without explicit parentheses around the expression where it is being used.

Beware of certain control behavior with boolean and non boolean values :  
  
<?php  
// Consider that the 0 could by any parameters including itself  
var\_dump(0 == 1); // false  
var\_dump(0 == (bool)'all'); // false  
var\_dump(0 == 'all'); // TRUE, take care  
var\_dump(0 === 'all'); // false  
  
// To avoid this behavior, you need to cast your parameter as string like that :  
var\_dump((string)0 == 'all'); // false  
?>

PHP does not break any rules with the values of true and false.  The value false is not a constant for the number 0, it is a boolean value that indicates false.  The value true is also not a constant for 1, it is a special boolean value that indicates true.  It just happens to cast to integer 1 when you print it or use it in an expression, but it's not the same as a constant for the integer value 1 and you shouldn't use it as one.  Notice what it says at the top of the page:  
  
A boolean expresses a truth value.  
  
It does not say "a boolean expresses a 0 or 1".  
  
It's true that symbolic constants are specifically designed to always and only reference their constant value.  But booleans are not symbolic constants, they are values.  If you're trying to add 2 boolean values you might have other problems in your application.

// someKey is a boolean true  
$array = array('someKey'=>true);  
  
// in the following 'false' string gets converted to a boolean true  
if($array['someKey'] != 'false')  
    echo 'The value of someKey is '.$array['someKey'];  
  
As a result the above will output nothing :)   
  
if($array['someKey'] == 'false')  
    echo 'The value of someKey is '.$array['someKey'];  
  
And the above will output  
The value of someKey is 1  
  
In short true == 'false' is true.

Dunno if someone else posted this solution already, but if not, here's a useful and function to convert strings to strict booleans.  
Note this one only checks for string and defaults to the PHP (boolean) cast where e.g. -1 returns true, but you easily add some elseifs for other datatypes.  
  
<?php  
function toStrictBoolean ($\_val, $\_trueValues = array('yes', 'y', 'true'), $\_forceLowercase = true)  
{  
    if (is\_string($\_val)) {  
        return (in\_array(  
             ($\_forceLowercase?strtolower($\_val):$\_val)  
            , $\_trueValues)  
        );  
    } else {  
        return (boolean) $\_val;  
    }  
}  
?>

It is correct that TRUE or FALSE should not be used as constants for the numbers 0 and 1. But there may be times when it might be helpful to see the value of the Boolean as a 1 or 0. Here's how to do it.   
  
<?php   
$var1 = TRUE;   
$var2 = FALSE;   
  
echo $var1; // Will display the number 1   
  
echo $var2; //Will display nothing   
  
/\* To get it to display the number 0 for   
a false value you have to typecast it: \*/   
  
echo (int)$var2; //This will display the number 0 for false.   
?>

Note you can also use the '!' to convert a number to a boolean, as if it was an explicit (bool) cast then NOT.  
  
So you can do something like:  
  
<?php  
$t = !0; // This will === true;  
$f = !1; // This will === false;  
?>  
  
And non-integers are casted as if to bool, then NOT.  
  
Example:  
  
<?php  
$a = !array();      // This will === true;  
$a = !array('a');   // This will === false;  
$s = !"";           // This will === true;  
$s = !"hello";      // This will === false;  
?>  
  
To cast as if using a (bool) you can NOT the NOT with "!!" (double '!'), then you are casting to the correct (bool).  
  
Example:  
  
<?php  
$a = !!array();   // This will === false; (as expected)  
/\*   
This can be a substitute for count($array) > 0 or !(empty($array)) to check to see if an array is empty or not  (you would use: !!$array).  
\*/  
  
$status = (!!$array ? 'complete' : 'incomplete');  
  
$s = !!"testing"; // This will === true; (as expected)  
/\*   
Note: normal casting rules apply so a !!"0" would evaluate to an === false  
\*/  
?>

Since I haven't seen it posted.  
Here is a function that you can use if you have a need to force strict boolean values.  
Hopefully this will save someone some time from searching for similar.  
<?php  
function strictBool($val=false){  
    return is\_integer($val)?false:$val == 1;  
}  
?>  
  
Simply put, it verifies that the value passed is (bool)true otherwise it's false.  
  
Examples:  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
<?php  
$myBool = strictBool(true);  
var\_dump($myBool);  
//returns (bool)true  
  
$myar = array(0 => true);  
$myBool = strictBool($myar[0]);  
var\_dump($myBool);  
//returns (bool)true  
  
$myBool = strictBool("hello");  
var\_dump($myBool);  
//returns (bool)false  
  
$myBool = strictBool(false);  
var\_dump($myBool);  
//returns (bool)false  
  
$myBool = strictBool(array(0 => "hello"));  
var\_dump($myBool);  
//returns (bool)false  
  
$myBool = strictBool(1);  
var\_dump($myBool);  
//returns (bool)false  
  
$myBool = strictBool();  
var\_dump($myBool);  
//returns (bool)false  
?>

Much of the confusion about booleans (but not limited to booleans) is the fact that PHP itself automatically makes a type cast or conversion for you, which may NOT be what you want or expect. In most cases, it's better to provide functions that give your program the exact behavior you want.  
<?php  
  
function boolNumber($bValue = false) {                      // returns integer  
  return ($bValue ? 1 : 0);  
}  
  
function boolString($bValue = false) {                      // returns string  
  return ($bValue ? 'true' : 'false');  
}  
  
$a = true;                                                  // boolean value  
echo 'boolean $a AS string = ' . boolString($a) . '<br>';   // boolean as a string  
echo 'boolean $a AS number = ' . boolNumber($a) . '<br>';   // boolean as a number  
echo '<br>';  
  
$b = (45 > 90);                                             // boolean value  
echo 'boolean $b AS string = ' . boolString($b) . '<br>';   // boolean as a string  
echo 'boolean $b AS number = ' . boolNumber($b) . '<br>';   // boolean as a number  
echo '<br>';  
  
$c = boolNumber(10 > 8) + boolNumber(!(5 > 10));            // adding booleans  
echo 'integer $c = ' . $c .'<br>';  
  
?>  
Results in the following being printed...  
  
boolean $a AS string = true  
boolean $a AS number = 1  
  
boolean $b AS string = false  
boolean $b AS number = 0  
  
integer $c = 2  
  
In other words, if we know what we want out of our program, we can create functions to accommodate. Here, we just wanted 'manual control' over numbers and strings, so that PHP doesn't confuse us.

Just a side note, doesn't really matters, the reason -1 is true and not false is because boolean type is treated as unsigned, so -1 would be for example, if it's unsigned int32 translate to hex: 0xFFFFFFFF and back to decimal: 4294967295 which is non-zero. there isn't really a "negative boolean". it's a binary thing. :o (since it used to be a bit and then there was only 0 and 1 as an option)

Beware that "0.00" converts to boolean TRUE !  
  
You may get such a string from your database, if you have columns of type DECIMAL or CURRENCY. In such cases you have to explicitly check if the value is != 0 or to explicitly convert the value to int also, not only to boolean.

Note that the symbolic constants TRUE and FALSE are treated differently.  I was told that this is a feature, not a bug.  
  
echo false ;  
echo (false) ;  
echo false+false ;  
echo (false+false) ;  
echo intval(false) ;  
echo '"'.false.'"' ;  
  
echo true ;  
echo (true) ;  
echo true+true ;  
echo (true+true) ;  
echo intval(true) ;  
echo '"'.true.'"' ;  
  
should produce  
  
00000"0"11221"1"  
  
but instead produces  
  
000""11221"1"  
  
In other words, the only way to output the underlying zero or use it in a string is to use 'false+false' or pass it through intval().  No such tricks are required to get at the 1 that underlies true.  
  
The whole idea of symbolic constants is that the underlying value \*always\* replaces them during translation, and thus anywhere you would otherwise have to use some obscure "magic number" such as 191, you can use a symbolic constant that makes sense, such as TOTAL\_NATIONS.    
  
Exactly what php gets out of breaking this rule was not explained to me.

Casting bools to string is not working as maybe expected:  
  
echo 'true as string gives [' . (string) true . "] not [true].\n";  
echo 'false as string gives [' . (string) false . "] not [false].\n";  
  
Output:  
  
true as string gives [1] not [true].  
false as string gives [] not [false].  
  
This helps around that behavior:  
  
true ? 'true' : 'false'

Function to sort array by elements and count of element (before php 5.3) (not use Lambda Functions, and Closures)  
  
<?php  
  
//-----------------------------  
  
function arraySortByElements($array2sort,$sortField,$order,$iscount=false) {  
       
        $functionString='  
        if ('.($iscount?'true':'false').'){  
              if(count($a["'.$sortField.'"]) > count($b["'.$sortField.'"])) return 1\*'.$order.';  
            if(count($a["'.$sortField.'"]) < count($b["'.$sortField.'"])) return -1\*'.$order.';  
          }else{  
            if($a["'.$sortField.'"] > $b["'.$sortField.'"]) return 1\*'.$order.';  
            if($a["'.$sortField.'"] < $b["'.$sortField.'"]) return -1\*'.$order.';  
          }  
        return 0;';  
          
     usort($array2sort, create\_function('$a,$b',$functionString));  
     return $array2sort;  
}  
  
//-----------------------------  
  
//init Array for testing :  
$testArray = array(   
          array('name' => 'Lenny', 'note' => 5, 'listId' => array(654,987,32165)),   
          array('name' => 'Olivier', 'note' =>3, 'listId' => array(2)),   
          array('name' => 'Gregory', 'note' => 1, 'listId' => array(45,58)),   
          array('name' => 'Clement', 'note' => 2, 'listId' => array(584,587,741,14781,147))  
        );  
  
//sorted Arrays :  
        $testArrayByNameASC = arraySortByElements($testArray,'name',1);  
        $testArrayByNoteDESC = arraySortByElements($testArray,'note',-1);  
        $testArrayByCountlistIdDESC = arraySortByElements($testArray,'listId',-1,true);  
  
?>

Be careful when assigning a value in the if statement, for example:  
  
if($var = $arg)  
  
$var might be assigned "1" instead of the expected value in $arg.  
  
<?php  
  
public function myMethod()  
{  
return 'test';  
}  
  
public function myOtherMethod()  
{  
return null;  
}  
  
if($val = $this->myMethod())  
{  
// $val might be 1 instead of the expected 'test'  
}  
  
if( ($val = $this->myMethod()) )  
{  
// now $val should be 'test'  
}  
  
// or to check for false  
if( !($val = $this->myMethod()) )  
{  
// this will not run since $val = 'test' and equates to true  
}  
  
// this is an easy way to assign default value only if a value is not returned:  
  
if( !($val = $this->myOtherMethod()) )  
{  
$val = 'default'  
}  
  
?>

Note that the comparison: (false == 0) evaluates to true and so will any value you set to false as well (without casting).

This is simple example how can convert String to Boolean.  
  
<?php  
  
// Convert string to boolean  
  
function convertStr($str) {  
    return is\_string($str) ? (bool) $str : (string) $str;  
}  
  
// Example  
  
$foo = convertStr('apple');  
var\_dump($foo);  // Return TRUE  
  
$foo = convertStr(1); // Return STRING  
var\_dump($foo);  
  
?>

A note when working with PostgreSQL - if you select a boolean field from the database, it returns 't' or 'f'. If you directly evaluate a variable storing a boolean from a PostgreSQL database, it will always return true.  
  
For example...  
  
<?php  
$x = pg\_query("SELECT someBool FROM atable");  
$x = pg\_fetch\_array($x);  
$x = $x['someBool'];  
  
if ($x) echo "true";  
else echo "false";  
?>  
  
...ALWAYS outputs true

altough it may be obvious to some, special value NaN evaluates to true, as it not in the false list  
  
the same goes with INF and -INF

PHP is very fussy converting strings to booleans. The only ones it recognizes are '0' or '', everything else evaluates to TRUE, even 'false' and '0.0' are evaluated as true! I suppose this can't be fixed without breaking a lot of existing code.  
  
Example:  
  
<?php  
  
print 'yes'."\t".((bool)'yes'? 1: 0)."\n";  
print 'true'."\t".((bool)'true'? 1: 0)."\n";  
print 'no'."\t".((bool)'no'? 1: 0)."\n";  
print 'false'."\t".((bool)'false'? 1: 0)."\n";  
print '1'."\t".((bool)'1'? 1: 0)."\n";  
print '0'."\t".((bool)'0'? 1: 0)."\n";  
print '0.0'."\t".((bool)'0.0'? 1: 0)."\n";  
print ''."\t".((bool)''? 1: 0)."\n";  
  
?>  
  
Output:  
  
yes     1  
true    1  
no      1  
false   1  
1       1  
0       0  
0.0     1  
        0

"always round it downwards"  
  
It seems to truncate, or round toward zero, rather than downward. If the float is negative, it is rounded up.

Be careful with using the modulo operation on big numbers, it will cast a float argument to an int and may return wrong results. For example:  
<?php  
    $i = 6887129852;  
    echo "i=$i\n";  
    echo "i%36=".($i%36)."\n";  
    echo "alternative i%36=".($i-floor($i/36)\*36)."\n";  
?>  
Will output:  
i=6.88713E+009  
i%36=-24  
alternative i%36=20

"The behaviour of converting to integer is undefined for other types. Do not rely on any observed behaviour, as it can change without notice."  
  
actually i use type casting to output Boolean like this:  
  
<?php  
echo (int)(function\_exists('imagecreatetruecolor'));  
?>  
  
so i see 0 or 1 in output

Here are some tricks to convert from a "dotted" IP address to a LONG int, and backwards. This is very useful because accessing an IP addy in a database table is very much faster if it's stored as a BIGINT rather than in characters.  
  
IP to BIGINT:  
<?php  
  $ipArr    = explode('.',$\_SERVER['REMOTE\_ADDR']);  
  $ip       = $ipArr[0] \* 0x1000000  
            + $ipArr[1] \* 0x10000  
            + $ipArr[2] \* 0x100  
            + $ipArr[3]  
            ;  
?>  
  
IP as BIGINT read from db back to dotted form:  
  
Keep in mind, PHP integer operators are INTEGER -- not long. Also, since there is no integer divide in PHP, we save a couple of S-L-O-W floor (<division>)'s by doing bitshifts. We must use floor(/) for $ipArr[0] because though $ipVal is stored as a long value, $ipVal >> 24 will operate on a truncated, integer value of $ipVal! $ipVint is, however, a nice integer, so   
we can enjoy the bitshifts.  
  
<?php  
        $ipVal = $row['client\_IP'];  
        $ipArr = array(0 =>  
                    floor(  $ipVal               / 0x1000000) );  
        $ipVint   = $ipVal-($ipArr[0]\*0x1000000); // for clarity  
        $ipArr[1] = ($ipVint & 0xFF0000)  >> 16;  
        $ipArr[2] = ($ipVint & 0xFF00  )  >> 8;  
        $ipArr[3] =  $ipVint & 0xFF;  
        $ipDotted = implode('.', $ipArr);  
?>

Sometimes you need to parse an unsigned  
32 bit integer. Here's a function I 've used:  
                                                                                  
    function parse\_unsigned\_int($string) {  
        $x = (float)$string;  
        if ($x > (float)2147483647)  
            $x -= (float)"4294967296";  
        return (int)$x;  
    }

To force the correct usage of 32-bit unsigned integer in some functions, just add '+0'  just before processing them.  
  
for example   
echo(dechex("2724838310"));  
will print '7FFFFFFF'  
but it should print 'A269BBA6'  
  
When adding '+0' php will handle the 32bit unsigned integer  
correctly  
echo(dechex("2724838310"+0));  
will print 'A269BBA6'

<?php   
//This is a simple function to return number of digits of an integer.   
  
//function declaration   
function count\_digit($number)   
{   
    $digit = 0;   
    do   
    {   
        $number /= 10;      //$number = $number / 10;   
        $number = intval($number);   
        $digit++;      
    }while($number!=0);   
    return $digit;   
}   
  
//function call   
$num = 12312;   
$number\_of\_digits = count\_digit($num); //this is call :)   
echo $number\_of\_digits;   
//prints 5   
?>

Here are some tricks to convert from a "dotted" IP address to a LONG int, and backwards. This is very useful because accessing an IP addy in a database table is very much faster if it's stored as a BIGINT rather than in characters.  
  
IP to BIGINT:  
<?php  
  $ipArr    = explode('.',$\_SERVER['REMOTE\_ADDR']);  
  $ip       = $ipArr[0] \* 0x1000000  
            + $ipArr[1] \* 0x10000  
            + $ipArr[2] \* 0x100  
            + $ipArr[3]  
            ;  
?>  
  
This can be written in a bit more efficient way:  
<?php  
  $ipArr    = explode('.',$\_SERVER['REMOTE\_ADDR']);  
  $ip       = $ipArr[0]<<24  
            + $ipArr[1]<<16  
            + $ipArr[2] <<8  
            + $ipArr[3]  
            ;  
?>  
  
shift is more cheaper.

Be careful when using integer conversion to test something to see if it evaluates to a positive integer or not.  You might get unexpected behaviour.  
  
To wit:  
<?php  
error\_reporting(E\_ALL);  
require\_once 'Date.php';  
  
$date = new Date();  
print "\$date is an instance of " . get\_class($date) . "\n";  
$date += 0;  
print "\$date is now $date\n";  
var\_dump($date);  
  
$foo = new foo();  
print "\$foo is an instance of " . get\_class($foo) . "\n";  
$foo += 0;  
print "\$foo is now $foo\n";  
var\_dump($foo);  
  
class foo {  
    var $bar = 0;  
    var $baz = "la lal la";  
    var $bak;  
  
    function foo() {  
        $bak = 3.14159;  
    }  
}  
?>  
  
After the integer conversion, you might expect both $foo and $date to evaluate to 0.  However, this is not the case:  
  
$date is an instance of Date  
  
Notice: Object of class Date could not be converted to int in /home/kpeters/work/sketches/ObjectSketch.php on line 7  
$date is now 1  
int(1)  
$foo is an instance of foo  
  
Notice: Object of class foo could not be converted to int in /home/kpeters/work/sketches/ObjectSketch.php on line 13  
$foo is now 1  
int(1)  
  
This is because the objects are first converted to boolean before being converted to int.

Please also note that the maximum stored in the integer depends on the platform / compilation; on windows xp 32 bits, the following value:  
  
0x5468792130ABCDEF  
  
echoes to:  
  
6.0822444802213E+18 (cast to float)  
  
On a fully 64 bits system, it echoes to:  
  
6082244480221302255

A leading zero in a numeric literal means "this is octal". But don't be confused: a leading zero in a string does not. Thus:  
$x = 0123;          // 83  
$y = "0123" + 0     // 123

PHP offers a slew of built-in functions and automatic type-casting routines which can get pretty complicated. But most of the time, you still have to take matters into your own hands and allow PHP to do its thing. In that case, and something that has NOT been mentioned, is how to construct your code. To keep things simple, I divide all my scripts in half. The top half gives my scripts the "capability" they need, and the lower half is the actual code to be "run" or "executed".  
  
<?php  
/\*  
\* build the program's capability - define variables and functions...  
\*/  
$item\_label = '';        // type string  
$item\_price = 0.0;       // type float  
$item\_qty = 1;           // type integer  
$item\_total = 0.0;       // type float - to set use calculate()  
  
function calculate(){  
  global $item\_price, $item\_qty, $item\_total;  
  $item\_price = number\_format($item\_price, 2);  
  $item\_total = number\_format(($item\_price \* $item\_qty), 2);  
}  
  
function itemToString() {  
  global $item\_label, $item\_price, $item\_qty, $item\_total;  
  return "$item\_label [price=\$$item\_price, qty=$item\_qty, total=\$$item\_total]";  
}  
  
/\*  
\* run the program - set data, call methods...  
\*/  
$item\_label = "Coffee";  
$item\_price = 3.89;  
$item\_qty = 2;  
calculate();           // set $item\_total  
echo itemToString();   // -> Coffee [price=$3.89, qty=2, total=$7.78]  
  
$item\_label = "Chicken";  
$item\_price = .80;     // per lb.  
$item\_qty = 3.5;       // lbs.  
calculate();           // set $item\_total  
echo itemToString();   // -> Chicken [price=$0.80, qty=3.5, total=$2.80]  
?>  
Note: All type-casting is done by PHP's built-in number\_format() method. This allows our program to enter any number (float or int) on item price or quantity in the runtime part of our script. Also, if we explicitly cast values to integer in the capability part of our script, then we start getting results that may not be desirable for this program. For example, if in the calculate method we cast item\_qty to integer, then we can no longer sell chicken by the pound!

On 64 bits machines max integer value is 0x7fffffffffffffff (9 223 372 036 854 775 807).

When doing large subtractions on 32 bit unsigned integers the result sometimes end up negative. My example script converts a IPv4 address represented as a 32 bit unsigned integer to a dotted quad (similar to ip2long()), and adds a "fix" to the operation.   
  
   /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
    \* int\_oct($ip)   
    \* Convert INTeger rep of IP to octal (dotted quad)  
    \*/  
   function int\_oct($ip) {  
  
      /\* Set variable to float \*/  
      settype($ip, float);  
  
      /\* FIX for silly PHP integer syndrome \*/  
      $fix = 0;  
      if($ip > 2147483647) $fix = 16777216;  
  
      if(is\_numeric($ip)) {  
         return(sprintf("%u.%u.%u.%u",  
                $ip / 16777216,  
                (($ip % 16777216) + $fix) / 65536,  
                (($ip % 65536) + $fix / 256) / 256,  
                ($ip % 256) + $fix / 256 / 256  
                )  
     );  
      }  
      else {  
         return('');  
      }  
   }

//This is a (simpler ?) function to return number of digits of an integer.  
  
//function declaration  
function count\_digit($number) {  
  return  strlen((string) $number);  
}  
  
//function call  
$num = 12312;  
$number\_of\_digits = count\_digit($num); //this is call :)  
echo $number\_of\_digits;   
//prints 5

Try this one instead:   
  
function iplongtostring($ip)  
{  
    $ip=floatval($ip); // otherwise it is capped at 127.255.255.255  
  
    $a=($ip>>24)&255;  
    $b=($ip>>16)&255;  
    $c=($ip>>8)&255;  
    $d=$ip&255;  
  
    return "$a.$b.$c.$d";  
}

In response to the comment by me at troyswanson dot net:  
  
-2147483648 falls into the range of 32 bit signed integers yet php treats it as a float.  However, -2147483647-1 is treated as an integer.  
  
The following code demonstrates:  
<?php  
    var\_dump(-2147483648); //float(-2147483648)  
    var\_dump(-2147483647 - 1); //int(-2147483648)  
?>  
  
This is probably very similar to the MS C bug which also treats -2147483648 as an UNSIGNED because it thinks it's out of the range of a signed int.  
  
The problem is that the parser does not view "-x" as a single token, but rather as two, "-" and "x".  Since "x" is out of the range of an INT, it is promoted to float, even though in this unique case, "-x" is in the range of an int.  
  
The best cure is probably to replace "-2147483648" with "0x80000000", as that is the hexadecimal equivalent of the same number.  
  
Hope that helps explain what's going on  
  
Peace  
  
- Eric / fez

A note about converting IP addresses for storage in database.  For MySQL, this is unnecessary as it has built in support via the INET functions.  Also, there is no need to use BIGINT.  UNSIGNED INT is, at 4 bytes, the perfect size for holding an IP (column must be defined as UNSIGNED).  This can basically halve the storage size, as BIGINT is an 8 byte data type.  
  
INET\_ATON() converts a dotted IP string to INT:  
INSERT table(ip) VALUES(INET\_ATON('127.0.0.1'));  
  
INET\_NTOA() converts an INT to dotted IP string:  
SELECT INET\_NTOA(ip) FROM table  
returns '127.0.0.1'

PHP\_INT\_SIZE seems to be 8 when it is 64 bit integers... so 8 means the number of bytes, or number of 8-bits.

You can make a signed, negative integer an unsigned integer (in string form) by doing the following:  
  
<?php  
$unsigned = sprintf('%u', -5);  
  
echo $unsigned; // prints 4294967291  
?>

If you need to convert a numeric string (or more to the point, an object that represents a numeric value) that is greater then PHP\_INT\_MAX, and you don't have GMP or BCMath installed, you can cast to float.  
  
For example, when using SimpleXMLElement, you sometimes have to cast the extracted values, such as xml attributes, because they are returned as SimpleXMLElements and not their values' native types. While print() has no trouble with converting them, other functions, such as max(), might not.  
  
But if you cast such a value with (int), and it is over PHP\_INT\_MAX, you will just get PHP\_INT\_MAX (and vice versa for negative numbers).   
  
The Q&D no-muss solution is to cast to (float) instead.

Integer arithmetic in PHP is more accurate than one might think. On a 32-bit system, the largest value that can be held in an INT is  2147483647.  
However, a FLOAT can accurately hold integer values up to 10000000000000.  
(this is because the significand precision of a double is 53-bits).

just a comment on something the "Floating point precision" inset, which goes: "This is related to .... 0.3333333."  
  
While the author probably knows what they are talking about, this loss of precision has nothing to do with decimal notation, it has to do with representation as a floating-point binary in a finite register, such as while 0.8 terminates in decimal, it is the repeating 0.110011001100... in binary, which is truncated.  0.1 and 0.7 are also non-terminating in binary, so they are also truncated, and the sum of these truncated numbers does not add up to the truncated binary representation of 0.8 (which is why (floor)(0.8\*10) yields a different, more intuitive, result).  However, since 2 is a factor of 10, any number that terminates in binary also terminates in decimal.

$x = 8 - 6.4;  // which is equal to 1.6  
$y = 1.6;  
var\_dump($x == $y); // is not true  
  
PHP thinks that 1.6 (coming from a difference) is not equal to 1.6. To make it work, use round()  
  
var\_dump(round($x, 2) == round($y, 2)); // this is true  
  
This happens probably because $x is not really 1.6, but 1.599999.. and var\_dump shows it to you as being 1.6.

In some cases you may want to get the maximum value for a float without getting "INF".  
  
var\_dump(1.8e308); will usually show: float(INF)  
  
I wrote a tiny function that will iterate in order to find the biggest non-infinite float value. It comes with a configurable multiplicator and affine values so you can share more CPU to get a more accurate estimate.  
  
I haven't seen better values with more affine, but well, the possibility is here so if you really thing it's worth the cpu time, just try to affine more.  
  
Best results seems to be with mul=2/affine=1. You can play with the values and see what you get. The good thing is this method will work on any system.  
  
<?php  
  function float\_max($mul = 2, $affine = 1) {  
    $max = 1; $omax = 0;  
    while((string)$max != 'INF') { $omax = $max; $max \*= $mul; }  
  
    for($i = 0; $i < $affine; $i++) {  
      $pmax = 1; $max = $omax;  
      while((string)$max != 'INF') {  
        $omax = $max;  
        $max += $pmax;  
        $pmax \*= $mul;  
      }  
    }  
    return $omax;  
  }  
?>

<?php  
   $binarydata32 = pack('H\*','00000000');  
   $float32 = unpack("f", $binarydata32); // 0.0  
  
   $binarydata64 = pack('H\*','0000000000000000');  
   $float64 = unpack("d", $binarydata64); // 0.0  
?>  
  
I get 0 both for 32-bit and 64-bit numbers.  
  
But, please don't use your own "functions" to "convert" from float to binary and vice versa. Looping performance in PHP is horrible. Using pack/unpack you use processor's encoding, which is always correct. In C++ you can access the same 32/64 data as either float/double or 32/64 bit integer. No "conversions".  
  
To get binary encoding:  
<?php  
   $float32 = pack("f", 5300231);  
   $binarydata32 =unpack('H\*',$float32); //"0EC0A14A"  
  
   $float64 = pack("d", 5300231);  
   $binarydata64 =unpack('H\*',$float64); //"000000C001385441"  
?>  
  
And my example from half a year ago:  
<?php  
    $binarydata32 = pack('H\*','0EC0A14A');  
    $float32 = unpack("f", $binarydata32); // 5300231  
     
    $binarydata64 = pack('H\*','000000C001385441');  
    $float64 = unpack("d", $binarydata64); // 5300231  
?>  
  
And please mind the Big and Little endian boys...

(19.6\*100) != 1960    
  
echo gettype(19.6\*100) returns 'double', However even .....   
  
(19.6\*100) !== (double)1960   
  
19.6\*100 cannot be compaired to anything without manually   
casting it as something else first.   
  
(string)(19.6\*100) == 1960  
  
Rule of thumb, if it has a decimal point, use the BCMath functions.

Convert a hex string into a 32-bit IEEE 754 float number.  This function is 2 times faster then the below hex to 32bit function.  This function only changes datatypes (string to int) once. Also, this function is a port from the hex to 64bit function from below.  
  
<?php  
function hexTo32Float($strHex) {  
    $v = hexdec($strHex);  
    $x = ($v & ((1 << 23) - 1)) + (1 << 23) \* ($v >> 31 | 1);  
    $exp = ($v >> 23 & 0xFF) - 127;  
    return $x \* pow(2, $exp - 23);  
}  
?>  
  
<?php  
//example  
echo hexTo32Float("C4028000"); // outputs: -522  
echo hexTo32Float("457F9000"); // outputs: 4089  
echo hexTo32Float("2D7F5");    // outputs: 6.00804264307E-39  
echo hexTo32Float("0002D7F5"); // outputs: 6.00804264307E-39  
echo hexTo32Float("47D9F95E"); // outputs: 111602.734375  
?>

The function returns 5 for 5,000 because if there is no decimal point, then the first strpos will be FALSE, and FALSE < 1 is TRUE so the condition will be still true.  
  
It should be checked whether strpos returns a valid position:  
  
<?php  
function str2num($str)  
{   
      if (strpos($str, '.') !== FALSE && strpos($str,    ',') !== FALSE && strpos($str, '.') < strpos($str,','))  
          {   
            $str = str\_replace('.','',$str);   
            $str = strtr($str,',','.');              
        }   
        else  
        {   
            $str = str\_replace(',','',$str);              
        }   
          
        return (float)$str;   
}  
?>

<?php  
/\*\* hex2float  
\* (Convert 8 digit hexadecimal value to float (single-precision 32bits)   
\* Accepts 8 digit hexadecimal values in a string  
\* @usage:   
\* hex2float32n("429241f0"); returns -> "73.128784179688"   
\* \*/  
function hex2float($number) {  
    $binfinal = sprintf("%032b",hexdec($number));  
    $sign = substr($binfinal, 0, 1);  
    $exp = substr($binfinal, 1, 8);  
    $mantissa = "1".substr($binfinal, 9);  
    $mantissa = str\_split($mantissa);  
    $exp = bindec($exp)-127;  
    $significand=0;  
    for ($i = 0; $i < 24; $i++) {  
        $significand += (1 / pow(2,$i))\*$mantissa[$i];  
    }  
    return $significand \* pow(2,$exp) \* ($sign\*-2+1);  
}  
?>

I'd like to point out a "feature" of PHP's floating point support that isn't made clear anywhere here, and was driving me insane.   
  
This test (where var\_dump says that $a=0.1 and $b=0.1)   
  
if ($a>=$b) echo "blah!";   
  
Will fail in some cases due to hidden precision (standard C problem, that PHP docs make no mention of, so I assumed they had gotten rid of it). I should point out that I originally thought this was an issue with the floats being stored as strings, so I forced them to be floats and they still didn't get evaluated properly (probably 2 different problems there).   
  
To fix, I had to do this horrible kludge (the equivelant of anyway):   
  
if (round($a,3)>=round($b,3)) echo "blah!";   
  
THIS works. Obviously even though var\_dump says the variables are identical, and they SHOULD BE identical (started at 0.01 and added 0.001 repeatedly), they're not. There's some hidden precision there that was making me tear my hair out. Perhaps this should be added to the documentation?

In MySQL, many floating point number types can have a range specified using 2 values, the "precision" and the "scale" E.g. 'float(precision,scale)' for the datatype. This syntax means a number may be <precision> bits long, but may only have <scale> bits after the decimal point. E.g. a 'float(5,2)' field may have the values -999.99 to 999.99.  
Here is a function to validate a PHP float using this syntax:  
<?php  
function validate\_float($float, $precision, $scale)  
{  
    $max = (float)str\_pad("", $precision-$scale, '9').'.'.str\_pad("", $scale, '9');  
    $min = (float)"-$max";  
  
    if(($float < $min) || ($float > $max)) return false;  
    else return true;  
}  
?>

General computing hint: If you're keeping track of money, do yourself and your users the favor of handling everything internally in cents and do as much math as you can in integers. Store values in cents if at all possible. Add and subtract in cents. At every operation that wii involve floats, ask yourself "what will happen in the real world if I get a fraction of a cent here" and if the answer is that this operation will generate a transaction in integer cents, do not try to carry fictional fractional accuracy that will only screw things up later.

PHP switches from the standard decimal notation to exponential notation for certain "special" floats. You can see a partial list of such "special" values with this:  
  
<?php  
for( $tmp = 0, $i = 0; $i < 100; $i++ ) {  
    $tmp += 100000;  
    echo round($tmp),"\n";  
}  
?>  
  
So, if you add two floats, end up with a "special" value, e.g. 1.2E+6, then put that value unmodified into an update query to store the value in a decimal column, say, you will likely get a failed transaction, since the database will see "1.2E+6" as varchar data, not decimal. Likewise, you will likely get an XSD validation error if you put the value into xml.  
  
I have to be honest: this is one of the strangest things I have seen in any language in over 20 years of coding, and it is a colossal pain to work around.

Calculations involving float types become inaccurate when it deals with numbers with more than approximately 8 digits long where ever the decimal point is.  This is because of how 32bit floats are commonly stored in memory.  This means if you rely on float types while working with tiny fractions or large numbers, your calculations can end up between tiny fractions to several trillion off.  
  
This usually won't matter when converting to binary memory storage form and editing many applications' float memory addresses directly, or dealing with smaller length numbers.  But if you're working with larger scale numbers and decimals, it's best to switch to working with other types: <http://www.php.net/manual/en/refs.math.php>

Just another note about the locales. Consider the following code:  
  
<?php   
    // in polish locale decimal separator is ","  
    setlocale(LC\_ALL, "pl\_PL");  
    $a = 5/2;  
    echo (float)(string)$a;  
    /// prints "2", so the decimal part is dropped  
?>  
  
This causes very serious problems in my opinion. In some locale combination the typecasting can be destructive.  
Maybe when locale decimal separator is ",", then (float)"2,5" should be recognized as "two and a half"?   
Anyway - bare that in mind and be very careful when casting floats to strings and back.

Be careful when using float values in strings that are used as code later, for example when generating JavaScript code or SQL statements. The float is actually formatted according to the browser's locale setting, which means that "0.23" will result in "0,23". Imagine something like this:  
  
$x = 0.23;  
$js = "var foo = doBar($x);";  
print $js;  
  
This would result in a different result for users with some locales. On most systems, this would print:  
  
var foo = doBar(0.23);  
  
but when for example a user from Germany arrives, it would be different:  
  
var foo = doBar(0,23);  
  
which is obviously a different call to the function. JavaScript won't state an error, additional arguments are discarded without notice, but the function doBar(a) would get 0 as parameter. Similar problems could arise anywhere else (SQL, any string used as code somewhere else). The problem persists, if you use the "." operator instead of evaluating the variable in the string.  
  
So if you REALLY need to be sure to have the string correctly formatted, use number\_format() to do it!

I was programming an accounting application in MySql that required me to sum a collection of floats and ensure that they equal zero before commiting a transaction, but as seen above a sum of floats cannot always be trusted (as was my case).  I kept getting a very small remainder (like 1.4512431231e-14).  Since I had used number\_format(num,2) to set the precision of the numbers in the database to only two (2) decimal places, when the time comes to calculate the sum I simply multiply every number by ten (10), therby eliminating and decimal places and leaving me with integers to preform my sum.  This worked great.

Floating point values have a limited precision. Hence a value might not have the same string representation after any processing. That also includes writing a floating point value in your script and directly printing it without any mathematical operations.  
  
If you would like to know more about "floats" and what IEEE 754 is read this: <http://docs.sun.com/source/806-3568/ncg_goldberg.html>

The 'floating point precision' box in practice means:  
  
<? echo (69.1-floor(69.1)); ?>  
Think this'll return 0.1?  
It doesn't - it returns 0.099999999999994  
  
<? echo round((69.1-floor(69.1))); ?>  
This returns 0.1 and is the workaround we use.  
  
Note that  
<? echo (4.1-floor(4.1)); ?>  
\*does\* return 0.1 - so if you, like us, test this with low numbers, you won't, like us, understand why all of a sudden your script stops working, until you spend a lot of time, like us, debugging it.  
  
So, that's all lovely then.

The was talk about "converting" 32 and 64 bit IEEE754 binary numbers to PHP float. The issue isn't as much converting, since they are already in binary form, as it is casting. PHP doesn't allow direct accessing of memory, but you can still get around a bit.  
  
The right was to read floats (32 and 64 bit) is this:  
  
<?php  
    $binarydata32 = pack('H\*','0EC0A14A');  
    $float32 = unpack("f", $binarydata32);  
      
    $binarydata64 = pack('H\*','000000C001385441');  
    $float64 = unpack("d", $binarydata64);  
      
    var\_dump($float32,$float64,$float32==$float64);     
?>  
  
The result of dump():  
<?php  
array(1) {  
  [1]=>  
  float(5300231)  
}  
array(1) {  
  [1]=>  
  float(5300231)  
}  
bool(true)  
?>  
  
Note: mind the Big and Little endian boys

convert 32bit HEX values into IEEE 754 floating point   
<?php   
  
$strHex = "C45F82ED";   
  
$bin = str\_pad(base\_convert($strHex, 16, 2), 32, "0", STR\_PAD\_LEFT);   
$sign = $bin[0];   
$exp = bindec(substr($bin, 1, 8)) - 127;   
$man = (2 << 22) + bindec(substr($bin, 9, 23));   
  
$dec = $man \* pow(2, $exp - 23) \* ($sign ? -1 : 1);   
  
echo "Answer = " . $dec . "<BR>\n";   
  
?>

Here is a function to convert an exponential-format float to a decimal-format float; e.g. 1.6e+12 to 1600000000000.  
It will help addressing the problem specified by kjohnson above.  
I have tested it, but not in any real world situation so any feedback/improvements/bug-reports would be appreciated.  
<?php  
function exp\_to\_dec($float\_str)  
// formats a floating point number string in decimal notation, supports signed floats, also supports non-standard formatting e.g. 0.2e+2 for 20  
// e.g. '1.6E+6' to '1600000', '-4.566e-12' to '-0.000000000004566', '+34e+10' to '340000000000'  
// Author: Bob  
{  
    // make sure its a standard php float string (i.e. change 0.2e+2 to 20)  
    // php will automatically format floats decimally if they are within a certain range  
    $float\_str = (string)((float)($float\_str));  
  
    // if there is an E in the float string  
    if(($pos = strpos(strtolower($float\_str), 'e')) !== false)  
    {  
        // get either side of the E, e.g. 1.6E+6 => exp E+6, num 1.6  
        $exp = substr($float\_str, $pos+1);  
        $num = substr($float\_str, 0, $pos);  
          
        // strip off num sign, if there is one, and leave it off if its + (not required)  
        if((($num\_sign = $num[0]) === '+') || ($num\_sign === '-')) $num = substr($num, 1);  
        else $num\_sign = '';  
        if($num\_sign === '+') $num\_sign = '';  
          
        // strip off exponential sign ('+' or '-' as in 'E+6') if there is one, otherwise throw error, e.g. E+6 => '+'  
        if((($exp\_sign = $exp[0]) === '+') || ($exp\_sign === '-')) $exp = substr($exp, 1);  
        else trigger\_error("Could not convert exponential notation to decimal notation: invalid float string '$float\_str'", E\_USER\_ERROR);  
          
        // get the number of decimal places to the right of the decimal point (or 0 if there is no dec point), e.g., 1.6 => 1  
        $right\_dec\_places = (($dec\_pos = strpos($num, '.')) === false) ? 0 : strlen(substr($num, $dec\_pos+1));  
        // get the number of decimal places to the left of the decimal point (or the length of the entire num if there is no dec point), e.g. 1.6 => 1  
        $left\_dec\_places = ($dec\_pos === false) ? strlen($num) : strlen(substr($num, 0, $dec\_pos));  
          
        // work out number of zeros from exp, exp sign and dec places, e.g. exp 6, exp sign +, dec places 1 => num zeros 5  
        if($exp\_sign === '+') $num\_zeros = $exp - $right\_dec\_places;  
        else $num\_zeros = $exp - $left\_dec\_places;  
          
        // build a string with $num\_zeros zeros, e.g. '0' 5 times => '00000'  
        $zeros = str\_pad('', $num\_zeros, '0');  
          
        // strip decimal from num, e.g. 1.6 => 16  
        if($dec\_pos !== false) $num = str\_replace('.', '', $num);  
          
        // if positive exponent, return like 1600000  
        if($exp\_sign === '+') return $num\_sign.$num.$zeros;  
        // if negative exponent, return like 0.0000016  
        else return $num\_sign.'0.'.$zeros.$num;  
    }  
    // otherwise, assume already in decimal notation and return  
    else return $float\_str;  
}  
?>

PHP will parse ".123" with no leading digit; just a decimal point. For a command-line example:  
  
php -r "echo 1 + .123;"  
  
The regular expression provided above does not parse it.  
My correction is:  
  
EXPONENT\_DNUM = "[+-]?({DNUM} | ({LNUM} | {DNUM}) [eE][+-]? {LNUM})"  
  
NOTE: {LNUM} by itself is an integer, not a floating point.

To simply convert 32 bits float from hex to float:   
  
<?php   
function hexfloat ($hex){   
    return (unpack("f", pack('H\*',$hex))[1]);   
}   
?>   
  
This may be useful for arduino interface with php.

When converting from float to string trailing zeros will be dropped. Consider the following example. Tested on PHP 5.3.1.  
  
<?php  
$a = 5.00500;  
$b = 30.00;  
echo "a = $a\n";  
echo "b = $b\n";  
/\* outputs:  
a = 5.005  
b = 30  
\*/  
?>

My BIN to FLOAT (IEEE754), the first one doesn't work for me:   
  
<?php   
        function binToFloat($bin) {   
            if(strlen($bin) > 32) {   
                return false;   
            } else if(strlen($bin) < 32) {   
                $bin = str\_repeat('0', (32 - strlen($bin))) . $bin;   
            }   
  
            $sign = 1;   
            if(intval($bin[0]) == 1) {   
                $sign = -1;   
            }   
  
            $binExponent = substr($bin, 1, 8);   
            $exponent = -127;   
            for($i = 0; $i < 8; $i++) {   
                $exponent += (intval($binExponent[7 - $i]) \* pow(2, $i));   
            }   
  
            $binBase = substr($bin, 9);              
            $base = 1.0;   
            for($x = 0; $x < 23; $x++) {   
                $base += (intval($binBase[$x]) \* pow(0.5, ($x + 1)));   
            }   
  
            $float = (float) $sign \* pow(2, $exponent) \* $base;   
  
            return $float;   
        }   
?>

I found that 00000000 hex was converting to 1.0 decimal. From the Wikipedia article on IEEE-754 floating point:  
  
The true significand includes 23 fraction bits to the right of the binary point and an implicit leading bit (to the left of the binary point) with value 1 unless the exponent is stored with all zeros.  
  
In hex2float32n, replace:  
  
      $intnumber=bindec("1".$binint);  
  
with  
  
   if ($exp <> -127)  
      { $intnumber=bindec("1".$binint); };  
  
and then 00000000 works correctly without affecting "normal" numbers.

Here is a simple formula to break down a number and get rid of the decimal values.  I built this to take a number in seconds and convert it to a readable value for Server Uptimes.  
  
<?php  
$day = floor(($uptime / 86400)\*1.0) ;  
$calc1 = $day \* 86400 ;  
$calc2 = $uptime - $calc1 ;  
$hour = floor(($calc2 / 3600)\*1.0) ;  
if ($hour < 10) {  
$hour = "0".$hour ;  
}  
$calc3 = $hour \* 3600 ;  
$calc4 = $calc2 - $calc3 ;  
$min = floor(($calc4 / 60)\*1.0) ;  
if ($min < 10) {  
$min = "0".$min ;  
}  
$calc5 = $min \* 60 ;  
$sec = floor(($calc4 - $calc5)\*1.0) ;  
if ($min < 10) {  
$sec = "0".$sec ;  
}  
$uptime2 = $day." Days, ".$hour.":".$min.":".$sec ;  
?>  
  
Place this where you want the results to be seen:  
<?php echo $uptime2 ; ?>  
  
For a Value of 1455587 seconds the results will show as followed:  
16 Days, 20:19:47  
  
Enjoy

As "m dot lebkowski+php at gmail dot com" (<http://www.php.net/language.types.float#81416>) noted 9 comments below :  
  
When PHP converts a float to a string, the decimal separator used depends on the current locale conventions.  
  
However, to declare a floating point number, one must always use a full stop otherwhise the code would be locale dependent (imagine the nightmare):  
<?php  
$float = 1.5;           // float(1.5)  
$float = 1,5;           // Parse error: syntax error, unexpected ','  
$float = (float) '1.5'; // float(1.5)  
$float = (float) '1,5'; // float(1)  
?>  
  
Now, if you have a string containing a localized number, you can convert it back to a floating point number using the following function:  
<?php  
/\*\*  
\* Convert a localized number string into a floating point number  
\*  
\* @param      string $sNumber The localized number string to convert.  
\* @return     float The converted number.  
\*/  
function str2num($sNumber)  
{  
    $aConventions = localeConv();  
    $sNumber = trim((string) $sNumber);  
    $bIsNegative = (0 === $aConventions['n\_sign\_posn'] && '(' === $sNumber{0} && ')' === $sNumber{strlen($sNumber) - 1});  
    $sCharacters = $aConventions['decimal\_point'].  
                   $aConventions['mon\_decimal\_point'].  
                   $aConventions['negative\_sign'];  
    $sNumber = preg\_replace('/[^'.preg\_quote($sCharacters).'\d]+/', '', trim((string) $sNumber));  
    $iLength = strlen($sNumber);  
    if (strlen($aConventions['decimal\_point']))  
    {  
        $sNumber = str\_replace($aConventions['decimal\_point'], '.', $sNumber);  
    }  
    if (strlen($aConventions['mon\_decimal\_point']))  
    {  
        $sNumber = str\_replace($aConventions['mon\_decimal\_point'], '.', $sNumber);  
    }  
    $sNegativeSign = $aConventions['negative\_sign'];  
    if (strlen($sNegativeSign) && 0 !== $aConventions['n\_sign\_posn'])  
    {  
        $bIsNegative = ($sNegativeSign === $sNumber{0} || $sNegativeSign === $sNumber{$iLength - 1});  
        if ($bIsNegative)  
        {  
            $sNumber = str\_replace($aConventions['negative\_sign'], '', $sNumber);  
        }  
    }  
    $fNumber = (float) $sNumber;  
    if ($bIsNegative)  
    {  
        $fNumber = -$fNumber;  
    }  
    return $fNumber;  
}  
?>  
  
Example:  
<?php  
setLocale(LC\_ALL, 'fr\_BE.UTF-8'); // decimal separator is now a comma  
$float = -123456.789;  
$string = (string) $float;  
var\_dump($float);           // float(-123456,789)  
var\_dump($string);          // string(11) "-123456,789"  
var\_dump((float) $string);  // float(-123456)  
var\_dump(str2num($string)); // float(-123456,789)  
?>  
  
It also works with strings returned by the number\_format() function:  
<?php  
setLocale(LC\_ALL, 'fr\_BE.UTF-8'); // decimal separator is now a comma  
$conv = localeconv();  
$float = -123456.789;  
$string = $conv['int\_curr\_symbol'].number\_format($float, $conv['frac\_digits'], $conv['decimal\_point'], $conv['thousands\_sep']);  
var\_dump($float);           // float(-123456,789)  
var\_dump($string);          // string(15) "EUR -123.456,79"  
var\_dump((float) $string);  // float(0)  
var\_dump(str2num($string)); // float(-123456,79)  
?>

Convert locale string into float number   
  
<?php   
function str2num($str){   
  if(strpos($str, '.') < strpos($str,',')){   
            $str = str\_replace('.','',$str);   
            $str = strtr($str,',','.');              
        }   
        else{   
            $str = str\_replace(',','',$str);              
        }   
        return (float)$str;   
}   
  
str2num('25,01'); //25.01   
str2num('2.5,01'); //25.01   
str2num('25.01'); //25.01   
str2num('2,5.01'); //25.01   
?>

The documentation does not mention, but a closing semicolon at the end of the heredoc is actually interpreted as a real semicolon, and as such, sometimes leads to syntax errors.  
  
This works:  
  
<?php  
$foo = <<<END  
abcd  
END;  
?>  
  
This does not:  
  
<?php  
foo(<<<END  
abcd  
END;  
);  
// syntax error, unexpected ';'  
?>  
  
Without semicolon, it works fine:  
  
<?php  
foo(<<<END  
abcd  
END  
);  
?>

To save Your mind don't read previous comments about dates  ;)  
  
When both strings can be converted to the numerics (in ("$a" > "$b") test) then resulted numerics are used, else FULL strings are compared char-by-char:  
  
<?php  
var\_dump('1.22' > '01.23'); // bool(false)  
var\_dump('1.22.00' > '01.23.00'); // bool(true)  
var\_dump('1-22-00' > '01-23-00'); // bool(true)  
var\_dump((float)'1.22.00' > (float)'01.23.00'); // bool(false)  
?>

Here is a possible gotcha related to oddness involved with accessing strings by character past the end of the string:   
  
$string = 'a';   
  
var\_dump($string[2]);  // string(0) ""   
var\_dump($string[7]);  // string(0) ""   
$string[7] === '';  // TRUE   
  
It appears that anything past the end of the string gives an empty string..  However, when E\_NOTICE is on, the above examples will throw the message:   
  
Notice:  Uninitialized string offset:  N in FILE on line LINE   
  
This message cannot be specifically masked with @$string[7], as is possible when $string itself is unset.   
  
isset($string[7]);  // FALSE   
$string[7] === NULL;  // FALSE   
  
Even though it seems like a not-NULL value of type string, it is still considered unset.

String conversion to numbers.  
  
Unfortunately, the documentation is not correct.  
  
«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).»  
  
It is not said and is not shown in examples throughout the documentation that, while converting strings to numbers, leading space characters are ignored, like with the strtod function.  
  
<?php  
    echo "     \v\f    \r   1234" + 1;    // 1235  
    var\_export ("\v\f    \r   1234" == "1234");    // true  
?>  
  
However, PHP's behaviour differs even from the strtod's. The documentation says that if the string contains a "e" or "E" character, it will be parsed as a float, and suggests to see the manual for strtod for more information. The manual says  
  
«A hexadecimal number consists of a "0x" or "0X" followed by a nonempty sequence of hexadecimal digits possibly containing a radix character, optionally followed by a binary exponent.  A binary exponent consists of a 'P' or 'p', followed by an optional plus or minus sign, followed by a nonempty sequence of decimal digits, and indicates multiplication by a power of 2.»  
  
But it seems that PHP does not recognise the exponent or the radix character.  
  
<?php  
    echo "0xEp4" + 1;     // 15  
?>  
  
strtod also uses the current locale to choose the radix character, but PHP ignores the locale, and the radix character is always 2E. However, PHP uses the locale while converting numbers to strings.  
  
With strtod, the current locale is also used to choose the space characters, I don't know about PHP.

Although current documentation says 'A string literal can be specified in four different ways: ...', actually there is a fifth way to specify a (binary) string:   
  
<?php $binary = b'This is a binary string'; ?>  
  
The above statement declares a binary string using the 'b' prefix, which is available since PHP 5.2.1. However, it will only have effect as of PHP 6.0.0, as noted on <http://www.php.net/manual/en/function.is-binary.php> .

I recently discovered the joys of using heredoc with sprintf and positions. Useful if you want some code to iterate, you can repeat placeholders.   
  
<?php   
  
function getNumber($num = 0) {   
    $foo = rand(1,20);   
    return ($foo + $num);   
}   
function getString() {   
    $foo = array("California","Oregon","Washington");   
    shuffle($foo);   
    return $foo[0];   
}   
function getDiv() {   
    $num = getNumber();   
    $div = sprintf( "<div>%s</div>", getNumber(rand(-5,5)) );   
    return $div;   
}   
$string = <<<THESTRING   
I like the state of %1\$s <br />   
I picked: %2\$d as a number, <br />   
I also picked %2\$d as a number again <br />   
%3\$s<br />   
%3\$s<br />   
%3\$s<br />   
%3\$s<br />   
%3\$s<br />   
THESTRING;   
  
$returnText = sprintf(  $string, getString(),getNumber(),getDiv()  );   
  
echo $returnText;   
  
?>   
  
Expected output of the above code:   
  
I like the state of Oregon   
I picked: 15 as a number,   
I also picked 15 as a number again   
5   
  
5   
  
5   
  
5   
  
5

Leading zeroes in strings are (least-surprise) not treated as octal.  
Consider:  
  $x = "0123"  + 0;     
  $y = 0123 + 0;  
  echo "x is $x, y is $y";    //prints  "x is 123, y is 83"  
in other words:  
\* leading zeros in numeric literals in the source-code are interpreted as "octal", c.f. strtol().  
\* leading zeros in strings (eg user-submitted data), when cast (implicitly or explicitly) to integer are ignored, and considered as decimal, c.f. strtod().

easy transparent solution for using constants in the heredoc format:  
DEFINE('TEST','TEST STRING');  
  
$const = get\_defined\_constants();  
  
echo <<<END  
{$const['TEST']}  
END;  
  
Result:  
TEST STRING

Heredoc literals delete any trailing space (tabs and blanks) on each line. This is unexpected, since quoted strings do not do this. This is probably done for historical reasons, so would not be considered a bug.

If you want to use a variable in an array index within a double quoted string you have to realize that when you put the curly braces around the array, everything inside the curly braces gets evaluated as if it were outside a string.  Here are some examples:  
  
<?php  
$i = 0;  
$myArray[Person0] = Bob;  
$myArray[Person1] = George;  
  
// prints Bob (the ++ is used to emphasize that the expression inside the {} is really being evaluated.)  
echo "{$myArray['Person'.$i++]}<br>";  
  
// these print George  
echo "{$myArray['Person'.$i]}<br>";  
echo "{$myArray["Person{$i}"]}<br>";  
  
// These don't work  
echo "{$myArray['Person$i']}<br>";  
echo "{$myArray['Person'$i]}<br>";  
  
// These both throw fatal errors  
// echo "$myArray[Person$i]<br>";  
//echo "$myArray[Person{$i}]<br>";  
?>

Here is an easy hack to allow double-quoted strings and heredocs to contain arbitrary expressions in curly braces syntax, including constants and other function calls:  
  
<?php  
  
// Hack declaration  
function \_expr($v) { return $v; }  
$\_expr = '\_expr';  
  
// Our playground  
define('qwe', 'asd');  
define('zxc', 5);  
  
$a=3;  
$b=4;  
  
function c($a, $b) { return $a+$b; }  
  
// Usage  
echo "pre {$\_expr(1+2)} post\n"; // outputs 'pre 3 post'  
echo "pre {$\_expr(qwe)} post\n"; // outputs 'pre asd post'  
echo "pre {$\_expr(c($a, $b)+zxc\*2)} post\n"; // outputs 'pre 17 post'  
  
// General syntax is {$\_expr(...)}  
?>

You can use the complex syntax to put the value of both object properties AND object methods inside a string.  For example...  
<?php  
class Test {  
    public $one = 1;  
    public function two() {  
        return 2;  
    }  
}  
$test = new Test();  
echo "foo {$test->one} bar {$test->two()}";  
?>  
Will output "foo 1 bar 2".  
  
However, you cannot do this for all values in your namespace.  Class constants and static properties/methods will not work because the complex syntax looks for the '$'.  
<?php  
class Test {  
    const ONE = 1;  
}  
echo "foo {Test::ONE} bar";  
?>  
This will output "foo {Test::one} bar".  Constants and static properties require you to break up the string.

In Example #8, above, consider the risk to the script if a programmer were to define('koolaid1', 'XYZ');  For this reason it's wise to use quotes around literal-string associative array keys.  As written without quotes, PHP should raise a Notice.

So you want to get the last character of a string using "String access and modification by character"?  Well negative indexes are not allowed so $str[-1] will return an empty string.  
  
<?php  
//Tested using: PHP 5.2.5  
  
$str = 'This is a test.';  
  
$last = $str[-1];                  //string(0) ""  
$realLast = $str[strlen($str)-1];  //string(1) "."  
$substr = substr($str,-1);         //string(1) "."  
  
echo '<pre>';  
var\_dump($last);  
var\_dump($realLast);  
var\_dump($substr);

Use caution when you need white space at the end of a heredoc. Not only is the mandatory final newline before the terminating symbol stripped, but an immediately preceding newline or space character is also stripped.  
  
For example, in the following, the final space character (indicated by \s -- that is, the "\s" is not literally in the text, but is only used to indicate the space character) is stripped:  
  
$string = <<<EOT  
this is a string with a terminating space\s  
EOT;  
  
In the following, there will only be a single newline at the end of the string, even though two are shown in the text:  
  
$string = <<<EOT  
this is a string that must be  
followed by a single newline  
  
EOT;

I commented on a php bug feature request for a string expansion function and figured I should post somewhere it might be useful:  
  
using regex, pretty straightforward:  
<?php  
function stringExpand($subject, array $vars) {  
    // loop over $vars map  
    foreach ($vars as $name => $value) {  
        // use preg\_replace to match ${`$name`} or $`$name`  
        $subject = preg\_replace(sprintf('/\$\{?%s\}?/', $name), $value,  
$subject);  
    }  
    // return variable expanded string  
    return $subject;  
}  
?>  
  
using eval() and not limiting access to only certain variables (entire current symbol table including [super]globals):  
  
<?php  
function stringExpandDangerous($subject, array $vars = array(), $random = true) {  
      
        // extract $vars into current symbol table  
        extract($vars);  
          
        $delim;  
        // if requested to be random (default), generate delim, otherwise use predefined (trivially faster)  
        if ($random)  
            $delim = '\_\_\_' . chr(mt\_rand(65,90)) . chr(mt\_rand(65,90)) . chr(mt\_rand(65,90)) . chr(mt\_rand(65,90)) . chr(mt\_rand(65,90)) . '\_\_\_';  
        else  
            $delim = '\_\_ASDFZXCV1324ZXCV\_\_';  // button mashing...  
          
        // built the eval code  
        $statement = "return <<<$delim\n\n" . $subject . "\n$delim;\n";  
          
        // execute statement, saving output to $result variable  
        $result = eval($statement);  
          
        // if eval() returned FALSE, throw a custom exception  
        if ($result === false)  
            throw new EvalException($statement);  
          
        // return variable expanded string  
        return $result;  
    }  
?>  
  
I hope that helps someone, but I do caution against using the eval() route even if it is tempting.  I don't know if there's ever a truely safe way to use eval() on the web, I'd rather not use it.

If you want a parsed variable surrounded by curly braces, just double the curly braces:   
  
<?php   
  $foo = "bar";   
  echo "{{$foo}}";   
?>   
  
will just show {bar}. The { is special only if followed by the $ sign and matches one }. In this case, that applies only to the inner braces. The outer ones are not escaped and pass through directly.

You may use heredoc syntax to comment out large blocks of code, as follows:  
<?php  
<<<\_EOC  
    // end-of-line comment will be masked... so will regular PHP:  
    echo ($test == 'foo' ? 'bar' : 'baz');   
    /\* c-style comment will be masked, as will other heredocs (not using the same marker) \*/  
    echo <<<EOHTML  
This is text you'll never see!          
EOHTML;  
    function defintion($params) {  
        echo 'foo';  
    }  
    class definition extends nothing     {  
       function definition($param) {  
          echo 'do nothing';  
       }         
    }  
  
    how about syntax errors?; = gone, I bet.  
\_EOC;  
?>  
  
Useful for debugging when C-style just won't do.  Also useful if you wish to embed Perl-like Plain Old Documentation; extraction between POD markers is left as an exercise for the reader.  
  
Note there is a performance penalty for this method, as PHP must still parse and variable substitute the string.

Simple function to create human-readably escaped double-quoted strings for use in source code or when debugging strings with newlines/tabs/etc.  
  
<?php  
function doubleQuote($str) {  
    $ret = '"';  
    for ($i = 0, $l = strlen($str); $i < $l; ++$i) {  
        $o = ord($str[$i]);  
        if ($o < 31 || $o > 126) {  
            switch ($o) {  
                case 9: $ret .= '\t'; break;  
                case 10: $ret .= '\n'; break;  
                case 11: $ret .= '\v'; break;  
                case 12: $ret .= '\f'; break;  
                case 13: $ret .= '\r'; break;  
                default: $ret .= '\x' . str\_pad(dechex($o), 2, '0', STR\_PAD\_LEFT);  
            }  
        } else {  
            switch ($o) {  
                case 36: $ret .= '\$'; break;  
                case 34: $ret .= '\"'; break;  
                case 92: $ret .= '\\\\'; break;  
                default: $ret .= $str[$i];  
            }  
        }  
    }  
    return $ret . '"';  
}  
?>

Unlike bash, we can't do   
  echo "\a"       #beep!  
  
Of course, that would be rather meaningless for PHP/web, but it's useful for PHP-CLI. The solution is simple:  echo "\x07"

As of (at least) PHP 5.2, you can no longer convert an object to a string unless it has a \_\_toString method. Converting an object without this method now gives the error:  
  
PHP Catchable fatal error:  Object of class <classname> could not be converted to string in <file> on line <line>  
  
Try this code to get the same results as before:  
  
<?php  
  
if (!is\_object($value) || method\_exists($value, '\_\_toString')) {  
    $string = (string)$value;  
} else {  
    $string = 'Object';  
}  
  
?>

Watch out for the "unexpected T\_SL" error.  This appears to occur when there is white space just after "<<<EOT" and since it's white space it's real hard to spot the error in your code.

If you need to emulate a nowdoc in PHP < 5.3, try using HTML mode and output capturing. This way '$' or '\n' in your string won't be a problem anymore (but unfortunately, '<?' will be).  
  
<?php  
  
// Start of script  
  
ob\_start(); ?>  
  A text with 'quotes'   
    and $$$dollars$$$.  
<?php $input = ob\_get\_contents(); ob\_end\_clean();  
  
// Do what you want with $input  
echo "<pre>" . $input . "</pre>";  
  
?>

watch out when comparing strings that are numbers. this example:  
  
<?php  
  
$x1 = '111111111111111111';  
$x2 = '111111111111111112';  
  
echo ($x1 == $x2) ? "true\n" : "false\n";  
  
?>  
  
will output "true", although the strings are different. With large integer-strings, it seems that PHP compares only the integer values, not the strings. Even strval() will not work here.  
  
To be on the safe side, use:  
  
$x1 === $x2

Just want to mention that if you want a literal { around a variable within a string, for example if you want your output to be something like the following:  
  
{hello, world}  
  
and all that you put inside the {} is a variable, you can do a double {{}}, like this:  
  
$test = 'hello, world';  
echo "{{$test}}";

I encountered the odd situation of having a string containing unexpanded escape sequences that I wanted to expand, but also contained dollar signs that would be interpolated as variables.  "$5.25\n", for example, where I want to convert \n to a newline, but don't want attempted interpolation of $5.  
  
Some muddling through docs and many obscenties later, I produced the following, which expands escape sequences in an existing string with NO interpolation.  
  
<?php  
  
// where we do all our magic  
function expand\_escape($string) {  
    return preg\_replace\_callback(  
        '/\\\([nrtvf]|[0-7]{1,3}|[0-9A-Fa-f]{1,2})?/',  
        create\_function(  
            '$matches',  
            'return ($matches[0] == "\\\\") ? "" : eval( sprintf(\'return "%s";\', $matches[0]) );'  
        ),  
        $string  
    );  
}  
  
// a string to test, and show the before and after  
$before = 'Quantity:\t500\nPrice:\t$5.25 each';  
$after = expand\_escape($before);  
var\_dump($before, $after);  
  
/\* Outputs:  
string(34) "Quantity:\t500\nPrice:\t$5.25 each"  
string(31) "Quantity:    500  
Price:    $5.25 each"  
\*/  
  
?>

Just some quick observations on variable interpolation:  
  
Because PHP looks for {? to start a complex variable expression in a double-quoted string, you can call object methods, but not class methods or unbound functions.  
  
This works:  
  
<?php  
class a {  
    function b() {  
        return "World";  
    }  
}  
$c = new a;  
echo "Hello {$c->b()}.\n"  
?>  
  
While this does not:  
  
<?php  
function b() {  
    return "World";  
}  
echo "Hello {b()}\n";  
?>  
  
Also, it appears that you can almost without limitation perform other processing within the argument list, but not outside it.  For example:  
  
<?  
$true = true;  
define("HW", "Hello World");  
echo "{$true && HW}";  
?>  
  
gives: Parse error: parse error, unexpected T\_BOOLEAN\_AND, expecting '}' in - on line 3  
  
There may still be some way to kludge the syntax to allow constants and unbound function calls inside a double-quoted string, but it isn't readily apparent to me at the moment, and I'm not sure I'd prefer the workaround over breaking out of the string at this point.

I noticed that the documentation does not mention that when you have an XML element which contains a dash (-) in its name can only be accessed using the bracelets notation.  
For example:  
<xml version="1">  
<root>  
   <element-one>value4element-one</element-one>  
</root>  
  
to access the above 'element-one' using SimpleXML you need to use the following:  
  
$simpleXMLObj->root->{'element-one'}  
  
to retrieve the value.  
  
Hope this helps,  
Denis R.

Regarding the lack of complex expression interpolation, just assign an identity function to a variable and call it:  
  
function id($arg) { return $arg; }  
  
$expr = id;  
  
echo "Field is: {$expr( "1 ". ucfirst('whatzit')) }";  
  
It is slower due to an additional function call, but it does avoid the assignment of a one-shot temporary variable. When there are a lot of very simple value transformations made just for display purposes, it can de-clutter code.

Empty strings seem to be no real strings, because they behave different to strings containing data. Here is an example.  
  
It is possible to change a character at a specific position using the square bracket notation:  
<?php  
$str = '0';  
$str[0] = 'a';  
echo $str."\n"; // => 'a'  
?>  
  
It is also possible to change a character with does not exist, if the index is "behind" the end of the string:  
<?php  
$str = '0';  
$str[1] = 'a';  
echo $str."\n"; // => 0a  
?>  
  
But if you do that on an empty string, the string gets silently converted into an array:  
<?php  
$str = '';  
$str[0] = 'a';  
echo $str."\n"; // => Array  
?>

A note on the heredoc stuff.   
  
If you're editing with VI/VIM and possible other syntax highlighting editors, then using certain words is the way forward.  if you use <<<HTML for example, then the text will be hightlighted for HTML!!   
  
I just found this out and used sed to alter all EOF to HTML.   
  
JAVASCRIPT also works, and possibly others.  The only thing about <<<JAVASCRIPT is that you can't add the <script> tags..,  so use HTML instead, which will correctly highlight all JavaScript too..   
  
You can also use EOHTML, EOSQL, and EOJAVASCRIPT.

The docs say: "Heredoc text behaves just like a double-quoted string, without the double quotes" but there is a notable hidden exception to that rule: the final newline in the string (the one before closing heredoc token) is elided. i.e. if you have:  
  
$foo = <<<EOF  
a  
b  
c  
EOF;  
  
the result is equivalent to "a\nb\nc", NOT "a\nb\nc\n" like the docs imply.

$my\_int = "12,140";  
echo  1 + $my\_int ;  
  
Returns 13 not the expected 12141

Expectedly <?php $string[$x] ?> and <?php substr($string, $x, 1) ?> will yield the same result... normally!   
  
However, when you turn on the  Function Overloading Feature (<http://php.net/manual/en/mbstring.overload.php>), this might not be true!   
  
If you use this Overloading Feature with 3rd party software, you should check for usage of the String access operator, otherwise you might be in for some nasty surprises.

It may be obvious to some, but it's convenient to note that variables \_will\_ be expanded inside of single quotes if these occur inside of a double-quoted string.  This can be handy in constructing exec calls with complex data to be passed to other programs.  e.g.:  
  
$foo = "green";  
echo "the grass is $foo";  
the grass is green  
  
echo 'the grass is $foo';  
the grass is $foo  
  
echo "the grass is '$foo'";  
the grass is 'green'

error control operator (@) with heredoc syntax:  
  
the error control operator is pretty handy for supressing minimal errors or omissions. For example an email form that request some basic non mandatory information to your users. Some may complete the form, other may not. Lets say you don't want to tweak PHP for error levels and you just wish to create some basic template that will be emailed to the admin with the user information submitted. You manage to collect the user input in an array called $form:  
  
<?php  
// creating your mailer  
$mailer = new SomeMailerLib();  
$mailer->from = ' System <mail@yourwebsite.com>';  
$mailer->to = 'admin@yourwebsite.com';  
$mailer->subject = 'New user request';  
// you put the error control operator before the heredoc operator to suppress notices and warnings about unset indices like this  
$mailer->body = @<<<FORM  
Firstname = {$form['firstname']}  
Lastname = {$form['lastname']}  
Email = {$form['email']}  
Telephone = {$form['telephone']}  
Address = {$form['address']}  
FORM;  
  
?>

If you require a NowDoc but don't have support for them on your server -- since your PHP version is less than PHP 5.3.0 -- and you are in need of a workaround, I'd suggest using PHP's \_\_halt\_compiler() which is basically a knock-off of Perl's \_\_DATA\_\_ token if you are familiar with it.  
  
Give this a run to see my suggestion in action:  
  
<?php  
//set $nowDoc to a string containing a code snippet for the user to read  
$nowDoc = file\_get\_contents(\_\_FILE\_\_,null,null,\_\_COMPILER\_HALT\_OFFSET\_\_);  
$nowDoc=highlight\_string($nowDoc,true);  
  
echo <<<EOF  
<!doctype html>  
<html>  
<head>  
<meta http-equiv="content-type" content="text/html; charset=UTF-8" />  
<title>NowDoc support for PHP &lt; 5.3.0</title>  
<meta name="author" content="Ultimater at gmail dot com" />  
<meta name="about-this-page"  
content="Note that I built this code explicitly for the  
php.net documenation for demonstrative purposes." />  
<style type="text/css">  
body{text-align:center;}  
table.border{background:#e0eaee;margin:1px auto;padding:1px;}  
table.border td{padding:5px;border:1px solid #8880ff;text-align:left;  
background-color:#eee;}  
code ::selection{background:#5f5color:white;}  
code ::-moz-selection{background:#5f5;color:white;}  
a{color:#33a;text-decoration:none;}  
a:hover{color:rgb(3,128,252);}  
</style>  
</head>  
<body>  
<h1 style="margin:1px auto;">  
<a  
href="<http://php.net/manual/en/language.types.string.php#example-77>">  
Example #8 Simple syntax example  
</a></h1>  
<table class="border"><tr><td>  
$nowDoc  
</td></tr></table></body></html>  
EOF;  
  
\_\_halt\_compiler()  
//Example code snippet we want displayed on the webpage  
//note that the compiler isn't actually stopped until the semicolon  
;<?php  
$juices = array("apple", "orange", "koolaid1" => "purple");  
  
echo "He drank some $juices[0] juice.".PHP\_EOL;  
echo "He drank some $juices[1] juice.".PHP\_EOL;  
echo "He drank some juice made of $juice[0]s.".PHP\_EOL; // Won't work  
echo "He drank some $juices[koolaid1] juice.".PHP\_EOL;  
  
class people {  
    public $john = "John Smith";  
    public $jane = "Jane Smith";  
    public $robert = "Robert Paulsen";  
      
    public $smith = "Smith";  
}  
  
$people = new people();  
  
echo "$people->john drank some $juices[0] juice.".PHP\_EOL;  
echo "$people->john then said hello to $people->jane.".PHP\_EOL;  
echo "$people->john's wife greeted $people->robert.".PHP\_EOL;  
echo "$people->robert greeted the two $people->smiths."; // Won't work  
?>

An interesting finding about Heredoc "syntax error, unexpected $end".  
I got this error because I did not use the php close tag "?>" and I had no code after the heredoc code.  
  
foo1.php code gives "syntax error, unexpected $end".  
But in foo2.php and foo3.php, when you add a php close tag or when you have some more code after heredoc it works fine.  
  
Example Code:  
foo1.php  
1. <?php  
2. $str = <<<EOD  
3. Example of string  
4. spanning multiple lines  
5. using heredoc syntax.  
6. EOD;  
7.   
  
foo2.php  
1. <?php  
2. $str = <<<EOD  
3. Example of string  
4. spanning multiple lines  
5. using heredoc syntax.  
6. EOD;  
7.   
8. echo $str;  
9.  
  
foo3.php  
1. <?php  
2. $str = <<<EOD  
3. Example of string  
4. spanning multiple lines  
5. using heredoc syntax.  
6. EOD;  
7. ?>

You incorrectly stated that thee documentation doesn't refer anything about the semicolon at the end of the heredocs and nowdocs  being interpreted as a "real" semicolon.  
  
If you read carefully, you will notice this, in the 1st sentence of the warning about heredocs:  
  
"It is very important to note that the line with the closing identifier must contain no other characters, except a semicolon (;)."  
  
Interesting...  
It is refering about semicolons...  
  
But wait, there is more:  
  
<http://php.net/manual/en/language.basic-syntax.instruction-separation.php>  
1st sentence says:  
"As in C or Perl, PHP requires instructions to be terminated with a semicolon at the end of each statement."  
  
So, here says that semicolons are statement separators, basicly...  
  
So, if you put a "real" semicolon at the end of these examples:  
<?php  
    $a=5;  
    $foo="String";  
    $bar=array();  
    $yep=null;  
    $other=func();  
?>  
Why shouldn't you put at the end of heredocs and nowdocs?  
After all, a heredoc or a nowdoc is simply a string.  
  
You should read more carefully the documentation first before saying any comment.  
  
About serious questions:  
I didn't read all comments here, but you can run functions inside strings and heredocs.  
  
And you can even nest them inside {}  
  
Example:  
<?php  
    $f=function($x){$a=func\_get\_args();unset($a[0]);return call\_user\_func\_array($x,$a);};  
    $d=0;  
    echo $b=<<<NUMBERS  
4.0909 rounded is: {$f('round',4.0909,$d)}  
Time now is: {$f('time')}  
Nested heredocs/nowdocs: {$f('sprintf',<<<OTHER  
Here is an %s of nested %s  
OTHER  
,"Example",<<<'NOW'  
heredocs and nowdocs  
NOW  
)}  
NUMBERS;  
  
/\*echoes (time is system and real time relative):  
4.0909 rounded is: 4  
Time now is: 1386670912  
Nested heredocs/nowdocs: Here is an Example of nested heredocs and nowdocs  
\*/  
?>  
  
It's not pretty, and is hard to read, but sometimes it is useful to confuse curious people (like minifying the code).  
  
Warning: if any function that runs inside a string or heredoc gives a fatal error, the script MAY continue!

Note that :  
  
<?php  
echo 'error' == 0, '<br>'; // TRUE  
echo 'error' == '0', '<br>'; // FALSE  
echo '0' == 0, '<br>'; // TRUE  
  
// So, 'error' != 'error' ?  
?>

please note that when arrays are copied, the "reference status" of their members is preserved (<http://www.php.net/manual/en/language.references.whatdo.php>).

Note that array value buckets are reference-safe, even through serialization.   
  
<?php   
$x='initial';   
$test=array('A'=>&$x,'B'=>&$x);   
$test=unserialize(serialize($test));   
$test['A']='changed';   
echo $test['B']; // Outputs "changed"   
?>   
  
This can be useful in some cases, for example saving RAM within complex structures.

"If you convert a NULL value to an array, you get an empty array."   
  
This turns out to be a useful property. Say you have a search function that returns an array of values on success or NULL if nothing found.   
  
<?php $values = search(...); ?>   
  
Now you want to merge the array with another array. What do we do if $values is NULL? No problem:   
  
<?php $combined = array\_merge((array)$values, $other); ?>   
  
Voila.

Why not to user one-based arrays:  
  
<?php  
$a  = array(1 => 'a', 'b', 'd');  
print\_r($a);  
array\_splice($a,2,0,'c');  
print\_r($a);  
?>  
  
output:  
Array ( [1] => a [2] => b [3] => d ) Array ( [0] => a [1] => b [2] => c [3] => d )

Beware that if you're using strings as indices in the $\_POST array, that periods are transformed into underscores:  
  
<html>  
<body>  
<?php  
    printf("POST: "); print\_r($\_POST); printf("<br/>");  
?>  
<form method="post" action="<?php echo $\_SERVER['PHP\_SELF']; ?>">  
    <input type="hidden" name="Windows3.1" value="Sux">  
    <input type="submit" value="Click" />  
</form>  
</body>  
</html>  
  
Once you click on the button, the page displays the following:  
  
POST: Array ( [Windows3\_1] => Sux )

Note that objects of classes extending ArrayObject SPL class are treated as arrays, and not as objects when converting to array.  
  
<?php  
class ArrayObjectExtended extends ArrayObject  
{  
    private $private = 'private';  
    public $hello = 'world';  
}  
  
$object = new ArrayObjectExtended();  
$array = (array) $object;  
  
// This will not expose $private and $hello properties of $object,  
// but return an empty array instead.  
var\_export($array);  
?>

Used to creating arrays like this in Perl?  
  
@array = ("All", "A".."Z");  
  
Looks like we need the range() function in PHP:  
  
<?php  
$array = array\_merge(array('All'), range('A', 'Z'));  
?>  
  
You don't need to array\_merge if it's just one range:  
  
<?php  
$array = range('A', 'Z');  
?>

Regarding the previous comment, beware of the fact that reference to the last value of the array remains stored in $value after the foreach:  
  
<?php  
foreach ( $arr as $key => &$value )  
{  
    $value = 1;  
}  
  
// without next line you can get bad results...  
//unset( $value );  
  
$value = 159;  
?>  
  
Now the last element of $arr has the value of '159'. If we remove the comment in the unset() line, everything works as expected ($arr has all values of '1').  
  
Bad results can also appear in nested foreach loops (the same reason as above).  
  
So either unset $value after each foreach or better use the longer form:  
  
<?php  
foreach ( $arr as $key => $value )  
{  
    $arr[ $key ] = 1;  
}  
?>

[Editor's note: You can achieve what you're looking for by referencing $single, rather than copying it by value in your foreach statement. See <http://php.net/foreach> for more details.]   
  
Don't know if this is known or not, but it did eat some of my time and maybe it won't eat your time now...   
  
I tried to add something to a multidimensional array, but that didn't work at first, look at the code below to see what I mean:   
  
<?php   
  
$a1 = array( "a" => 0, "b" => 1 );   
$a2 = array( "aa" => 00, "bb" => 11 );   
  
$together = array( $a1, $a2 );   
  
foreach( $together as $single ) {   
    $single[ "c" ] = 3 ;   
}   
  
print\_r( $together );   
/\* nothing changed result is:   
Array   
(   
    [0] => Array   
        (   
            [a] => 0   
            [b] => 1   
        )   
    [1] => Array   
        (   
            [aa] => 0   
            [bb] => 11   
        )   
) \*/   
  
foreach( $together as $key => $value ) {   
    $together[$key]["c"] = 3 ;   
}   
  
print\_r( $together );   
  
/\* now it works, this prints   
Array   
(   
    [0] => Array   
        (   
            [a] => 0   
            [b] => 1   
            [c] => 3   
        )   
    [1] => Array   
        (   
            [aa] => 0   
            [bb] => 11   
            [c] => 3   
        )   
)   
\*/   
  
?>

<?php  
  
error\_reporting(E\_ALL);  
  
$res=42;  
  
echo("+".$res[0]."+".$res[10]."+");  
  
$res[0]=1;  
  
?>  
  
The last line correctly states "Warning: Cannot use a scalar value as an array", but the echo produces +++ as output with no warning. This seems strange to me.

There is another kind of array (php>=  5.3.0) produced by   
  
$array = new SplFixedArray(5);  
  
Standard arrays, as documented here, are marvellously flexible and, due to the underlying hashtable, extremely fast for certain kinds of lookup operation.   
  
Supposing a large string-keyed array   
  
$arr=['string1'=>$data1, 'string2'=>$data2 etc....]  
  
when getting the keyed data with   
  
$data=$arr['string1'];  
  
php does \*not\* have to search through the array comparing each key string to the given key ('string1') one by one, which could take a long time with a large array. Instead the hashtable means that php takes the given key string and computes from it the memory location of the keyed data, and then instantly retrieves the data. Marvellous! And so quick. And no need to know anything about hashtables as it's all hidden away.  
  
However, there is a lot of overhead in that. It uses lots of memory, as hashtables tend to (also nearly doubling on a 64bit server), and should be significantly slower for integer keyed arrays than old-fashioned (non-hashtable) integer-keyed arrays. For that see more on SplFixedArray :  
  
<http://uk3.php.net/SplFixedArray>  
  
Unlike a standard php (hashtabled) array, if you lookup by integer then the integer itself denotes the memory location of the data, no hashtable computation on the integer key needed. This is much quicker. It's also quicker to build the array compared to the complex operations needed for hashtables. And it uses a lot less memory as there is no hashtable data structure. This is really an optimisation decision, but in some cases of large integer keyed arrays it may significantly reduce server memory and increase performance (including the avoiding of expensive memory deallocation of hashtable arrays at the exiting of the script).

Please note that adding the magic \_\_toString() method to your objects will not allow you to seek an array with it, it still throws an Illegal Offset warning.  
  
The solution is to cast it to a string first, like this  
  
$array[(string) $stringableObject]

PHP array\_diff\_assoc() Function  
  
You can compare the keys and values of two arrays, and return the differences:  
  
<?php  
$a1=array("red","green","blue","yellow");  
$a2=array("red","green","blue");  
  
$result=array\_diff\_assoc($a1,$a2);  
print\_r($result);  
?>   
  
[http://www.show-ip.org](http://www.show-ip.org/)

One thing to be careful of is making any assumptions about the underlying implementation with respect to performance. For example, the documentation talks about hash-maps, which might lead you to expect O(1) key lookups.  
  
<?php  
  
function find\_val($n) {  
  $t = array();  
  $last = null;  
    
  for ($x = 0; $x < $n; $x++) {  
    $last = "" . $x;  
    $t[] = $last;  
  }  
    
  var\_dump(in\_array($last, $t));  
}    
  
    
function find\_key($n) {  
  $t = array();  
  $last = null;  
    
  for ($x = 0; $x < $n; $x++) {  
    $last = "" . $x;  
    $t[$last] = true;  
  }  
    
  var\_dump(array\_key\_exists($last, $t));  
}    
  
$n = 1600000;  
  
find\_val($n);  
// Time taken: 1123ms  
  
find\_key($n);  
// Time taken: 803  
  
/\*   
  
Additional Timings:   
  
n        find\_val(ms)  find\_key(ms)  
100000   99             82  
200000   169            130  
400000   301            217  
800000   570            416  
1600000  1123           803  
  
\*/  
?>  
  
In my tests, both in\_array and array\_key\_exists exhibited the same order of growth.

This page should include details about how associative arrays are implemened inside PHP; e.g. using hash-maps or b-trees.  
  
This has important implictions on the permance characteristics of associative arrays and how they should be used; e.g. b-tree are slow to insert but handle collisions better than hashmaps.  Hashmaps are faster if there are no collisions, but are slower to retrieve when there are collisions.  These factors have implictions on how associative arrays should be used.

On array recursion...   
  
Given the following code:   
  
<?php   
$myarray = array('test',123);   
$myarray[] = &$myarray;   
print\_r($myarray);   
?>   
  
The print\_r() will display \*RECURSION\* when it gets to the third element of the array.   
  
There doesn't appear to be any other way to scan an array for recursive references, so if you need to check for them, you'll have to use print\_r() with its second parameter to capture the output and look for the word \*RECURSION\*.   
  
It's not an elegant solution, but it's the only one I've found, so I hope it helps someone.

It is true that "array assignment always involves value copying", but the copy is a "lazy copy". This means that the data of the two variables occupy the same memory as long as no array element changes.  
  
E.g., if you have to pass an array to a function that only needs to read it, there is no advantage at all in passing it by reference.

Be very careful when using a result as an array. <?php echo $a['foo']['bar']['baz'] ?> will throw an error if $a is an object, and throw a warning if $a is an array but does not have the right keys, but it will silently return true if $a is null or boolean or int, and if $a is a string, it will return its first character. (This is true even with E\_STRICT set.) This can be a major gotcha with functions which return null or false if they are unsuccessful.

I think there's a mistake in the las example:  
...'<?php  
$arr1 = array(2, 3);  
$arr2 = $arr1;  
$arr2[] = 4; // $arr2 ha cambiado,  
             // $arr1 sigue siendo array(2, 3)  
              
$arr3 = &$arr1;  
$arr3[] = 4; // ahora $arr1 y $arr3 son iguales  
?>'  
  
I think it should be: ...'ahora $arr2 y $arr3 son iguales'...  
  
Thanks.

Its worth noting that there does not appear to be any functional limitations on the length or content of string indexes. The string indexes for your arrays can contain any characters, including new line characters, and can be of any length:  
  
<?php  
  
$key = "XXXXX";  
$test = array($key => "test5");  
  
for ($x = 0; $x < 500; $x++) {  
  $key .= "X";  
  $value = "test" . strlen($key);  
  $test[$key] = $value;  
}  
  
echo "<pre>";  
print\_r($test);  
echo "</pre>";  
  
?>  
  
Keep in mind that using extremely long array indexes is not a good practice and could cost you lots of extra CPU time. However, if you have to use a long string as an array index you won't have to worry about the length or content.

In response to sirbinam.  
You cannot call a function or method before it exists. In your example, the global instance of stdout is just being passed around to differnet references (pointers). It however exists in the "dump" function scope via the global keyword.   
  
The code below works fine and illustrates that "stdout" has been defined before its instantiation.   
  
<?php  
  
class profiler{  
  function profiler(){  
    $this->starttime = microtime();  
  }  
  
  function dump(){  
    global $stdout;  
    $this->endtime = microtime();  
    $duration = $this->endtime - $this->starttime;  
    $stdout->write($duration);  
  }  
}  
  
class stdout{  
  function write($msg){  
    echo $msg;  
  }  
}  
  
$stdout =& new stdout();  
$profiler =& new profiler();  
$profiler->dump();  
  
?>  
  
All classes and functions declarations within a scope exist even before the php execution reaches them. It does not matter if you have your classes defined on the first or last line, as long as they are in the same scope as where they are called and are not in a conditional statement that has not been evaluated yet.

In response to harmor: if an array contains another array as a value, you can recursively convert all arrays with:  
  
<?php  
function arrayToObject( $array ){  
  foreach( $array as $key => $value ){  
    if( is\_array( $value ) ) $array[ $key ] = arrayToObject( $value );  
  }  
  return (object) $array;  
}  
?>

By far the easiest and correct way to instantiate an empty generic php object that you can then modify for whatever purpose you choose:   
  
<?php $genericObject = new stdClass(); ?>   
  
I had the most difficult time finding this, hopefully it will help someone else!

Do you remember some JavaScript implementations?  
  
// var timestamp = (new Date).getTime();  
  
Now it's possible with PHP 5.4.\*;  
  
<?php  
class Foo  
{  
    public $a = "I'm a!";  
    public $b = "I'm b!";  
    public $c;  
      
    public function getB() {  
        return $this->b;  
    }  
      
    public function setC($c) {  
        $this->c = $c;  
        return $this;  
    }  
      
    public function getC() {  
        return $this->c;  
    }  
}  
  
print (new Foo)->a;      // I'm a!  
print (new Foo)->getB(); // I'm b!  
?>  
  
or  
  
<?php  
// $\_GET["c"] = "I'm c!";  
print (new Foo)  
       ->setC($\_GET["c"])  
       ->getC(); // I'm c!  
?>

You can create [recursive] objects with something like:  
<?php  
  $literalObjectDeclared = (object) array(  
     'foo' => (object) array(  
          'bar' => 'baz',  
          'pax' => 'vax'  
      ),  
      'moo' => 'ui'  
   );  
print $literalObjectDeclared->foo->bar; // outputs "baz"!  
?>

CAUTION:  
"Arrays convert to an object with properties named by keys, and corresponding values".  
  
This is ALWAYS true, which means that even numeric keys are accepted when converting.  
But the resulting properties cannot be accessed, since they don't match the variables naming rules.  
  
So this:  
<?php  
$x = (object) array('a'=>'A', 'b'=>'B', 'C');  
echo '<pre>'.print\_r($x, true).'</pre>';  
?>  
works and displays:  
stdClass Object  
(  
    [a] => A  
    [b] => B  
    [0] => C  
)  
  
But this:  
<?php  
echo '<br />'.$x->a;  
echo '<br />'.$x->b;  
echo '<br />'.$x->{0}; # (don't use $x->0, which is obviously a syntax error)  
?>  
fails and displays:  
A  
B  
Notice: Undefined property: stdClass::$0 in...

Why is there nothing in the docs about instantiating a generic object?  ie new object() - it does not work.  
  
If one wants to create a dynamic object on the fly, the only option I see is to create it implicitly, by creating a child of it - ghetto:  
  
unset($obj);  
$obj->blah = 3;

i would like to share a curious behavior on casted objects. Casting an object from a class with private/protected attributes results a stdClass with a private/protected attribute for get.  
Example:  
<?PHP  
class Foo{  
private $priv = 1;  
public $pub = 2;  
public function getSimple(){  
  return (object)(array) $this; //the array cast is to force a stdClass result  
}  
}  
$bar = new Foo();  
var\_dump($bar->getSimple();// output: object(stdClass)#3 (2) { ["priv:private"]=> int(1) ["pub"]=> int(2) }  
  
var\_dump($bar->getSimple()->priv);// output: NULL, not a Fatal Error  
var\_dump($bar->getSimple()->pub);// output: int(2)  
  
$barSimple = $bar->getSimple();  
$barSimple->priv = 10;  
var\_dump($barSimple->priv);// output: int(10)  
  
?>

Here a new updated version of 'stdObject' class. It's very useful when extends to controller on MVC design pattern, user can create it's own class.  
  
Hope it help you.  
  
<?php  
class stdObject {  
    public function \_\_construct(array $arguments = array()) {  
        if (!empty($arguments)) {  
            foreach ($arguments as $property => $argument) {  
                $this->{$property} = $argument;  
            }  
        }  
    }  
  
    public function \_\_call($method, $arguments) {  
        $arguments = array\_merge(array("stdObject" => $this), $arguments); // Note: method argument 0 will always referred to the main class ($this).  
        if (isset($this->{$method}) && is\_callable($this->{$method})) {  
            return call\_user\_func\_array($this->{$method}, $arguments);  
        } else {  
            throw new Exception("Fatal error: Call to undefined method stdObject::{$method}()");  
        }  
    }  
}  
  
// Usage.  
  
$obj = new stdObject();  
$obj->name = "Nick";  
$obj->surname = "Doe";  
$obj->age = 20;  
$obj->adresse = null;  
  
$obj->getInfo = function($stdObject) { // $stdObject referred to this object (stdObject).  
    echo $stdObject->name . " " . $stdObject->surname . " have " . $stdObject->age . " yrs old. And live in " . $stdObject->adresse;  
};  
  
$func = "setAge";  
$obj->{$func} = function($stdObject, $age) { // $age is the first parameter passed when calling this method.  
    $stdObject->age = $age;  
};  
  
$obj->setAge(24); // Parameter value 24 is passing to the $age argument in method 'setAge()'.  
  
// Create dynamic method. Here i'm generating getter and setter dynimically  
// Beware: Method name are case sensitive.  
foreach ($obj as $func\_name => $value) {  
    if (!$value instanceOf Closure) {  
  
        $obj->{"set" . ucfirst($func\_name)} = function($stdObject, $value) use ($func\_name) {  // Note: you can also use keyword 'use' to bind parent variables.  
            $stdObject->{$func\_name} = $value;  
        };  
  
        $obj->{"get" . ucfirst($func\_name)} = function($stdObject) use ($func\_name) {  // Note: you can also use keyword 'use' to bind parent variables.  
            return $stdObject->{$func\_name};  
        };  
  
    }  
}  
  
$obj->setName("John");  
$obj->setAdresse("Boston");  
  
$obj->getInfo();  
?>

If you call var\_export() on an instance of stdClass, it attempts to export it using ::\_\_set\_state(), which, for some reason, is not implemented in stdClass.  
  
However, casting an associative array to an object usually produces the same effect (at least, it does in my case). So I wrote an improved\_var\_export() function to convert instances of stdClass to (object) array () calls. If you choose to export objects of any other class, I'd advise you to implement ::\_\_set\_state().  
  
<?php  
/\*\*  
\* An implementation of var\_export() that is compatible with instances  
\* of stdClass.  
\* @param mixed $variable The variable you want to export  
\* @param bool $return If used and set to true, improved\_var\_export()  
\*     will return the variable representation instead of outputting it.  
\* @return mixed|null Returns the variable representation when the  
\*     return parameter is used and evaluates to TRUE. Otherwise, this  
\*     function will return NULL.  
\*/  
function improved\_var\_export ($variable, $return = false) {  
    if ($variable instanceof stdClass) {  
        $result = '(object) '.improved\_var\_export(get\_object\_vars($variable), true);  
    } else if (is\_array($variable)) {  
        $array = array ();  
        foreach ($variable as $key => $value) {  
            $array[] = var\_export($key, true).' => '.improved\_var\_export($value, true);  
        }  
        $result = 'array ('.implode(', ', $array).')';  
    } else {  
        $result = var\_export($variable, true);  
    }  
  
    if (!$return) {  
        print $result;  
        return null;  
    } else {  
        return $result;  
    }  
}  
  
// Example usage:  
$obj = new stdClass;  
$obj->test = 'abc';  
$obj->other = 6.2;  
$obj->arr = array (1, 2, 3);  
  
improved\_var\_export((object) array (  
    'prop1' => true,  
    'prop2' => $obj,  
    'assocArray' => array (  
        'apple' => 'good',  
        'orange' => 'great'  
    )  
));  
  
/\* Output:  
(object) array ('prop1' => true, 'prop2' => (object) array ('test' => 'abc', 'other' => 6.2, 'arr' => array (0 => 1, 1 => 2, 2 => 3)), 'assocArray' => array ('apple' => 'good', 'orange' => 'great'))  
\*/  
?>  
  
Note: This function spits out a single line of code, which is useful to save in a cache file to include/eval. It isn't formatted for readability. If you want to print a readable version for debugging purposes, then I would suggest print\_r() or var\_dump().

PHP supports recursive type definitions as far as I've tried. The class below (a \_very\_ simple tree) is an example:  
  
class Tree {  
  
var $\_value = null;  
var $\_children = array();  
  
function Tree ($value) {  
  $this->\_value = $value;  
}  
  
function addChild ($value) {  
  $aux\_node = new Tree ($value);  
  $this->\_children [] = $aux\_node;  
  return $aux\_node;   
}  
}  
  
As you can see, in addChild we reference Tree again...  
  
However, you must be careful about references. See the chapter "References explained" for more details.  
  
Hope this helps.

You can create a new object using the built-in stdClass or by using type-casting:  
  
<?php  
  
// This is the proper way  
$object1 = new stdClass();  
  
// This works too  
$object2 = (object) NULL;  
  
// This will create an object from an array  
$monkey\_array = array('title'=>'Spider Monkey', 'src'=>'monkey.jpg');  
$monkey\_object = (object) $monkey\_array;  
print $monkey\_object->title . ' ' . $monkey\_object->src;  
  
// You can type-cast in the middle of an expression  
function customHTML($some\_object) {  
// this function expects an object as the argument and returns some output  
}  
print '<p>Writing some output ' . customHTML( (object) array('rows'=>3, 'cols'=>4) );  
  
?>

If you use new to create items in an array, you may not get the results you want since the parameters to array will be copies of the original and not references.  
  
By Example:  
class Store {  
    var $item = 3;  
}  
  
    $a = array( new Store() );  
    $b = $a;  
    $a[0]->item = 2;  
    print( "|" . $b[0]->item . "| <br>" );   //shows 3  
  
    $a = array();  
    $a[] =& new Store();  
    $b = $a;  
    $a[0]->item = 2;  
    print( "|" . $b[0]->item . "| <br>" );   //shows 2  
  
This is extremely important if you intend on passing arrays of classes to functions and expect them to always use the same object instance!  
  
Note: The following syntax is desired (or maybe even the default notation should translate as this):  
   $a = array( &new Store() );

In reply to the usort thing, you can access a property of an object dynamically by:  
<?php  
$obj = (object)array("Test" => "bar")  
$var = "Test";  
echo $obj->$var;  
?>  
This will output "bar", and do notice I call on ->$var and not just ->var.

To sort an array, that contains an object, after one fieldname inside the object, im using this function:  
  
function objectSort($objectarray, $field)  
{  
    for ($a=0;$a < (count($objectarray)); $a++)  
    {  
        for ($b=0;$b < (count($objectarray)); $b++)  
        {      
            if ($objectarray[$a]->$field < $objectarray[$b]->$field)  
            {  
                $temp = $objectarray[$a];  
                $objectarray[$a] = $objectarray[$b];  
                $objectarray[$b] = $temp;  
            }  
        }  
    }  
      
    return $objectarray;  
}

If having  
  
class BugDetails extends Bug  
  
you want to cast an object of a Bug to BugDetails like this  
<?php  
    $clone = (BugDetails) clone $this;  
// OR  
    $clone = (BugDetails) $bug;  
?>  
which doesn't work in PHP, you have two options:  
1. Copying all (including private) properties manually (you could also use get\_object\_vars(), but this is shorter):  
<?php  
    $clone = new BugDetails();  
    foreach ($this as $key => $val) {  
        $clone->$key = $val;  
    }  
?>  
2. Serialize an object Bug, manipulate the resulting string so that it has BugDetails inside and unserialize it.  
See here: <http://blog.adaniels.nl/articles/a-dark-corner-of-php-class-casting/>  
  
Spent two hours looking for more elegant solution, that's my findings.

php at electricsurfer.com,  
  
More than a year later and here's some clarification of what's happening in your code, via comments in an otherwise verbatim copy.  
  
<?  
class c  
{  
   var $a = array('a'=>'aa','b'=>'ab');  
   var $b = 'c';  
    
   function show()  
   {  
       echo $this->a['a']; // -> 1st  
       echo $this->a['b']; // outputs 'ab'  
        
       $a = 'a';  
       $b = 'b';  
        
       echo $this->$a[$a]; // [] 1st, not what I expected  
       //Above first becomes $this->$a['a'] by looking at the function's local $a  
       //Next it becomes $this->a by again looking at the function's local $a, which references the class variable $a with no subscripts.  
       // In order to reference elements of the class variable $a,   
       // you want to use $this->a[$a]  
  
       echo $this->$a[$b]; // does NOT output 'ab'  
       // Same as above, but the first step $b becomes 'b'  
  
       $this\_a =& $this->$a; // work-around  
        
       echo $this\_a[$a]; // no question  
       echo $this\_a[$b];  
        
       $a\_arr = array('a'=>'b');  
        
       echo $this->$a\_arr[$a]; // [] 1st => outputs 'c'  
       // This becomes $this->$a\_arr['a'] which becomes $this->c,  
       // by referencing the local variables first.  
   }  
}  
$c = new c();  
$c->show();  
?>

Initialization, Instantiation and Instances are terms that can be confusing at first. Let's try to sort it out starting with this simple class definition.  
  
<?php  
  
class User  
{  
  public $first\_name;  
  public $last\_name;  
  
  public function \_\_toString()  
  {  
    return "User [first='$this->first\_name', last='$this->last\_name']";  
  }  
}  
  
?>  
  
Now create several INSTANCES of User by INSTANTIATING the User class above.  
  
<?php  
$user\_1 = new User;        // $user\_1 is an INSTANCE of User  
$user\_2 = new User;        // $user\_2 is an INSTANCE of User  
echo $user\_1 . '<br>';     // User [first='', last='']  
echo $user\_2 . '<br>';     // User [first='', last='']  
?>  
  
Here we have (2) two INSTANCES of User, but each instance was only INSTANTIATED once - when we used the 'new' operator.  
  
And now looking at the printed output, you can see there are no values for their first or last names. This means that the objects themselves have NOT been INITIALIZED. To remedy this situation, rewrite the class definition by adding a \_\_construct() method.  
  
<?php  
  
class User  
{  
  public $first\_name;  
  public $last\_name;  
  
  public function \_\_construct($first, $last)    // Require first and last names when INSTANTIATING  
  {  
    $this->first\_name = $first;                 // INITIALIZE $first\_name;  
    $this->last\_name = $last;                   // INITIALIZE $last\_name;  
  }  
  
  public function \_\_toString()  
  {  
    return "User [first='$this->first\_name', last='$this->last\_name']";  
  }  
}  
  
?>  
  
Now try it again.  
  
<?php  
$user\_1 = new User('John', 'Doe');      // $user\_i is an INSTANCE of User    
$user\_2 = new User('Jane', 'Doe');      // $user\_2 is an INSTANCE of User  
echo $user\_1 . '<br>';                  // prints: User [first='John', last='Doe']  
echo $user\_2 . '<br>';                  // prints: User [first='Jane', last='Doe']  
?>  
  
The \_\_construct() method is called automatically by PHP when it sees the 'new' operator. Our \_\_construct() method above requires the first and last names to be passed in as arguments and uses them to INITIALIZE objects when INSTANTIATING new INSTANCES.

In response to Harmor and Mithras,  you can use the json functions to convert multi-dimensional arrays to objects very reliably.  
  
Also, note that just using (object)$x doesn't allow you to access properties inline.  For example, this is invalid:  
  
<?php  
$x = array("foo"=>"bar");  
echo ((object)$x)->foo; // PHP Parse error, unexpected T\_OBJECT\_OPERATOR  
?>  
  
However, this function will let you do that, and will also handle multi-dimensional arrays without any hassle.  
  
<?php  
function to\_object ($x) {  
    return (is\_object($x) || is\_array($x)) ? json\_decode(json\_encode($x)) : (object) $x;  
}  
  
echo to\_object( array("foo"=>"bar") )->foo; // "bar"  
?>  
  
Note that \*numeric\* arrays will not be converted to objects using this method.

Bob, I think you solution is to use stdClass and add properties on the fly...  
  
$obj = new stdClass();  
$obj->data ="This is the solution";

In response to Bobs note below (and Rick's somewhat): It is much better to predefine the object you want to create. This is because in all the programs you write, you will still create objects On-the-Fly anyway:  
  
$foo = 'My String Object';      // on-the-fly!  
$bar = 45;                      // on-the-fly!  
$obj = new stdClass();          // on-the-fly!  
$obj->blah = 3;                 // on-the-fly!  
  
So what's missing above? Real control over WHAT your program does, HOW it does it, and HOW your data is organized. Let's say your code $obj->blah = 3 was 'blah' because 3 was your last Test Score (take no offense, I've done worse!). In that case, you could use code like so:  
  
<?php  
  
class Test{  
  const MIN = 0, MAX = 100;     // acceptable range for a test  
  
  protected $person;            // person taking the test  
  protected $score;             // use the word 'score' because 'blah' is meaningless  
  
  public function \_\_construct($p, $s){  
    $this->person = $p;  
    $this->score = self::clamp($s);  
  }  
  
  protected static function clamp($val){  
    if($val < self::MIN) $val = self::MIN;  
    if($val > self::MAX) $val = self::MAX;  
    return $val;  
  }  
  
  public function \_\_toString(){  
    return "Test [person=$this->person, score=$this->score%]";  
  }  
}  
  
$obj = new Test("John Doe", 83);    // on-the-fly!  
echo (object)$obj;                  // outputs 'Test [person=John Doe, score=83%]'  
?>  
  
Writing your own class definitions help maintain the "integerity" of your objects - see how the clamp() function keeps test scores between 0 and 100 percent? PHP  uses similar techniques to set min and max vals on Numbers. And, classes like Test are way more useful than objects created with an empty shell, uh em, stdClass. In addition, you can see WHAT, WHERE, WHY, and HOW your program actually works... On-the-Fly!  
  
-->

Class like stdClass but with the possibility to add and execute function.  
  
class stdObject {  
    public function \_\_construct(array $arguments = array()) {  
        if (!empty($arguments)) {  
            foreach ($arguments as $property => $argument) {  
                if ($argument instanceOf Closure) {  
                    $this->{$property} = $argument;  
                } else {  
                    $this->{$property} = $argument;  
                }  
            }  
        }  
    }  
  
    public function \_\_call($method, $arguments) {  
        if (isset($this->{$method}) && is\_callable($this->{$method})) {  
            return call\_user\_func\_array($this->{$method}, $arguments);  
        } else {  
            throw new Exception("Fatal error: Call to undefined method stdObject::{$method}()");  
        }  
    }  
}

In PHP 5.3.6, even if the following code doesn't have any apparent syntax errors:  
  
<?php  
class Test {  
  public $a = SplFixedArray::fromArray(array());  
}  
?>  
  
, the result is:  
  
Parse error: syntax error, unexpected '(', expecting ',' or ';' in ~\test.php on line 3  
  
This is because on an implicit object initialization of $a, which is not allowed in PHP.  
Hope this helps someone.

Note: empty array is converted to null by non-strict equal '==' comparison. Use is\_null() or '===' if there is possible of getting empty array.  
  
$a = array();  
  
$a == null  <== return true  
$a === null < == return false  
is\_null($a) <== return false

Funny. It looks like, that there is one, and only one possible value for variable $a that will pass this test:  
  
($a != NULL) && ((bool)$a == NULL)  
  
It's "0" and it works because casting string "0" to boolean gives FALSE (and it's the only non empty string, that works this way). So remember that casting is not "transitive".

This simple shorthand seems to work for setting new variables to NULL:   
  
<?php   
$Var;   
?>   
  
The above code will set $Var to NULL   
  
UPDATE: After further testing it appears the code only works in the global scope and does not work inside functions.   
  
<?php   
function Example(){   
  $Var;   
  var\_dump($Var);   
}   
?>   
  
Would not work as expected.

Watch out. You can define a new constant with the name NULL with define("NULL","FOO");. But you must use the function constant("NULL"); to get it's value. NULL without the function call to the constant() function will still retrieve the special type NULL value.  
Within a class there is no problem, as const NULL="Foo"; will be accessible as myClass::NULL.

   Null is the Constant in PHP. it is use to assign a empty value to the variable like  
  
  $a=NULL;  
  
  At this time $a has is NULL or $a has no value;  
  
  When we declaire a veriable in other languages than that veriable has some value depending on the value of memory location at which it is pointed but in php when we declaire a veriable than php assign a NULL to a veriable.

if you declare something like this :   
  
<?php   
class toto   
{   
    public $a = array();   
  
    public function load()   
    {   
        if ($this->a == null) // ==> the result is true   
            $a = other\_func();   
    }   
  
}   
?>   
  
be carefull, that's strange but an empty array is considered as a null variable

Nulls are almost the same as unset variables and it is hard to tell the difference without creating errors from the interpreter:   
  
<?php   
$var = NULL;   
?>   
  
isset($var) is FALSE   
empty($var) is TRUE   
is\_null($var) is TRUE   
  
isset($novar) is FALSE   
empty($novar) is TRUE   
is\_null($novar) gives an Undefined variable error   
  
$var IS in the symbol table (from get\_defined\_vars())   
$var CAN be used as an argument or an expression.   
  
So, in most cases I found that we needed to use !isset($var) intead of is\_null($var) and then set $var = NULL if the variable needs to be used later to guarantee that $var is a valid variable with a NULL value instead of being undefined.

Be careful using NULL together with namespaces. If a NULL constant is redefined in a namespace other than global, you will get unexpected results when comparing to NULL inside the namespace. Instead always use \NULL, \FALSE, and \TRUE when comparing. Otherwise it may lead to application failures and potential security issues where certain checks could be effectively disabled.   
  
A simple example to demonstrate the behavior:   
  
<?php   
namespace RedefinedConstants {   
  
    // redefining global namespace constants has no effect   
    define('NULL', 'I am not global NULL!');   
    define('TRUE', 'I am not global TRUE!');   
    define('FALSE', 'I am not global FALSE!');   
  
    // redefining local namespace constants will work   
    define('RedefinedConstants\NULL', 'I am not NULL!', \TRUE);   
    define('RedefinedConstants\FALSE', 'I am not FALSE!', \TRUE);   
    define('RedefinedConstants\TRUE', 'I am not TRUE!', \TRUE);   
  
    var\_dump(   
        NULL, \NULL, null, \null, Null, \Null,   
        FALSE, \FALSE, false, \false, False, \False,   
        TRUE, \TRUE, true, \true, True, \True   
    );   
  
}   
?>

Note the following:  
$test ='';  
  
if (isset($test)){  
    echo 'Variable test exists';  
}  
if (empty($test)){ // same result as $test = ''  
    echo ' and is\_empty';  
}  
if ($test == null){ // not the same result as is\_null($test)  
    echo 'and is\_null';  
}  
The result would be:  
Variable test exists and is\_empty and is\_null  
  
But for the following code...:  
$test ='';  
  
if (isset($test)){  
    echo 'Variable test exists';  
}  
if (empty($test)){ // same result as $test = ''  
    echo ' and is\_empty';  
}  
if ($test === null){  // same result as is\_null($test)  
    echo 'and is\_null';  
}  
The result would be:  
Variable test exists and is\_empty  
  
Therefore, for empty string variables, seems that 'empty' and 'null' has the same value but different type.

a quick note about the magic function \_\_get() :   
  
<?php   
class Foo{   
      
    protected $bar;   
      
    public function \_\_construct(){   
          
        $this->bar = NULL;   
        var\_dump( $this->bar ); //prit 'NULL' but won't call the magic method \_\_get()   
          
        unset( $this->bar );   
        var\_dump( $this->bar ); //print 'GET bar' and 'NULL'   
              
    }   
      
    public function \_\_get( $var ){ echo "GET " . $var; }   
          
}   
  
new Foo();   
?>

empty() is\_null() !isset()  
  
$var = "";  
  
empty($var) is true.  
is\_null($var) is false.  
!isset($var) is false.

// Difference between "unset($a);" and "$a = NULL;" :  
<?php  
// unset($a)  
$a = 5;  
$b = & $a;  
unset($a);  
print "b $b "; // b 5   
  
// $a = NULL; (better I think)  
$a = 5;  
$b = & $a;  
$a = NULL;  
print "b $b "; // b   
print(! isset($b)); // 1   
?>

Performance note: The callable type hint, like is\_callable(), will trigger an autoload of the class if the value looks like a static method callback.

You can also use the $this variable to specify a callback:   
  
<?php   
class MyClass {   
  
    public $property = 'Hello World!';   
  
    public function MyMethod()   
    {   
        call\_user\_func(array($this, 'myCallbackMethod'));   
    }   
  
    public function MyCallbackMethod()   
    {   
        echo $this->property;   
    }   
  
}   
?>

When specifying a call back in array notation (ie. array($this, "myfunc") ) the method can be private if called from inside the class, but if you call it from outside you'll get a warning:   
  
<?php   
  
class mc {   
   public function go(array $arr) {   
       array\_walk($arr, array($this, "walkIt"));   
   }   
  
   private function walkIt($val) {   
       echo $val . "<br />";   
   }   
  
    public function export() {   
        return array($this, 'walkIt');   
    }   
}   
  
$data = array(1,2,3,4);   
  
$m = new mc;   
$m->go($data); // valid   
  
array\_walk($data, $m->export()); // will generate warning   
  
?>   
  
Output:   
1<br />2<br />3<br />4<br />   
Warning: array\_walk() expects parameter 2 to be a valid callback, cannot access private method mc::walkIt() in /in/tfh7f on line 22

you can pass an object as a callable if its class defines the \_\_invoke() magic method..

> As of PHP 5.2.3, it is also possible to pass 'ClassName::methodName'  
  
You can also use 'self::methodName'.  This works in PHP 5.2.12 for me.

i did the same thing mentioned above but the following code not working. Why?  
---------------------------------------------------------  
class Alfa1 {  
    public static function falfa() {  
        echo "In Alfa";  
    }  
}  
  
class Beta1 extends Alfa1 {  
    public static function falfa() {  
        echo "In Beta";  
    }  
}  
  
call\_user\_func(array('Beta1','parent::flafa'));  
  
Tested in two latest version of PHP both gives:  
  
Warning: call\_user\_func() expects parameter 1 to be a valid callback, class 'Alfa1' does not have a method 'flafa' in Xampp\htdocs\a.php on line 15

I noticed two important thing about putting callbacks into an arg list when calling a function:  
  
1. The function to which the callback refers must be defined earlier in the source stream. So for example:  
  
function main() {...; usort($array, 'sortfunction'); ... }  
function sortfunction($a, $b){ return 0; }  
  
Will NOT work, but this will:  
  
function sortfunction($a, $b){ return 0; }  
function main() {...; usort($array, 'sortfunction'); ... }  
  
2. It's not really just a string. For example, this doesn't work:  
  
usort($array, ($reverse?'reversesorter':'forwardsorter'));  
  
I found these two discoveries quite counterintuitive.

The documentation is a little confusing, and with the recent OO changes it adds a little more to the confusion.  
  
I was curious whether you could pass an object through the user func, modify it in that callback and have the actual object updated or whether some cloning was going on behind the scenes.  
  
<?php  
    class Test  
    {  
        var $sValue = 'abc';  
  
        function testing($objTest)  
        {  
            $objTest->sValue = '123';  
        }  
    }  
  
    $obj = new Test();  
  
    call\_user\_func(array($obj, 'testing'), $obj);  
  
    var\_dump($obj);  
  
?>  
  
This works as expected: The object is not cloned, and $sValue is properly set to '123'. With the OO changes in PHP 5, you don't need to do "function testing(&$objTest)" as it is already passed by reference.

An example with PHP 5.3 and lambda functions  
  
<?php  
  
  array\_map (function ($value) {  
    return new MyFormElement ($value);  
  }, $\_POST);  
  
?>

To recap mr dot lilov at gmail dot com's comment: If you want to pass a function as an argument to another function, for example "array\_map", do this:  
  
regular functions:   
<?   
array\_map(intval, $array)  
?>  
  
static functions in a class:  
<?  
array\_map(array('MyClass', 'MyFunction'), $array)  
?>  
  
functions from an object:  
<?  
array\_map(array($this, 'MyFunction'), $array)  
?>  
  
I hope this clarifies things a little bit

I find these explanations and descriptions utterly useless.  
  
After reading all of the PHP Documentation multiple times, only one question remains:   
  
Were the college professors high on acid when they decided that it would be a good idea to assign the writing of PHP's Documentation as an extra credit assignment to a group of English as a Second Langauge (ESL) undergraduates who needed to get an "A" in their COMPUTER SCIENCE 101  class in order to avoid a beating from their parents over christmas break.

If you pass a string as the callback function (i.e., 2nd parm to preg\_replace\_callback()), then PHP will interpret it as a function's name in the current scope -- and Main::dada\_cb is not a valid function name in any scope.  
  
If you want to specify a static method of a class as the callback (i.e., "Main::dada\_cb"), then you must pass as 2nd parm to preg\_replace\_callback:  
  
array( 'Main', 'dada\_cb')  
  
And, if you want to use as a callback some method of an instantiated object (i.e., $object->dada\_cb), then you must pass as the 2nd parm to preg\_replace\_callback:  
  
array( $object, 'dada\_cb' )

The mixed pseudotype is explained as meaning "multiple but not necessarily all" types, and the example of str\_replace(mixed, mixed, mixed) is given where "mixed" means "string or array".  
Keep in mind that this refers to the types of the function's arguments \_after\_ any type juggling.

Parent methods for callbacks should be called 'parent::method', so if you wish to call a non-static parent method via a callback, you should use a callback of  
<?  
// always works  
$callback = array($this, 'parent::method')   
  
// works but gives an error in PHP5 with E\_STRICT if the parent method is not static  
$callback array('parent', 'method');   
?>

Note that (e.g.) usort calls on static methods of classes in a namespace need to be laid out as follows:  
  
usort($arr, array('\Namespace\ClassName', 'functionName'));

I find these explanations and descriptions utterly useless.  
  
After reading all of the PHP Documentation multiple times, only one question remains: Were the college professors high on acid when they decided that it would be a good idea to assign the writing of PHP's Documentation as an extra credit assignment to a group of English as a Second Langauge (ESL) undergraduates who needed to get an "A" COMPUTER SCIENCE 101 in order to avoid a beating from their parents over christmas break.

Uneven division of an integer variable by another integer variable will result in a float by automatic conversion -- you do not have to cast the variables to floats in order to avoid integer truncation (as you would in C, for example):  
  
$dividend = 2;  
$divisor = 3;  
$quotient = $dividend/$divisor;  
print $quotient; // 0.66666666666667

Printing or echoing a FALSE boolean value or a NULL value results in an empty string:  
(string)TRUE //returns "1"  
(string)FALSE //returns ""  
echo TRUE; //prints "1"  
echo FALSE; //prints nothing!

incremental operator ("++") doesn't make type conversion from boolean to int, and if an variable is boolean and equals TRUE than after ++ operation it remains as TRUE, so:  
  
$a = TRUE;   
echo ($a++).$a;  // prints "11"

There are some shorter and faster (at least on my machine) ways to perform a type cast.  
<?php  
$string='12345.678';  
$float=+$string;   
$integer=0|$string;  
$boolean=!!$string;  
?>

in response to bhsmither at gmail.com  
  
It raises a warning because of the bad enquoted variable  
  
<?php  
  
error\_reporting( E\_ALL | E\_STRICT );  
  
$foo['ten'] = 10;        // $foo['ten'] is an array holding an integer at key "ten"  
$str = "{$foo['ten']}";  // works "10"  
$str = "$foo[ten]";      // DO NOT work!

Re: the typecasting between classes post below... fantastic, but slightly flawed. Any class name longer than 9 characters becomes a problem... SO here's a simple fix:  
  
function typecast($old\_object, $new\_classname) {  
  if(class\_exists($new\_classname)) {  
    // Example serialized object segment  
    // O:5:"field":9:{s:5:...   <--- Class: Field  
    $old\_serialized\_prefix  = "O:".strlen(get\_class($old\_object));  
    $old\_serialized\_prefix .= ":\"".get\_class($old\_object)."\":";  
  
    $old\_serialized\_object = serialize($old\_object);  
    $new\_serialized\_object = 'O:'.strlen($new\_classname).':"'.$new\_classname . '":';  
    $new\_serialized\_object .= substr($old\_serialized\_object,strlen($old\_serialized\_prefix));  
   return unserialize($new\_serialized\_object);  
  }  
  else  
   return false;  
}  
  
Thanks for the previous code. Set me in the right direction to solving my typecasting problem. ;)

I found it tricky to check if a posted value was an integer.  
  
<?php  
  
$\_POST['a'] = "42";  
  
is\_int( $\_POST['a'] ); //false  
is\_int( intval( "anything" ) ); //always true  
?>  
  
A method I use for checking if a string represents an integer value.  
  
<?php  
function check\_int( $str )  
                {  
                    return  is\_numeric( $str ) && intval( $str ) - $str == 0;  
                }  
?>

function strhex($string)  
{  
   $hex="";  
   for ($i=0;$i<strlen($string);$i++)  
       $hex.=dechex(ord($string[$i]));  
   return $hex;  
}  
function hexstr($hex)  
{  
   $string="";  
   for ($i=0;$i<strlen($hex)-1;$i+=2)  
       $string.=chr(hexdec($hex[$i].$hex[$i+1]));  
   return $string;  
}  
  
to convert hex to str and vice versa

If you want to do not only typecasting between basic data types but between classes, try this function. It converts any class into another. All variables that equal name in both classes will be copied.  
  
function typecast($old\_object, $new\_classname) {  
  if(class\_exists($new\_classname)) {  
    $old\_serialized\_object = serialize($old\_object);  
    $new\_serialized\_object = 'O:' . strlen($new\_classname) . ':"' . $new\_classname . '":' .   
                             substr($old\_serialized\_object, $old\_serialized\_object[2] + 7);  
    return unserialize($new\_serialized\_object);  
  }  
  else  
    return false;  
}  
  
Example:  
  
class A {  
  var $secret;  
  function A($secret) {$this->secret = $secret;}  
  function output() {echo("Secret class A: " . $this->secret);}  
}  
  
class B extends A {  
  var $secret;  
  function output() {echo("Secret class B: " . strrev($this->secret));}  
}  
  
$a = new A("Paranoia");  
$b = typecast($a, "B");  
  
$a->output();  
$b->output();  
echo("Classname \$a: " . get\_class($a) . "Classname \$b: " . get\_class($b));  
  
Output of the example code above:  
  
Secret class A: Paranoia  
Secret class B: aionaraP  
Classname $a: a  
Classname $b: b

Checking for strings to be integers?  
How about if a string is a float?  
  
<?php  
  
/\* checks if a string is an integer with possible whitespace before and/or after, and also isolates the integer \*/  
$isInt=preg\_match('/^\s\*([0-9]+)\s\*$/', $myString, $myInt);  
  
echo 'Is Integer? ',  ($isInt) ? 'Yes: '.$myInt[1] : 'No', "\n";  
  
/\* checks if a string is an integer with no whitespace before or after  \*/  
$isInt=preg\_match('/^[0-9]+$/', $myString);  
  
echo 'Is Integer? ',  ($isInt) ? 'Yes' : 'No', "\n";  
  
/\* When checking for floats, we assume the possibility of no decimals needed.  If you MUST require decimals (forcing the user to type 7.0 for example) replace the sequence:  
[0-9]+(\.[0-9]+)?   
with   
[0-9]+\.[0-9]+   
\*/  
  
/\* checks if a string is a float with possible whitespace before and/or after, and also isolates the number \*/  
$isFloat=preg\_match('/^\s\*([0-9]+(\.[0-9]+)?)\s\*$/', $myString, $myNum);  
  
echo 'Is Number? ',  ($isFloat) ? 'Yes: '.$myNum[1] : 'No', "\n";  
  
/\* checks if a string is a float with no whitespace before or after \*/  
$isInt=preg\_match('/^[0-9]+(\.[0-9]+)?$/', $myString);  
  
echo 'Is Number? ',  ($isFloat) ? 'Yes' : 'No', "\n";  
  
?>

Type casting from string to int and vice versa is probably the most common conversation. PHP does this very simply through the +. and .= operators, removing any explicit casting:  
  
<?php  
$x = 1;  
var\_dump($x); // int(1)  
$x .= 1;  
var\_dump($x); // string(2) "11"; also an empty string ("") would cast to string without changing $x  
  
$x = "1";  
var\_dump($x);  // string(1) "1"  
$x += 1;  
var\_dump($x); // int(2); also a zero value (0) would cast to int without changing $x  
?>

Cast a string to binary using PHP < 5.2.1  
  
$binary = unpack('c\*', $string);

If you have a boolean, performing increments on it won't do anything despite it being 1.  This is a case where you have to use a cast.  
  
<html>  
<body> <!-- don't want w3.org to get mad... -->  
<?php  
$bar = TRUE;  
?>  
I have <?=$bar?> bar.  
<?php  
$bar++;  
?>  
I now have <?=$bar?> bar.  
<?php  
$bar = (int) $bar;  
$bar++;  
?>  
I finally have <?=$bar?> bar.  
</body>  
</html>  
  
That will print  
  
I have 1 bar.  
I now have 1 bar.  
I finally have 2 bar.

Casting objects to arrays is a pain. Example:  
  
<?php  
  
class MyClass {  
  
    private $priv = 'priv\_value';  
    protected $prot = 'prot\_value';  
    public $pub = 'pub\_value';  
    public $MyClasspriv = 'second\_pub\_value';  
  
}  
  
$test = new MyClass();  
echo '<pre>';  
print\_r((array) $test);  
  
/\*  
Array  
(  
    [MyClasspriv] => priv\_value  
    [\*prot] => prot\_value  
    [pub] => pub\_value  
    [MyClasspriv] => second\_pub\_value  
)  
\*/  
  
?>  
  
Yes, that looks like an array with two keys with the same name and it looks like the protected field was prepended with an asterisk. But that's not true:  
  
<?php  
  
foreach ((array) $test as $key => $value) {  
    $len = strlen($key);  
    echo "{$key} ({$len}) => {$value}<br />";  
    for ($i = 0; $i < $len; ++$i) {  
        echo ord($key[$i]) . ' ';  
    }  
    echo '<hr />';  
}  
  
/\*  
MyClasspriv (13) => priv\_value  
0 77 121 67 108 97 115 115 0 112 114 105 118  
\*prot (7) => prot\_value  
0 42 0 112 114 111 116  
pub (3) => pub\_value  
112 117 98  
MyClasspriv (11) => second\_pub\_value  
77 121 67 108 97 115 115 112 114 105 118  
\*/  
  
?>  
  
The char codes show that the protected keys are prepended with '\0\*\0' and private keys are prepended with '\0'.\_\_CLASS\_\_.'\0' so be careful when playing around with this.

If you want to convert a string automatically to float or integer (e.g. "0.234" to float and "123" to int), simply add 0 to the string - PHP will do the rest.  
  
e.g.  
  
$val = 0 + "1.234";  
(type of $val is float now)  
  
$val = 0 + "123";  
(type of $val is integer now)

The object casting methods presented here do not take into account the class hierarchy of the class you're trying to cast your object into.  
  
/\*\*  
     \* Convert an object to a specific class.  
     \* @param object $object  
     \* @param string $class\_name The class to cast the object to  
     \* @return object  
     \*/  
    public static function cast($object, $class\_name) {  
        if($object === false) return false;  
        if(class\_exists($class\_name)) {  
            $ser\_object     = serialize($object);  
            $obj\_name\_len     = strlen(get\_class($object));  
            $start             = $obj\_name\_len + strlen($obj\_name\_len) + 6;  
            $new\_object      = 'O:' . strlen($class\_name) . ':"' . $class\_name . '":';  
            $new\_object     .= substr($ser\_object, $start);  
            $new\_object     = unserialize($new\_object);  
            /\*\*  
             \* The new object is of the correct type but  
             \* is not fully initialized throughout its graph.  
             \* To get the full object graph (including parent  
             \* class data, we need to create a new instance of   
             \* the specified class and then assign the new   
             \* properties to it.  
             \*/  
            $graph = new $class\_name;  
            foreach($new\_object as $prop => $val) {  
                $graph->$prop = $val;  
            }  
            return $graph;  
        } else {  
            throw new CoreException(false, "could not find class $class\_name for casting in DB::cast");  
            return false;  
        }  
    }

It seems (unset) is pretty useless. But for people who like to make their code really compact (and probably unreadable). You can use it to use an variable and unset it on the same line:  
  
Without cast:   
  
<?php  
  
$hello = 'Hello world';  
print $hello;  
unset($hello);  
  
?>  
  
With the unset cast:   
  
<?php  
  
$hello = 'Hello world';  
$hello = (unset) print $hello;  
  
?>  
  
Hoorah, we lost another line!

It would be useful to know the precedence (for lack of a better word) for type juggling.  This entry currently explains that "if either operand is a float, then both operands are evaluated as floats, and the result will be a float" but could (and I think should) provide a hierarchy that indicates, for instance, "between an int and a boolean, int wins; between a float and an int, float wins; between a string and a float, string wins" and so on (and don't count on my example accurately capturing the true hierarchy, as I haven't actually done the tests to figure it out).  Thanks!

namaroulis stated "I found it tricky to check if a posted value was an integer"; to test if a variable is a number or a numeric string (such as form input, which is always a string), you must use is\_numeric():  
  
<?php  
$\_POST['a'] = "42";  
  
is\_numeric( $\_POST['a'] ); // true  
?>

<?php  
$foo['ten'] = 10;            // $foo['ten'] is an array holding an integer at key "ten"  
$str = "$foo['ten']";        // throws T\_ENCAPSED\_AND\_WHITESPACE error  
$str = "$foo[ten]";          // works because constants are skipped in quotes  
$fst = (string) $foo['ten']; // works with clear intention  
?>

json\_decode users consider this, when casting stdClass to array:  
<?php  
$obj = new stdClass();  
$obj->{"2"} = "id";  
$arr = (array) $obj;  
$result = isset($arr["2"]) || array\_key\_exists(2, $arr); // false  
?>  
..though casting is at least 2x faster than foreach.

WHERE'S THE BEEF?  
  
Looks like type-casting user-defined objects is a real pain, and ya gotta be nuttin' less than a brain jus ta cypher-it. But since PHP supports OOP, you can add the capabilities right now. Start with any simple class.  
<?php  
class Point {  
  protected $x, $y;  
  
  public function \_\_construct($xVal = 0, $yVal = 0) {  
    $this->x = $xVal;  
    $this->y = $yVal;  
  }  
  public function getX() { return $this->x; }  
  public function getY() { return $this->y; }  
}  
  
$p = new Point(25, 35);  
echo $p->getX();      // 25  
echo $p->getY();      // 35  
?>  
Ok, now we need extra powers. PHP gives us several options:  
  A. We can tag on extra properties on-the-fly using everyday PHP syntax...  
    $p->z = 45; // here, $p is still an object of type [Point] but gains no capability, and it's on a per-instance basis, blah.  
  B. We can try type-casting it to a different type to access more functions...  
    $p = (SuperDuperPoint) $p; // if this is even allowed, I doubt it. But even if PHP lets this slide, the small amount of data Point holds would probably not be enough for the extra functions to work anyway. And we still need the class def + all extra data. We should have just instantiated a [SuperDuperPoint] object to begin with... and just like above, this only works on a per-instance basis.  
  C. Do it the right way using OOP - and just extend the Point class already.  
<?php  
class Point3D extends Point {  
  protected $z;                                // add extra properties...  
  
  public function \_\_construct($xVal = 0, $yVal = 0, $zVal = 0) {  
    parent::\_\_construct($xVal, $yVal);  
    $this->z = $zVal;  
  }  
  public function getZ() { return $this->z; }  // add extra functions...  
}  
  
$p3d = new Point3D(25, 35, 45);  // more data, more functions, more everything...  
echo $p3d->getX();               // 25  
echo $p3d->getY();               // 35  
echo $p3d->getZ();               // 45  
?>  
Once the new class definition is written, you can make as many Point3D objects as you want. Each of them will have more data and functions already built-in. This is much better than trying to beef-up any "single lesser object" on-the-fly, and it's way easier to do.

@alexgr (20-Jun-2008)  
  
Correct me if I'm wrong, but that is not a cast, it might be useful sometimes, but the IDE will not reflect what's really happening:  
  
<?php  
class MyObject {  
    /\*\*  
     \* @param MyObject $object  
     \* @return MyObject  
     \*/  
    static public function cast(MyObject $object) {  
        return $object;  
    }  
    /\*\* Does nothing \*/  
    function f() {}  
}  
  
class X extends MyObject {  
    /\*\* Throws exception \*/  
    function f() { throw new exception(); }  
}  
  
$x = MyObject::cast(new X);  
$x->f(); // Your IDE tells 'f() Does nothing'  
?>  
  
However, when you run the script, you will get an exception.

For a Cast to a User Defined Object you can define a cast method:  
  
class MyObject {  
    /\*\*  
     \* @param MyObject $object  
     \* @return MyObject  
     \*/  
    static public function cast(MyObject $object) {  
        return $object;  
    }  
}  
  
In your php page code you can:  
$myObject = MyObject::cast($\_SESSION["myObject"]);  
  
Then, PHP will validate the value and your IDE will help you.

In my much of my coding I have found it necessary to type-cast between objects of different class types.  
  
More specifically, I often want to take information from a database, convert it into the class it was before it was inserted, then have the ability to call its class functions as well.  
  
The following code is much shorter than some of the previous examples and seems to suit my purposes.  It also makes use of some regular expression matching rather than string position, replacing, etc.  It takes an object ($obj) of any type and casts it to an new type ($class\_type).  Note that the new class type must exist:  
  
function ClassTypeCast(&$obj,$class\_type){  
    if(class\_exists($class\_type,true)){  
        $obj = unserialize(preg\_replace"/^O:[0-9]+:\"[^\"]+\":/i",   
          "O:".strlen($class\_type).":\"".$class\_type."\":", serialize($obj)));  
    }  
}

For some reason the code-fix posted by philip\_snyder at hotmail dot com [27-Feb-2004 02:08]  
didn't work for me neither with long\_class\_names nor with short\_class\_names. I'm using PHP v4.3.5 for Linux.  
Anyway here's what I wrote to solve the long\_named\_classes problem:  
  
<?php  
function typecast($old\_object, $new\_classname) {  
    if(class\_exists($new\_classname)) {  
        $old\_serialized\_object = serialize($old\_object);  
        $old\_object\_name\_length = strlen(get\_class($old\_object));  
        $subtring\_offset = $old\_object\_name\_length + strlen($old\_object\_name\_length) + 6;  
        $new\_serialized\_object  = 'O:' . strlen($new\_classname) . ':"' . $new\_classname . '":';  
        $new\_serialized\_object .= substr($old\_serialized\_object, $subtring\_offset);  
        return unserialize($new\_serialized\_object);  
     } else {  
         return false;  
     }  
}  
?>

Just a little experiment on the (unset) type cast:  
  
<?php  
$var = 1;  
$var\_unset = (unset) $var;  
$var\_ref\_unset &= (unset) $var;  
var\_dump($var);  
var\_dump($var\_unset);  
var\_dump($var\_ref\_unset);  
?>  
  
output:  
int(1)  
NULL  
int(0)

IMAGINATION REQUIRED...  
  
We can be a witness to PHP's 'type-jugglin' in real-time with a simple implementation of a MemoryMap. For the sake our purposes, pretend that this is an empty MemoryMap.  
+-------+------+------+-------+  
| index | $var | type | value |  
+-------+------+------+-------+  
|     1 |  --- | NULL |  null |  
|     2 |  --- | NULL |  null |  
|     3 |  --- | NULL |  null |  
|     4 |  --- | NULL |  null |  
+-------+------+------+-------+  
  
<?php  
# create some variables...  
$a = 10;  
$b = "Hello";  
$c = array(55.45, 98.65);  
# Now look at map...  
?>  
+-------+-------+---------+--------+  
| index |  $var |    type |  value |  
+-------+-------+---------+--------+  
|     1 |    $a | INTEGER |     10 |  
|     2 |    $b |  STRING |  Hello |  
|     3 | $c[0] |   FLOAT |  55.45 |  
|     4 | $c[1] |   FLOAT |  98.65 |  
+-------+-------+---------+--------+  
<?php  
# Now, change the variable types...  
$a = "Bye";  
$b = 2;  
$c[0] = "Buy";  
$c[1] = "Now!";  
#Look at map...  
?>  
+-------+-------+---------+--------+  
| index |  $var |    type |  value |  
+-------+-------+---------+--------+  
|     1 |    $a |  STRING |    Bye | <- used to be INTEGER  
|     2 |    $b | INTEGER |      2 | <- used to be STRING  
|     3 | $c[0] |  STRING |    Buy | <- used to be FLOAT  
|     4 | $c[1] |  STRING |  Right | <- used to be FLOAT  
+-------+-------+---------+--------+