

Teile des PHP Manuals

Quelle: <http://www.php.net>

© 1997-2005 PHP-Dokumentationsgruppe

Verteilt von Manuel Blechschmidt newthinking Store GmbH

<http://store.newthinking.de>

Kapitel 10. Grundlagen der Syntax

Inhaltsverzeichnis

[Den HTML-Bereich der Datei verlassen](#)

[Abgrenzung von Anweisungen](#)

[Kommentare](#)

Den HTML-Bereich der Datei verlassen

Während des Parsens einer Datei geht PHP den Text solange einfach durch, bis einer der speziellen Tags gefunden wird, der PHP mitteilt, dass ab nun mit der Interpretation des Textes als PHP Code zu beginnen ist. Der Parser führt nun den Code solange aus, bis er auf einen schließenden PHP Tag stößt, welcher dem Parser mitteilt, den Text ab hier wieder nur einfach durchzugehen. Das ist der Mechanismus der es erlaubt, PHP Code in HTML einzubinden: alles außerhalb der PHP Tags wird einfach alleine gelassen, während alles innerhalb dieser Tags als Code geparsed wird.

Es gibt vier Arten von Tags, welche die Blöcke mit PHP Code kennzeichnen. Davon sind nur zwei immer verfügbar (`<?php. . ?>` und `<script language="php">. . .</script>`), die anderen können in der Konfigurationsdatei `php.ini` aktiviert bzw. deaktiviert werden. Die Tags in der Kurzform bzw. im ASP-Stil mögen zwar praktisch erscheinen, sind jedoch nicht so portabel wie die längeren Versionen. Wenn Sie des Weiteren beabsichtigen, PHP Code in XML oder XHTML einzubinden, werden Sie die XML-konformen `<?php. . ?>` verwenden müssen.

Die von PHP unterstützten Tags sind:

Beispiel 10-1. Möglichkeiten, den HTML-Bereich zu verlassen

1. `<?php echo("In XHTML- oder XML-konformen Dokumenten benutzen Sie diese Syntax\n"); ?>`
2. `<? echo ("Die einfachste Möglichkeit: eine SGML-´processing instruction´\n"); ?>`
`<?= expression ?>` Dies ist eine Abkürzung für "`<? echo expression ?>`"
3. `<script language="php">`
`echo ("manche Editoren(wie FrontPage) mögen`
`keine ´processing instructions´");`
`</script>`

```
4. <% echo ("Optional können Sie auch Tags im ASP-Stil verwenden");
%>
<%= $variable; # Das ist ein Abkürzung fuer "<% echo . . ." %>
```

Die erste Methode (<?php. .?>) wird bevorzugt, da sie auch die Verwendung von PHP in XML-konformen Code, wie XHTML, erlaubt.

Die zweite Methode ist nicht immer verfügbar. Die Kurzform der Tags ist nur möglich, wenn diese zuvor aktiviert wurde. Dies kann mittels der Funktion **short_tags()** (nur PHP 3), dem Setzen der Konfigurationsoption [short_open_tag](#) in der PHP Konfigurationsdatei, oder durch das Kompilieren von PHP mit der Option --enable-short-tags. Auch wenn diese Methode standardmäßig in der php.ini.dist aktiviert ist, wird die Verwendung dieser Kurzform nicht empfohlen.

Die vierte Methode ist nur verfügbar, wenn die Konfigurationsoption [asp_tags](#) aktiviert ist.

Anmerkung: Die Unterstützung der Tags im ASP-Stil wurde in Version 3.0.4. hinzugefügt.

Anmerkung: Die Verwendung der 'short tags' sollten Sie vermeiden, wenn Sie Applikationen oder Bibliotheken entwickeln, die verteilt oder auf PHP-Servern eingesetzt werden soll, die nicht unter Ihrer Kontrolle stehen, da die 'short tags' auf dem einzusetzenden Server nicht unterstützt werden könnten. Stellen Sie also sicher, dass für portablen, weiterverteilbaren Code keine 'short tags' verwendet werden.

Der schließende Tag für den PHP-Block schließt eine sofort folgende Zeilenschaltung mit ein, falls diese vorhanden ist. Außerdem beinhaltet der schließende Tag ein implizites Semikolon; Sie müssen also die letzte Zeile eines PHP-Blocks nicht mit einem Semikolon beenden.

PHP ermöglicht auch die Verwendung folgender Strukturen:

Beispiel 10-2. Erweitertes Verlassen

```
<?php
if ($expression) {
    ?>
    <strong>Das ist richtig.</strong>
    <?php
} else {
    ?>
    <strong>Das ist falsch.</strong>
    <?php
}
?>
```

Dies funktioniert wie erwartet, da PHP nach dem Auffinden eines ?> schließenden Tags einfach alles ausgibt, bis es wieder auf einen öffnenden Tag stößt. Das obige Beispiel ist natürlich gekünstelt, aber für die Ausgabe von

großen Textblöcken ist der Ausstieg aus dem Parse-Modus generell effizienter, als den gesamten Text durch [echo\(\)](#), [print\(\)](#), etc. zu jagen.

Kapitel 11. Typen

Inhaltsverzeichnis

[Einführung](#)

[Boolscher Typ](#)

[Integer Typen](#)

[Fließkomma-Zahlenwerte](#)

[Strings / Zeichenketten](#)

[Arrays](#)

[Objekte](#)

[Resource](#)

[NULL](#)

[In dieser Dokumentation verwendete Pseudo-Typen](#)

[Typen-Tricks](#)

Einführung

PHP unterstützt acht primitive Typen.

Vier skalare Typen:

- Boolean
- Integer
- Fließkomma-Zahl (float)
- String / Zeichenkette

Zwei zusammengesetzte Typen:

- Array
- Object

Und zuletzt zwei spezielle Typen:

- Resource
- **[NULL](#)**

Für die bessere Lesbarkeit führt dieses Handbuch ein paar [Pseudo-Typen](#) ein:

- [Mixed](#)
- [Number](#)
- [Callback](#)

Sie werden auch ein paar Hinweise auf den Typ "Double" finden. Betrachten Sie Double als dasselbe wie Float. Die beiden Bezeichnungen existieren nur aus

historischen Gründen.

Der Typ einer Variablen wird normalerweise nicht vom Programmierer bestimmt. Zur Laufzeit von PHP wird entschieden, welchen Typs eine Variable ist, abhängig vom Zusammenhang in dem die Variable benutzt wird.

Anmerkung: Um den Typ und den Wert eines bestimmten [Ausdrucks \(Expression\)](#) zu überprüfen, können Sie [var_dump\(\)](#) benutzen.

Anmerkung: Wenn Sie zur Fehlersuche einfach nur eine lesbare Darstellung eines Typs benötigen, benutzen Sie [gettype\(\)](#). Um auf einen bestimmten Typ zu prüfen, sollten Sie *nicht* [gettype\(\)](#) benutzen. Stattdessen sollten Sie die *is_type* Funktionen verwenden. Ein paar Beispiele:

```
<?php
$bool = TRUE; // ein Boolean (Wahrheitswert)
$str = "foo"; // ein String (Zeichenkette)
$int = 12; // ein Integer (Ganzzahl)

echo gettype($bool); // gibt "boolean" aus
echo gettype($str); // gibt "string" aus

// Falls es ein Integer ist, erhöhe ihn um vier
if (is_int($int)) {
    $int += 4;
}

// Falls $bool ein String ist, gebe ihn aus
// (gibt überhaupt nichts aus)
if (is_string($bool)) {
    echo "String: $bool";
}
?>
```

Wenn sie die Umwandlung in einen bestimmten Variablen-Typ erzwingen wollen, erreichen Sie dies entweder durch [Typ-Umwandlung](#) oder durch Gebrauch der Funktion [settype\(\)](#).

Beachten Sie, dass eine Variable abhängig vom Typ, dem die Variable zu dem Zeitpunkt entspricht, in bestimmten Situationen unterschiedlich ausgewertet werden kann. Weitere Informationen entnehmen Sie dem Abschnitt zur [Typ-Veränderung](#). Schauen Sie sich außerdem [PHP type comparison tables](#) an, wenn Sie an Beispielen verschiedener typenbezogener Vergleiche interessiert sind.

Kapitel 12. Variablen

Inhaltsverzeichnis

[Grundlegendes](#)

[Vordefinierte Variablen](#)
[Geltungsbereich von Variablen](#)
[Variable Variablen](#)
[Variablen außerhalb von PHP](#)

Grundlegendes

Variablen werden in PHP dargestellt durch ein Dollar-Zeichen (\$) gefolgt vom Namen der Variablen. Bei Variablen-Namen wird zwischen Groß- und Kleinschreibung unterschieden (case-sensitive).

Variablen-Namen werden in PHP nach den gleichen Regeln wie andere Bezeichner erstellt. Ein gültiger Variablen-Name beginnt mit einem Buchstaben oder einem Unterstrich (" _"), gefolgt von einer beliebigen Anzahl von Buchstaben, Zahlen oder Unterstrichen. Als regulärer Ausdruck (regular expression) würde das wie folgt ausgedrückt: '[a-zA-Z_\x7f-\xff][a-zA-Z0-9_\x7f-\xff]*'.

Anmerkung: Unserem Zweck entspricht also ein Buchstabe von a bis z bzw. A bis Z oder einem ASCII-Zeichen von 127 bis 255 (0x7f bis 0xff).

```
<?php
$var = "Du";
$vaR = "und";
$Var = "ich";
$vAr = "wir lernen PHP"
echo "$var $vaR $Var, $vAr"; // gibt "Du und ich, wir lernen PHP" aus

$4site = 'nicht jetzt'; // ungültig, da Anfang eine Zahl
$_4site = 'nicht jetzt'; // gültig, da Unterstrich am Anfang
$täbyte = 'irgendwas'; // gültig, da 'ä' dem (Erweiterten)
ASCII-Wert 228 entspricht
?>
```

Variablen werden in PHP 3 durch ihren Wert bestimmt. Das heisst, wenn Sie einer Variablen einen Ausdruck zuweisen, wird der gesamte Inhalt des Originalausdrucks in die Zielvariable kopiert. Die Folge ist, dass eine Variable, die ihren Inhalt von einer anderen Variablen erhalten hat, ihren Inhalt behält, auch wenn Sie danach den Inhalt der anderen (Quell- / Ursprungs-)Variablen ändern. Die Inhalte der Ziel- und Quellvariablen sind also insoweit unabhängig voneinander. Für weitere Informationen lesen Sie bitte das Kapitel unter [Expressions / Ausdrücke](#).

PHP 4 bietet eine andere Möglichkeit der Wertzuweisung bei Variablen: [Zuweisung durch Referenzierung](#). Das bedeutet, dass der Wert der neuen Variablen eine Referenz zur Ursprungs-Variablen darstellt (mit anderen Worten: Der Wert ist ein Alias bzw. Zeiger auf den Inhalt der Ursprungsvariablen). Beide Variablen zeigen also auf die selbe(n) Speicherstelle(n). Änderungen der neuen

Variablen ändern auch deren Ursprungs-Variable und umgekehrt. Der Wert / Inhalt wird also nicht kopiert. Die Übertragung geschieht dadurch auch schneller als in PHP 3. Dies wird sich aber nur bei umfangreichen Schleifen oder bei der Übertragung von grossen [Arrays](#) oder [Objekten](#) bemerkbar machen.

Für die Zuweisung per Referenz müssen Sie lediglich ein & der (Ausgangs-, Quell-) Variablen voranstellen, die sie einer anderen Variablen zuweisen wollen. Der folgende Skript- Ausschnitt wird zweimal 'Mein Name ist Bob' ausgeben:

```
<?php
$foo = 'Bob';           // 'Bob' der Variablen $foo zuweisen.
$bar = &$foo;          // Zeiger auf $foo in $bar erzeugen.
$bar = "Ich hei&szlig;e $bar"; // $bar verändern...
echo $bar;
echo $foo;             // $foo wurde dadurch ebenfalls verändert.
?>
```

Zu beachten ist, dass nur Variablenbezeichner referenziert werden können.

```
<?php
$foo = 25;
$bar = &$foo; // Gültige Zuweisung.
$bar = &(24 * 7); // Ungültig, da kein Variablenbezeichner
                // zugewiesen wird.

function test() {
    return 25;
}

$bar = &test(); // Ungültig.
?>
```

Kapitel 13. Konstanten

Inhaltsverzeichnis

[Syntax](#)

[Magische Konstanten](#)

Eine Konstante ist ein Bezeichner (Name) für eine simple Variable. Wie der Name schon ausdrückt, kann sich der Wert einer Konstanten zur Laufzeit eines Skripts nicht ändern (eine Ausnahme bilden die [Magischen Konstanten](#), die aber tatsächlich keine Konstanten sind.) Eine Konstante unterscheidet zwischen Groß- und Kleinschreibung (case-sensitive). Nach gängiger Konvention werden Konstanten immer in Großbuchstaben geschrieben.

Der Name einer Konstanten folgt den gleichen Regeln wie alle anderen Bezeichner in PHP. Ein gültiger Name beginnt mit einem Buchstaben oder einem Unterstrich, gefolgt von beliebig vielen Buchstaben, Nummern oder Unterstrichen. Als regulärer Ausdruck geschrieben: `[a-zA-Z_\x7f-\xff][a-zA-Z0-`

9_ |x7f-|xff]*

Beispiel 13-1. Gültige und ungültige Namen für Konstanten

```
<?php
// Gültige Namen für Konstanten
define("F00",    "irgendwas");
define("F002",   "etwas anderes");
define("F00_BAR", "irgendwas ganz anderes")

// Invalid constant names
define("2F00",   "irgendwas");

// Folgendes ist korrekt, sollte aber vermieden werden:
// Eines Tages könnte es in PHP eine Magische Konstante
// __F00__ geben und Ihr Skript wird dadurch nicht mehr
// wie gewünscht funktionieren.

define("__F00__", "irgendwas");

?>
```

Anmerkung: Für unsere Zwecke ist ein Buchstabe a-z, A-Z und die ASCII-Zeichen von 127 bis 255 (0x7f-0xff).

Wie bei [superglobals](#) ist der Gültigkeitsbereich einer Konstanten global. Unabhängig vom Gültigkeitsbereich können Sie in Ihrem Skript überall auf eine Konstante zugreifen. Für mehr Information zum Gültigkeitsbereich lesen Sie bitte den Abschnitt über den [Geltungsbereich von Variablen](#).

Syntax

Eine Konstante können Sie über die Funktion [define\(\)](#) definieren. Einmal definiert, kann eine Konstante weder verändert noch gelöscht werden.

Konstanten können nur skalare Daten ([boolean](#), [integer](#), [float](#) und [string](#)) enthalten.

Den Wert einer Konstanten erhalten Sie ganz einfach durch die Angabe ihres Namens. Einer Konstanten sollten Sie *nicht*, wie bei Variablen, ein `$` voranstellen. Ebenso können Sie die Funktion [constant\(\)](#) benutzen um den Wert einer Konstanten auszulesen, wenn Sie den Namen der Konstanten dynamisch erhalten wollen. Benutzen Sie [get_defined_constants\(\)](#) um eine Liste aller definierten Konstanten zu erhalten.

Anmerkung: Konstanten und (globale) Variablen befinden sich in unterschiedlichen Namensräumen. Das hat zum Beispiel zur Folge, dass `TRUE` und `$TRUE` grundsätzlich unterschiedlich sind.

Falls Sie eine undefinierte Konstante verwenden nimmt PHP an, dass Sie den Namen der Konstanten selber meinen, genauso als ob Sie einen **string** (CONSTANT vs "CONSTANT") angegeben hätten. Falls das passiert, wird Ihnen ein Fehler vom Typ **E_NOTICE** ausgegeben. Lesen Sie hier nach, warum die Angabe `$foo[bar]` falsch ist (zumindest solange Sie nicht zuerst `bar` mittels **define()** als Konstante definiert haben). Möchten Sie einfach nur nachprüfen, ob eine Konstante definiert ist, benutzen Sie die Funktion **defined()** für diesen Zweck.

Das hier sind die Unterschiede zwischen Konstanten und Variablen:

- Konstanten haben kein Dollarzeichen (\$) vorangestellt;
- Konstanten können nur über die Funktion **define()** definiert werden, nicht durch einfache Zuweisung;
- Konstanten können überall definiert werden und auf Ihren Wert können Sie ohne Rücksicht auf Namensraumregeln von Variablen zugreifen;
- Sobald Konstanten definiert sind, können sie nicht neu definiert oder gelöscht werden; und
- Konstanten können nur skalare Werte haben.

Beispiel 13-2. Definition von Konstanten

```
<?php
define("CONSTANT", "Hallo Welt.");
echo CONSTANT; // Ausgabe: "Hallo Welt."
echo Constant; // Ausgabe: "Constant" und eine Notice.
?>
```

Kapitel 14. Ausdrücke

Ausdrücke (Expressions) sind die wichtigsten Bausteine von PHP. In PHP ist fast alles, was geschrieben wird, ein Ausdruck. Die einfachste, aber auch zutreffendste Definition für einen Ausdruck ist "alles, was einen Wert hat".

Die grundlegendste Formen von Ausdrücken sind Konstanten und Variablen. Wenn man "`$a = 5`" schreibt, weist man `$a` den Ausdruck `'5'` zu. `'5'` hat offensichtlich den Wert 5. Anders ausgedrückt: `'5'` ist ein Ausdruck mit dem Wert 5 (in diesem Fall ist `'5'` eine Integer-Konstante).

Nach dieser Zuweisung würde man erwarten, dass der Wert von `$a` nun ebenfalls 5 ist, wenn man also `$b = $a` schreibt, sollte dasselbe dabei herauskommen, als hätte man `$b = 5` geschrieben. Anders ausgedrückt: `$a` wäre ebenfalls ein Ausdruck mit dem Wert 5. Wenn alles richtig funktioniert, wird genau das passieren.

Etwas kompliziertere Beispiele für Ausdrücke sind Funktionen:

```
function foo () {  
    return 5;  
}
```

Angenommen, Sie sind mit dem Konzept von Funktionen vertraut (wenn Sie es nicht sind, lesen Sie das Kapitel über Funktionen), dann würden Sie annehmen, dass die Eingabe von $\$c = \text{foo}()$ grundsätzlich dasselbe bedeutet, als würde man schreiben $\$c = 5$, und genau das trifft zu. Funktionen sind Ausdrücke mit dem Wert ihres Rückgabewertes. Da $\text{foo}()$ den Wert 5 zurückgibt, ist der Wert des Ausdruckes 'foo()' 5. Normalerweise geben Funktionen nicht einfach einen statischen Wert zurück, sondern berechnen irgendetwas.

Natürlich müssen Werte in PHP keine Integer-Zahlen sein, und oft sind sie es auch nicht. PHP unterstützt drei skalare Datentypen: integer values (Integer-Zahlen), floating point values (Fließkommazahlen) und string values (Zeichenketten). (Skalare sind Datentypen, die man nicht in kleinere Stücke 'brechen' kann, im Gegensatz zu Arrays). PHP unterstützt auch zwei zusammengesetzte (nicht-skalare) Datentypen: Arrays und Objekte. Jeder dieser Datentypen kann Variablen zugewiesen und von Funktionen zurückgegeben werden.

Bis hierhin sollten Benutzer von PHP/FI 2 keine Veränderung bemerkt haben. PHP fasst den Begriff 'Ausdruck' aber noch viel weiter, wie es auch andere Programmiersprachen tun. PHP ist in dem Sinne eine ausdrucksorientierte Sprache, dass fast alles ein Ausdruck ist. Zurück zu dem Beispiel, mit dem wir uns schon beschäftigt haben: $\$a = 5$. Es ist einfach zu erkennen, dass hier zwei Werte enthalten sind: Der Wert der Integer-Konstanten '5' und der Wert von $\$a$, der auf 5 geändert wird. In Wirklichkeit ist aber noch ein zusätzlicher Wert enthalten, nämlich der Wert der Zuweisung selbst. Die Zuweisung selbst enthält den zugewiesenen Wert, in diesem Fall 5. In der Praxis bedeutet dies, dass $\$a = 5$, egal was es tut, immer einen Ausdruck mit dem Wert 5 darstellt. Folglich ist $\$b = (\$a = 5)$ gleichbedeutend mit $\$a = 5; \$b = 5;$ (ein Semikolon markiert das Ende einer Anweisung). Da Wertzuweisungen von rechts nach links geparkt werden, kann man auch $\$b = \$a = 5$ schreiben.

Ein anderes gutes Beispiel für die Ausdrucksorientierung von PHP sind Prä- und Post-Inkrement sowie die entsprechenden Dekremente. Benutzer von PHP/FI 2 und vielen anderen Sprachen sind vermutlich mit den Notationen 'variable++' und 'variable--' vertraut. Dies sind Inkrement- und Dekrement-Operatoren. IN PHP/FI 2 hat das Statement $\$a++$ keinen Wert (es ist kein Ausdruck) und daher kann man es nicht als Wert zuweisen oder in irgendeiner Weise benutzen. PHP erweitert die Eigenschaften von Dekrement und Inkrement, indem es die beiden ebenfalls zu Ausdrücken macht. In PHP gibt es, wie in C, zwei Arten von Inkrementen - Prä-Inkrement und Post-Inkrement. Grundsätzlich erhöhen sowohl Prä- als auch Post-Inkrement den Wert der Variable, und der Effekt auf die Variable ist derselbe. Der Unterschied ist der Wert des Inkrement-Ausdruckes. Das Prä-Inkrement, das $++\$variable$ geschrieben wird, enthält als Wert den Wert der erhöhten Variable (PHP erhöht den Wert der Variable, bevor es ihren Wert ausliest, daher der Name 'PRÄ-Inkrement'). Das Post-Inkrement, das $\$variable++$ geschrieben wird, enthält dagegen den ursprünglichen Wert der

Variablen vor der Erhöhung (PHP erhöht den Wert der Variablen, nachdem es ihren Wert ausgelesen hat, daher der Name 'POST-Inkrement').

Ein sehr gebräuchlicher Typ von Ausdrücken sind Vergleichsausdrücke. Diese Ausdrücke geben entweder 0 (=FALSCH) oder 1 (=WAHR) zurück. PHP unterstützt > (größer), >= (größer oder gleich), == (gleich), != (ungleich), < (kleiner), und <= (kleiner oder gleich). Diese Ausdrücke werden meistens in bedingten Anweisungen, wie z. B. in *if*-Anweisungen, verwendet.

Im letzten Beispiel für Ausdrücke befassen wir uns mit kombinierten Zuweisungs- und Operator-Ausdrücken. Wir haben schon gezeigt, wie man den Wert von \$a um 1 erhöht, nämlich indem man einfach '\$a++' oder '++\$a' schreibt. Aber was tut man, wenn man den Wert um mehr als eins erhöhen will, z. B. um 3? Man könnte mehrer Male '\$a++' schreiben, aber das ist offensichtlich weder effizient noch sehr komfortabel. Viel üblicher ist es, einfach '\$a = \$a + 3' zu schreiben. '\$a + 3' gibt den Wert von \$a plus 3 zurück, dieser wird wieder \$a zugewiesen, was dazu führt, dass \$a nun um 3 erhöht wurde. In PHP - wie in einigen anderen Programmiersprachen, z. B. in C - kann man dies aber noch kürzer schreiben, was mit der Zeit klarer wird und auch einfacher zu verstehen ist. Um 3 zu dem aktuellen Wert hinzuzufügen, schreibt man '\$a += 3'. Das bedeutet exakt: "Nimm' den Wert von \$a, addiere 3 hinzu und weise \$a den entstandenen Wert zu". Zusätzlich dazu, dass diese Schreibweise kürzer und klarer ist, resultiert sie auch in einer schnelleren Ausführung. Der Wert von '\$a += 3' ist, wie der Wert einer regulären Zuweisung, der zugewiesene Wert. Es ist zu beachten, dass dieser Wert NICHT 3, sondern dem kombinierten Wert von \$a plus 3 entspricht (Das ist der Wert, der \$a zugewiesen wird). Jeder Operator, der zwei Elemente verbindet, kann in dieser Schreibweise verwendet werden, z. B. '\$a -= 5' (vermindert den Wert von \$a um 5) oder '\$a *= 7' (multipliziert den Wert von \$a mit 7 und weist das Ergebnis \$a zu), usw.

Es gibt einen weiteren Ausdruck, der Ihnen vielleicht seltsam vorkommt, wenn Sie ihn bisher noch in keiner Programmiersprache kennengelernt haben, den dreifach konditionalen Operator:

```
$eins ? $zwei : $drei
```

Wenn der Wert des ersten Sub-Ausdruckes (hier: \$eins) wahr ist (d. h. nicht **NULL**), dann wird der Wert des zweiten Subausdrucks (hier: \$zwei) zurückgegeben und ist das Ergebnis des konditionalen Ausdrucks. Andernfalls (d. h. wenn der erste Ausdruck falsch ist), wird der Wert des dritten Subausdruckes (hier: \$drei) zurückgegeben.

Das folgende Beispiel sollte das Verständnis von Prä- und Post-Inkrement und von Ausdrücken im allgemeinen erleichtern:

```
function verdoppeln($i) {
    return $i*2;
}
$b = $a = 5;          /* Weise den Variablen $a und $b beiden den Wert 5 zu */
$c = $a++;           /* Post-Inkrement, der urspruengliche Wert von $a (also 5)
                    wird $c zugewiesen. */
```

```

$e = $d = ++$b;      /* Prae-Inkrement, der erhöhte Wert von $b (= 6) wird $d und
                    $e zugewiesen. */

/* An diesem Punkt sind $d und $e beide gleich 6 */

$f = verdoppeln($d++); /* Weise $f den doppelten Wert von $d vor
                    der Erhöhung um eins zu, 2*6 = 12 */
$g = double(++$e);   /* Weise $g den doppelten Wert von $e nach
                    der Erhöhung zu, 2*7 = 14 to $g */
$h = $g += 10;       /* Zuerst wie $g um 10 erhöht und hat schliesslich den Wert
                    24. Der Wert dieser Zuweisung (24) wird dann $h zugewiesen,
                    womit $h ebenfalls den Wert von 24 hat. */

```

Am Anfang dieses Kapitels hatten wir gesagt, wir würden die verschiedenen Arten von Anweisungen beschreiben und, wie versprochen, Ausdrücke können Anweisungen sein. Trotzdem ist nicht jeder Ausdruck eine Anweisung. In diesem Fall hat eine Anweisung die Form 'Ausdr' ';', d. h. ein Ausdruck gefolgt von einem Semikolon. In '\$b=\$a=5;' ist '\$a=5' ein gültiger Ausdruck, aber für sich allein keine Anweisung. '\$b=\$a=5;' ist jedoch eine gültige Anweisung.

Ein letzter Punkt, der noch zu erwähnen ist, ist der 'wahr'-Wert von Ausdrücken. In vielen Fällen, hauptsächlich in bedingten Anweisungen und Schleifen, ist man nicht am spezifischen Wert eines Ausdrucks interessiert, sondern kümmert sich nur darum, ob er WAHR oder FALSCH bedeutet (PHP hat keinen speziellen boolean-Datentyp). Der Wahrheitswert eines Ausdrucks in PHP wird ähnlich bestimmt wie in Perl. Jeder numerische Wert, der nicht **NULL** ist, bedeutet WAHR, **NULL** bedeutet FALSCH. Es ist zu beachten, dass negative Werte nicht **NULL** sind und deshalb als WAHR aufgefasst werden! Eine leere Zeichenkette und die Zeichenkette "0" sind FALSCH; alle anderen Zeichenketten sind WAHR. Nicht-skalare Datentypen (Arrays und Objekte) werden als FALSCH betrachtet, wenn sie keine Elemente enthalten, andernfalls geben sie WAHR zurück.

PHP stellt eine vollständige und mächtige Implementat von Ausdrücken bereit und, deren vollständige Dokumentation den Rahmen dieses Manuals sprengen würde. Die obigen Beispiele sollten Ihnen einen guten Eindruck darüber verschaffen, was Ausdrücke sind und wie man nützliche Ausdrücke konstruieren kann. Im Rest dieses Manuals werden wir *ausdr* schreiben, um ausdrücken, dass an dieser Stelle jeder gültige PHP-Ausdruck stehen kann.

Kapitel 15. Operatoren

Inhaltsverzeichnis

[Operator-Rangfolge](#)

[Arithmetische Operatoren](#)

[Zuweisungsoperatoren](#)

[Bit-Operatoren](#)

[Vergleichs-Operatoren](#)

[Fehler-Kontroll-Operatoren](#)

[Operatoren zur Programmausführung](#)
[Inkrement- bzw. Dekrementoperatoren](#)
[Logische Operatoren](#)
[Zeichenketten-Operatoren](#)
[Array Operatoren](#)
[Typ Operatoren](#)

Ein Operator ist etwas das Sie mit einem oder mehreren Werten füttern (oder Ausdrücken, um im Programmierjargon zu sprechen) und Sie erhalten als Ergebnis einen anderen Wert (damit wird diese Konstruktion selbst zu einem Ausdruck). Als Eselsbrücke können Sie sich Operatoren als Funktionen oder Konstrukte vorstellen, die Ihnen einen Wert zurück liefern (ähnlich print) und alles, was Ihnen keinen Wert zurück liefert (ähnlich echo) als irgend etwas Anderes.

Es gibt drei Arten von Operatoren. Als erstes gibt es den unären Operator, der nur mit einem Wert umgehen kann, zum Beispiel ! (der Verneinungsoperator) oder ++ (der Inkrementoperator). Die zweite Gruppe sind die sogenannten binären Operatoren; diese Gruppe enthält die meisten Operatoren, die PHP unterstützt. Eine Liste dieser Operatoren finden Sie weiter unten im Abschnitt [Operator-Rangfolge](#).

Die dritte Gruppe bildet der ternäre Operator : ?. Dieser sollte eher benutzt werden um abhängig von einem dritten Ausdruck eine Auswahl zwischen zwei Ausdrücken zu treffen, als zwischen zwei Sätzen oder Pfaden der Programmausführung zu wählen. Übrigens ist es eine sehr gute Idee ternäre Ausdrücke in Klammern zu setzen.

Operator-Rangfolge

Die Operator-Rangfolge legt fest, wie "eng" ein Operator zwei Ausdrücke miteinander verbindet. Zum Beispiel ist das Ergebnis des Ausdruckes $1 + 5 * 3$ 16 und nicht 18, da der Multiplikations-Operator ("*") in der Rangfolge höher steht als der Additions-Operator ("+"). Wenn nötig, können Sie Klammern setzen, um die Rangfolge der Operatoren zu beeinflussen. Zum Beispiel ergibt: $(1 + 5) * 3$ 18. Ist die Rangfolge der Operatoren gleich, wird links nach rechts Assoziativität benutzt.

Die folgende Tabelle zeigt die Rangfolge der Operatoren, oben steht der Operator mit dem höchsten Rang.

Tabelle 15-1. Operator-Rangfolge

Assoziativität	Operator
keine Richtung	new
rechts	[
rechts	! ~ ++ -- (int) (float) (string) (array) (object) @

Assoziativität	Operator
links	* / %
links	+ - .
links	<< >>
keine Richtung	< <= > >=
keine Richtung	== != === !==
links	&
links	^
links	
links	&&
links	
links	? :
rechts	= += -= *= /= .= %= &= = ^= <<= >>=
rechts	print
links	and
links	xor
links	or
links	,

Anmerkung: Obwohl ! einen höheren Rang gegenüber = hat, erlaubt es Ihnen PHP immer noch ähnliche Ausdrücke wie den folgenden zu schreiben: *if (!\$a =foo())*. In diesem Ausdruck wird die Ausgabe von *foo()* der Variablen *\$a* zugewiesen.

Arithmetische Operatoren

Erinnern Sie sich noch an die Grundrechenarten aus der Schule? Die arithmetischen Operatoren funktionieren genauso:

Tabelle 15-2. Arithmetische Operatoren

Beispiel	Name	Ergebnis
<code>\$a + \$b</code>	Addition	Summe von \$a und \$b.
<code>\$a - \$b</code>	Subtraktion	Differenz von \$a und \$b.
<code>\$a * \$b</code>	Multiplikation	Produkt von \$a und \$b.
<code>\$a / \$b</code>	Division	Quotient von \$a und \$b.
<code>\$a % \$b</code>	Modulus	Rest von \$a geteilt durch \$b.

Der Divisions-Operator ("/") gibt immer eine Fließkommazahl zurück, sogar wenn die zwei Operanden Ganzzahlen sind (oder Zeichenketten, die nach Ganzzahlen

umgewandelt wurden).

Siehe auch im Handbuch das Kapitel über [Mathematische Funktionen](#).

Zuweisungsoperatoren

Der einfachste Zuweisungsoperator ist "=". Wahrscheinlich kommt man als erstes auf die Idee, ihn mit "ist gleich" zu bezeichnen. Das ist falsch. In Wirklichkeit bedeutet er, dass dem linken Operanden der Wert des Ausdrucks auf der rechten Seite zugewiesen wird (man müsste ihn also mit "wird gesetzt auf den Wert von" übersetzen).

Der Wert eines Zuweisungs-Ausdruckes ist der zugewiesene Wert. D.h. der Wert des Ausdruckes "\$a = 3" ist 3. Das erlaubt es, einige raffinierte Dinge anzustellen:

```
<?php
$a = ($b = 4) + 5; // $a ist nun gleich 9 und $b wurde auf den Wert 4
gesetzt.
?>
```

Zusätzlich zu dem oben vorgestellten Zuweisungsoperator "=" gibt es "kombinierte Operatoren" für alle [binären](#), [arithmetischen](#) und String-Operatoren, die es erlauben, den Wert einer Variablen in einem Ausdruck zu benutzen, und dieser anschließend das Ergebnis des Ausdrucks als neuen Wert zuzuweisen. Zum Beispiel:

```
<?php
$a = 3;
$a += 5; // setzt $a auf den Wert 8, als ob wir geschrieben haetten:
$a = $a + 5;
$b = "Hallo ";
$b .= "Du!"; // setzt $b auf den Wert "Hallo Du!", aequivalent zu
              // $b = $b . "Du!";
?>
```

Man beachte, dass die Zuweisung nur den Wert der Ursprungsvariable der neuen Variable zuweist (Zuweisung als Wert, sie "kopiert"), weshalb sich Änderungen an der einen Variablen nicht auf die andere auswirken werden. Das kann wichtig sein, wenn man ein großes Array o. ä. in einer Schleife kopieren muss. PHP 4 unterstützt 'assignment by reference' (Zuweisung als Verweis), mit Hilfe der Schreibweise \$var = &\$othervar;, das funktioniert jedoch nicht in PHP 3. 'Assignment by reference' bedeutet, dass beide Variablen nach der Zuweisung die selben Daten repräsentieren und nichts kopiert wird. Um mehr über Referenzen zu lernen, lesen Sie bitte den Abschnitt [Referenzen erklärt](#).

Bit-Operatoren

Bit-Operatoren erlauben es, in einem Integer bestimmte Bits "ein- oder auszuschalten" (auf 0 oder 1 zu setzen). Wenn beide, der links- und rechtsseitige Parameter, Zeichenketten sind, arbeiten die Bit-Operatoren mit ASCII-Werten der einzelnen Zeichen.

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

echo "12" ^ "9"; // Ausgabe: das Backspace-Zeichen (ascii 8)
                // ('1' (ascii 49)) ^ ('9' (ascii 57)) = #8

echo "hallo" ^ "hello"; // Gibt die ASCII-Werte #0 #4 #0 #0 #0
                        // 'a' ^ 'e' = #4 aus
?>
```

Tabelle 15-3. Bit-Operatoren

Beispiel	Name	Ergebnis
<code>\$a & \$b</code>	Und	Bits, die in \$a und \$b gesetzt sind werden gesetzt.
<code>\$a \$b</code>	Oder	Bits, die in \$a oder \$b gesetzt sind werden gesetzt.
<code>\$a ^ \$b</code>	Entweder oder (Xor)	Bits, die entweder in \$a oder \$b gesetzt sind, werden gesetzt aber nicht in beiden.
<code>~ \$a</code>	Nicht	Die Bits, die in \$a nicht gesetzt sind, werden gesetzt und umgekehrt.
<code>\$a << \$b</code>	Nach links verschieben	Verschiebung der Bits von \$a um \$b Stellen nach links (jede Stelle entspricht einer Multiplikation mit zwei).
<code>\$a >> \$b</code>	Nach rechts verschieben	Verschiebt die Bits von \$a um \$b Stellen nach rechts (jede Stelle entspricht einer Division durch zwei).

Vergleichs-Operatoren

Vergleichs-Operatoren erlauben es - wie der Name schon sagt - zwei Werte zu vergleichen. Wenn Sie an Beispielen verschiedener auf Typen bezogener Vergleiche interessiert sind, können Sie sich [PHP type comparison tables](#) anschauen.

Tabelle 15-4. Vergleichsoperatoren

Beispiel	Name	Ergebnis
<code>\$a == \$b</code>	Gleich	Gibt TRUE zurück, wenn \$a gleich \$b ist.
<code>\$a === \$b</code>	Identisch	Gibt TRUE zurück wenn \$a gleich \$b ist und beide vom gleichen Typ sind(nur PHP 4).
<code>\$a != \$b</code>	Ungleich	Gibt TRUE zurück, wenn \$a nicht gleich \$b ist.
<code>\$a <> \$b</code>	Ungleich	Gibt TRUE zurück, wenn \$a nicht gleich \$b ist.
<code>\$a !== \$b</code>	Nicht identisch	Gibt TRUE zurück, wenn \$a nicht gleich \$b ist, oder wenn beide nicht vom gleichen Typ sind (nur PHP 4).
<code>\$a < \$b</code>	Kleiner Als	Gibt TRUE zurück, wenn \$a kleiner als \$b ist.
<code>\$a > \$b</code>	Größer Als	Gibt TRUE zurück, wenn \$a größer als \$b ist.
<code>\$a <= \$b</code>	Kleiner Gleich	Gibt TRUE zurück, wenn \$a kleiner oder gleich \$b ist.
<code>\$a >= \$b</code>	Größer Gleich	Gibt TRUE zurück, wenn \$a größer oder gleich \$b ist.

Ein weiterer Vergleichs-Operator ist der "?:"- oder Trinitäts-Operator.

```
<?php
// Beispielanwendung für den Trinitäts-Operator
$action = (empty($_POST['action'])) ? 'standard' : $_POST['action'];

// Obiges ist mit dieser if/else-Anweisung identisch
if (empty($_POST['action'])) {
    $action = 'standard';
} else {
    $action = $_POST['action'];
}
?>
```

Der Ausdruck $(ausdr1) ? (ausdr2) : (ausdr3)$ gibt *ausdr2* zurück, wenn *ausdr1* **TRUE** zurückgibt und *ausdr3*, wenn *ausdr1* **FALSE** zurückgibt.

Siehe auch [strcasecmp\(\)](#), [strcmp\(\)](#), [Array Operatoren](#) und den Abschnitt über [Typen](#).

Fehler-Kontroll-Operatoren

PHP unterstützt einen Operator zur Fehlerkontrolle: Das @-Symbol. Stellt man das @ in PHP vor einen Ausdruck werden alle Fehlermeldungen, die von diesem Ausdruck erzeugt werden könnten, ignoriert.

Ist das [track_errors](#)-Feature aktiviert, werden alle Fehlermeldungen, die von diesem Ausdruck erzeugt werden, in der Variablen `$php_errormsg` gespeichert. Da diese Variable mit jedem neuen Auftreten eines Fehlers überschrieben wird, sollte man sie möglichst bald nach Verwendung des Ausdrucks überprüfen, wenn

man mit ihr arbeiten will.

```
<?php
/* Beabsichtigter Dateifehler */
$my_file = @file ('nicht_vorhandene_Datei') or
    die ("Datei konnte nicht geöffnet werden: Fehler
war: '$php_errormsg'");

// Das funktioniert bei jedem Ausdruck, nicht nur bei Funktionen:
$value = @$cache[$key];
// erzeugt keine Notice, falls der Index $key nicht vorhanden ist.

?>
```

Anmerkung: Der @-Operator funktioniert nur bei [Ausdrücken](#). Eine einfache Daumenregel: wenn Sie den Wert von etwas bestimmen können, dann können Sie den @-Operator davor schreiben. Zum Beispiel können Sie ihn vor Variablen, Funktionsaufrufe und vor [include\(\)](#) setzen, vor Konstanten und so weiter. Nicht verwenden können Sie diesen Operator vor Funktions- oder Klassendefinitionen oder vor Kontrollstrukturen wie zum Beispiel *if* und *foreach* und so weiter.

Siehe auch [error_reporting\(\)](#) und den Abschnitt über [Error Handling and Logging Functions](#).

Anmerkung: Der @ Fehler-Kontroll-Operator verhindert jedoch keine Meldungen, welche aus Fehlern beim Parsen resultieren.

Warnung

Zum gegenwärtigen Zeitpunkt deaktiviert der "@" Fehler-Kontrolloperator die Fehlermeldungen selbst bei kritischen Fehlern, die die Ausführung eines Skripts beenden. Unter anderem bedeutet das, wenn Sie "@" einer bestimmten Funktion voranstellen, diese aber nicht zur Verfügung steht oder falsch geschrieben wurde, Ihr PHP-Skript einfach beendet wird, ohne Hinweis auf die Ursache.

Operatoren zur Programmausführung

PHP unterstützt einen Operator zur Ausführung externer Programme: Die sog. Backticks (` `). Achtung: Die Backticks sind keine einfachen Anführungszeichen! PHP versucht, den Text zwischen den Backticks als Kommandozeilen-Befehl auszuführen. Die Ausgabe des aufgerufenen Programms wird zurückgegeben (d.h. wird nicht einfach ausgegeben, sondern kann einer Variablen zugewiesen

werden). Die Verwendung des Backtick-Operators ist mit [shell_exec\(\)](#) identisch.

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

Anmerkung: Der Backtick-Operator steht nicht zur Verfügung, wenn [Safe Mode](#) aktiviert ist oder die Funktion [shell_exec\(\)](#) deaktiviert wurde.

Siehe auch den Abschnitt über [Funktionen zur Programmausführung](#), [popen\(\)](#), [proc_open\(\)](#) und [Using PHP from the commandline](#).

Inkrement- bzw. Dekrementoperatoren

PHP unterstützt Prä- und Post-Inkrement- und Dekrementoperatoren im Stil der Programmiersprache C.

Tabelle 15-5. Inkrement- und Dekrementoperatoren

Beispiel	Name	Auswirkung
++\$a	Prä-Inkrement	Erhöht den Wert von \$a um eins (inkrementiert \$a) und gibt anschließend den neuen Wert von \$a zurück.
\$a++	Post-Inkrement	Gibt zuerst den aktuellen Wert von \$a zurück und erhöht dann den Wert von \$a um eins.
--\$a	Prä-Dekrement	Vermindert den Wert von \$a um eins (dekrementiert \$a) und gibt anschließend den neuen Wert von \$a zurück.
\$a--	Post-Dekrement	Gibt zuerst den aktuellen Wert von \$a zurück und erniedrigt dann den Wert von \$a um eins.

Ein einfaches Beispiel-Skript:

```
<?php
echo "<h3>Post-Inkrement</h3>";
$a = 5;
echo "Sollte 5 sein: " . $a++ . "<br />\n";
echo "Sollte 6 sein: " . $a . "<br />\n";

echo "<h3>Pre-Inkrement</h3>";
$a = 5;
echo "Sollte 6 sein: " . ++$a . "<br />\n";
echo "Sollte 6 sein: " . $a . "<br />\n";

echo "<h3>Post-Dekrement</h3>";
```

```

$a = 5;
echo "Sollte 5 sein: " . $a-- . "<br />\n";
echo "Sollte 4 sein: " . $a . "<br />\n";

echo "<h3>Pre-Dekrement</h3>";
$a = 5;
echo "Sollte 4 sein: " . --$a . "<br />\n";
echo "Sollte 4 sein: " . $a . "<br />\n";
?>

```

PHP folgt bei der Behandlung arithmetischer Operationen an Zeichenvariablen der Perl-Konvention und nicht der von C. Zum Beispiel wird in Perl aus 'Z'+1 'AA', während aus 'Z'+1 in C '[' wird (ord('Z') == 90, ord '[') == 91). Beachten Sie, dass Zeichenvariablen zwar inkrementiert aber nicht dekrementiert werden können.

Beispiel 15-1. Arithmetrische Operationen an Zeichenvariablen

```

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

/*
    Erzeugt in etwa folgende Ausgabe:

X
Y
Z
AA
AB
AC

*/
?>

```

Logische Operatoren

Tabelle 15-6. Logische Operatoren

Beispiel	Name	Ergebnis
\$a and \$b	Und	TRUE wenn sowohl \$a als auch \$b TRUE ist.
\$a or \$b	Oder	TRUE wenn \$a oder \$b TRUE ist.
\$a xor \$b	Entweder Oder	TRUE wenn entweder \$a oder \$b TRUE ist, aber nicht beide.

Beispiel	Name	Ergebnis
! \$a	Nicht	TRUE wenn \$a nicht TRUE ist.
\$a && \$b	Und	TRUE wenn sowohl \$a als auch \$b TRUE ist.
\$a \$b	Oder	TRUE wenn \$a oder \$b TRUE ist.

Der Grund dafür, dass es je zwei unterschiedliche Operatoren für die "Und"- und die "Oder"-Verknüpfung gibt ist der, dass die beiden Operatoren jeweils Rangfolgen haben. (siehe auch [Operator-Rangfolge](#).)

Zeichenketten-Operatoren

Es gibt in PHP zwei Operatoren für **string** (Zeichenkette). Der erste ist der Vereinigungs-Operator ('.'), dessen Rückgabewert eine zusammengesetzte Zeichenkette aus dem rechten und dem linken Argument ist. Der zweite ist der Vereinigungs-Zuweisungsoperator ('.='), der das Argument auf der rechten Seite an das Argument der linken Seite anhängt. Siehe [Zuweisungs-Operatoren](#) für weitere Informationen.

```
<?php
$a = "Hallo ";
$b = $a . "Welt!"; // $b enthält jetzt den Text "Hallo Welt!"

$a = "Hallo ";
$a .= "Welt!";    // $a enthält jetzt den Text "Hallo Welt!"
?>
```

Siehe auch die Abschnitte über [Strings / Zeichenketten](#) und [String-Funktionen](#).

Array Operatoren

Tabelle 15-7. Array Operatoren

Beispiel	Name	Ergebnis
\$a + \$b	Vereinigung	Vereinigung von \$a und \$b.
\$a == \$b	Gleichwertigkeit	TRUE wenn \$a und \$b die gleichen Elemente enthalten.
\$a === \$b	Identität	TRUE wenn \$a und \$b die gleichen Elemente in der gleichen Reihenfolge enthalten.
\$a != \$b	Ungleichheit	TRUE wenn \$a nicht gleich \$b ist.
\$a <> \$b	Ungleichheit	TRUE wenn \$a nicht gleich \$b ist.
\$a !== \$b	nicht identisch	TRUE wenn \$a nicht identisch zu \$b ist.

Der + Operator hängt das rechtsstehende Array an das linksstehende Array an, wobei doppelte Schlüssel NICHT überschrieben werden.

```

<?php
$a = array("a" => "Apfel", "b" => "Banane");
$b = array("a" =>"pear", "b" => "Erdbeere", "c" => "Kirsche");

$c = $a + $b; // Vereinigung von $a mit $b;
echo "Vereinigung von \$a mit \$b: \n";
var_dump($c);

$c = $b + $a; // Vereinigung von $b mit $a;
echo "Vereinigung von \$b mit \$a: \n";
var_dump($c);
?>

```

Dieses Skript gibt folgendes aus:

```

Vereinigung von $a mit $b:
array(3) {
  ["a"]=>
  string(5) "Apfel"
  ["b"]=>
  string(6) "Banane"
  ["c"]=>
  string(7) "Kirsche"
}
Vereinigung von $b mit $a:
array(3) {
  ["a"]=>
  string(4) "pear"
  ["b"]=>
  string(8) "Erdbeere"
  ["c"]=>
  string(7) "Kirsche"
}

```

Beim Vergleich werden Arrayelemente als gleich angesehen, wenn diese dieselben Schlüssel und Werte beinhalten.

Beispiel 15-2. Array Vergleiche

```

<?php
$a = array("a" => "Apfel", "b" => "Banane");
$b = array(1 => "Banane", "0" => "Apfel");

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

```

Siehe auch die Abschnitte über [Arrays](#) und [Array Funktionen](#).

Typ Operatoren

In PHP gibt es einen einzigen Typ Operator: *instanceof*. *instanceof* wird dazu verwendet um festzustellen, ob ein gegebenes Objekt ein Objekt ist, das zu einer bestimmten [Klasse](#) gehört.

instanceof wurde in PHP 5 eingeführt. Vorher wurde [is_a\(\)](#) benutzt, aber [is_a\(\)](#) ist veraltet und *instanceof* sollte stattdessen benutzt werden.

```
<?php
class A { }
class B { }

$ding = new A;

if ($ding instanceof A) {
    echo 'A';
}
if ($ding instanceof B) {
    echo 'B';
}
?>
```

Da *\$ding* ein **object** vom Typ A und nicht von B ist, wird nur der Programmblock ausgeführt, der abhängig von Typ A ist:

```
A
```

See auch [get_class\(\)](#) und [is_a\(\)](#).

Kapitel 16. Kontroll-Strukturen

Inhaltsverzeichnis

[if](#)

[else](#)

[elseif](#)

[Alternative Syntax für Kontroll-Strukturen](#)

[while](#)

[do..while](#)

[for](#)

[foreach](#)

[break](#)

[continue](#)

[switch](#)

[declare](#)

[return](#)

[require\(\)](#)

[include\(\)](#)

[require_once\(\)](#) [include_once\(\)](#)

Jedes PHP-Skript besteht aus einer Reihe von Anweisungen. Eine Anweisung kann eine Zuweisung, ein Funktionsaufruf, eine Schleife, eine bedingte Anweisung oder ein Befehl sein, der nichts macht (eine leere Anweisung). Jede Anweisung endet gewöhnlich mit einem Semikolon. Darüber hinaus können Anweisungen zu einer Anweisungsgruppe zusammengefasst werden, welche durch geschweifte Klammern begrenzt wird. Eine Anweisungsgruppe selbst ist auch wieder eine Anweisung. Die unterschiedlichen Arten von Anweisungen werden in diesem Kapitel erläutert.

if

Das *if*-Konstrukt ist eine der wichtigsten Möglichkeiten vieler Programmiersprachen, PHP eingeschlossen. Es erlaubt die bedingte Ausführung von Programmteilen. PHP kennt eine *if*-Struktur, die ähnlich wie in der Programmiersprache C implementiert ist:

```
<?php
  if (ausdr)
    Anweisung
?>
```

Wie im Abschnitt über [Ausdrücke](#) beschrieben, wird *ausdr* auf seinen booleschen Wertinhalt ausgewertet. Wenn *ausdr* als **TRUE** ausgewertet wird, führt PHP die *Anweisung* aus. Falls die Auswertung **FALSE** ergibt, wird die *Anweisung* übergangen. Mehr Informationen drüber welche Werte als **FALSE** ausgewertet werden finden Sie im Abschnitt '[Umwandlung nach boolean](#)'.

Das folgende Beispiel wird a ist größer als b anzeigen, wenn *\$a* größer als *\$b* ist:

```
<?php
if ($a > $b)
  echo "a ist größer als b";
?>
```

Oft werden Sie die bedingte Ausführung von mehr als einer Anweisung wollen. Selbstverständlich ist es nicht erforderlich, jede Anweisung mit einer *if*-Bedingung zu versehen. Statt dessen können Sie mehrere Anweisungen in Gruppen zusammenfassen. Der folgende Programm-Code wird zum Beispiel a ist größer als b anzeigen, wenn *\$a* größer als *\$b* ist und danach wird der Wert von *\$a* in *\$b* gespeichert:

```
<?php
if ($a > $b) {
  print "a ist größer als b";
```

```
$b = $a;
}
?>
```

if-Anweisungen können beliebig oft innerhalb anderer *if*-Anweisungen definiert werden. Das ermöglicht ihnen völlige Flexibilität bei der bedingten Ausführung verschiedenster Programmteile.

else

Häufig möchten Sie eine Anweisung auszuführen, wenn eine bestimmte Bedingung erfüllt ist und eine andere Anweisung, falls sie nicht erfüllt ist. Dafür gibt es *else*. *else* erweitert eine *if*-Anweisung um die Ausführung von Anweisungen, sobald der Ausdruck der *if*-Anweisung als **FALSE** ausgewertet wird. Der folgende Code wird z.B. *a ist größer als b* ausgeben, wenn *\$a* größer als *\$b* ist, anderenfalls *a ist NICHT größer als b*:

```
<?php
if ($a > $b) {
    print "a ist größer als b";
} else {
    print "a ist NICHT größer als b";
}
?>
```

Die *else*-Anweisung wird nur ausgeführt, wenn der *if*-Ausdruck als **FALSE** ausgewertet wurde und wenn bei vorhandenen *elseif*-Ausdrücken diese ebenfalls **FALSE** sind (siehe [elseif](#)).

elseif

Wie der Name schon sagt ist *elseif* eine Verbindung von *if* und *else*. Wie *else* erweitert sie eine *if*-Anweisung um die Ausführung anderer Anweisungen, sobald die normale *if*-Bedingung als **FALSE** ausgewertet wird. Anders als bei *else* wird die Ausführung dieser alternativen Anweisungen nur durchgeführt, wenn die bei *elseif* angegebene alternative Bedingung als **TRUE** ausgewertet wird. Der folgende Code wird z.B. *a ist größer als b*, *a ist gleich b* oder *a ist kleiner als b* ausgeben:

```
<?php
if ($a > $b) {
    echo "a ist größer als b";
} elseif ($a == $b) {
    echo "a ist gleich b";
} else {
```

```
    echo "a ist kleiner als b";
}
?>
```

Es kann mehrere *elseif*-Anweisungen innerhalb einer *if*-Anweisung geben. Die erste *elseif*-Bedingung (falls vorhanden), die **TRUE** ist, wird ausgeführt. In PHP kann man auch 'else if' schreiben (zwei Wörter). Das Verhalten ist identisch zu 'elseif' (ein Wort). Die Bedeutung der Syntax ist leicht unterschiedlich (falls Sie mit C vertraut sind, das ist das gleiche Verhalten) aber der Grundtenor ist der, dass beide Schreibweisen, bezogen auf das Ergebnis, sich exakt gleich verhalten.

Die *elseif*-Anweisung wird nur ausgeführt, wenn die vorausgehende *if*-Bedingung sowie jede vorherige *elseif*-Bedingung als **FALSE** ausgewertet wird und die aktuelle *elseif*-Bedingung **TRUE** ist.

Alternative Syntax für Kontroll-Strukturen

PHP bietet eine alternative Syntax für einige seiner Kontroll-Strukturen, als da sind *if*, *while*, *for*, *foreach* und *switch*. Die öffnende Klammer muss immer durch einen Doppelpunkt ":" und die schließende Klammer entsprechend durch *endif*;, *endwhile*;, *endfor*;, *endforeach*;; oder *endswitch*;; ersetzt werden.

```
<?php if ($a == 5): ?>
A ist gleich 5
<?php endif; ?>
```

Im obigen Beispiel ist der HTML-Bereich "A ist gleich 5" in eine *if*-Anweisung mit alternativer Syntax eingebettet. Der HTML-Bereich wird nur ausgegeben, wenn *\$a* gleich 5 ist.

Die alternative Syntax kann auch auf *else* und *elseif* angewendet werden. Es folgt eine *if*-Struktur mit *elseif* und *else* im alternativen Format:

```
<?php
if ($a == 5):
    echo "a ist gleich 5";
    echo "...";
elseif ($a == 6):
    echo "a ist gleich 6";
    echo "!!!";
else:
    echo "a ist weder 5 noch 6";
endif;
?>
```

Siehe auch [while](#), [for](#) und [if](#) für weitere Beispiele.

while

Die *while*-Schleifen sind die einfachste Form von Schleifen in PHP. Sie funktionieren genau wie in C. Die Grundform einer *while*-Anweisung lautet:

```
while (ausdr) Anweisung
```

Die Bedeutung einer *while*-Anweisung ist einfach. Sie weist PHP an, einen in ihr eingebetteten Befehl so lange zu wiederholen, wie die *while*-Bedingung als **TRUE** ausgewertet wird. Der Wert der Bedingung wird immer am Anfang der Schleife geprüft. Wird der Wert während der Ausführung der Anweisungen innerhalb der *while*-Schleife verändert, endet die Ausführung dieses Anweisungsblocks erst mit dem Ende der Iteration (Jeder Schleifendurchlauf ist eine Iteration). Falls die *while*-Bedingung bereits beim ersten Mal **FALSE** ist, werden die Anweisungen der *while*-Schleife nicht ein einziges Mal durchlaufen.

Wie bei der *if*-Anweisung können Sie mehrere Anweisungen innerhalb der gleichen *while*-Schleife angeben, indem Sie dieses mit geschweiften Klammern umschließen oder die alternative Syntax verwenden:

```
while (ausdr): Anweisung ... endwhile;
```

Die folgenden Beispiele sind identisch; beide geben Zahlen von 1 bis 10 aus:

```
/* Beispiel 1 */

$i = 1;
while ($i <= 10) {
    echo $i++; /* es wird erst $i ausgegeben, bevor der Wert erhöht wird
               (Post-Inkrement) */
}

/* Beispiel 2 */

$i = 1;
while ($i <= 10):
    echo $i;
    $i++;
endwhile;
```

do..while

do..while-Schleifen sind den *while*-Schleifen sehr ähnlich, außer dass der Wahrheitsgehalt des Ausdrucks erst am Ende jedes Durchlaufs geprüft wird, statt am Anfang. Der Hauptunterschied zu gewöhnlichen *while*-Schleifen ist der, dass die Schleife bei *do..while* in jeden Fall einmal durchlaufen wird (die Bedingung wird erst am Ende eines Durchlaufs geprüft). Bei *while*-Schleifen hingegen kann es durchaus passieren, dass die Schleife nie durchlaufen wird

(die Bedingung wird immer am Anfang eines Durchlaufs überprüft. Wird diese Bedingung von Anfang an als **FALSE** ausgewertet endet die Ausführung der Schleife sofort).

Es gibt nur eine Syntax für *do..while*-Schleifen:

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

Die obige Schleife wird genau einmal durchlaufen, da nach der ersten Wiederholung die Erfüllung der Bedingung geprüft wird. Diese Bedingung ist aber nicht erfüllt (i ist nicht größer als 0), wird als **FALSE** ausgewertet, und die Schleifenausführung beendet.

Erfahrene C-Anwender kennen auch die Möglichkeit, Programm-Blöcke mit *do..while* (0) einzuschliessen und dann die [*break*](#) Anweisung zu benutzen. Der folgende Programm-Ausschnitt zeigt diese Möglichkeit:

```
do {
    if ($i < 5) {
        echo "i ist nicht groß genug";
        break;
    }
    $i *= $factor;
    if ($i < $minimum_limit) {
        break;
    }
    echo "i ist ok";

    /* mach was mit i */
} while (0);
```

Es ist nicht weiter tragisch, wenn Sie dieses Beispiel nicht oder nur zum Teil verstehen. Sie können auch ohne dieses 'Feature' effektive PHP-Programme und Skripte schreiben.

for

Die *for*-Schleifen sind die komplexesten Schleifen in PHP. Sie funktionieren wie ihr Gegenstück in C. Die Syntax einer *for*-Schleife sieht wie folgt aus:

```
for (ausdr1; ausdr2; ausdr3) Anweisung
```

Der erste Ausdruck (*ausdr1*) wird beim Schleifenbeginn (ohne jegliche Vorbedingung) geprüft bzw. ausgeführt.

Zu Beginn jedes Durchlaufs wird nun *ausdr2* geprüft. Wenn dieser **TRUE** ist, fährt die Schleife mit der Ausführung der nachfolgenden Anweisung(en) fort. Ist das Ergebnis **FALSE**, wird die Schleife beendet.

Am Ende jedes Durchlaufs wird *ausdr3* geprüft (ausgeführt).

Jeder Ausdruck kann leer sein. Ist *ausdr2* leer, wird die Schleife endlos oft durchlaufen (PHP wertet diesen, wie in C, implizit als **TRUE**). Das ist gar nicht so sinnlos, wie Sie vielleicht zunächst glauben, weil man häufig eine Schleife erst durch eine bedingte *break*-Anweisung statt durch eine unwahr werdende *for*-Bedingung beenden möchte.

Beachten Sie die folgenden Beispiele. Alle geben Zahlen von 1 bis 10 aus:

```
<?php
/* Beispiel 1 */

for ($i = 1; $i <= 10; $i++) {
    echo $i;
}

/* Beispiel 2 */

for ($i = 1; ; $i++) {
    if ($i > 10) {
        break;
    }
    echo $i;
}

/* Beispiel 3 */

$i = 1;
for ( ; ; ) {
    if ($i > 10) {
        break;
    }
    echo $i;
    $i++;
}

/* Beispiel 4 */

for ($i = 1; $i <= 10; echo $i, $i++);
?>
```

Selbstverständlich sieht das erste (oder vielleicht das vierte) Beispiel am besten aus, aber Sie werden noch feststellen, dass es oftmals ganz nützlich sein kann, leere Parameter in *for*-Schleifen zu verwenden.

PHP unterstützt auch bei *for*-Schleifen die alternative "Doppelpunkt-Syntax".

```
for (ausdr1; ausdr2; ausdr3): Anweisung; ...; endfor;
```

Andere Sprachen haben für das Durchlaufen eines Hashs oder Arrays eine *foreach*-Anweisung. PHP 3 hat dies nicht; im Gegensatz zu PHP 4 (vgl. [foreach](#)). In PHP 3 können Sie für diesen Zweck [while](#) mit der [list\(\)](#)- und [each\(\)](#)-Funktion kombinieren. Beispiele finden Sie in der Dokumentation zu diesen Funktionen.

foreach

PHP 4 (nicht PHP 3) enthält ein *foreach* Konstrukt, genau wie Perl und einige andere Sprachen. Diese ermöglicht es, auf einfache Weise ein Array zu durchlaufen. *foreach* funktioniert nur in Verbindung mit Arrays. Wenn Sie versuchen *foreach* mit Variablen eines anderen Datentyps oder nicht initialisierten Variablen zu benutzen, gibt PHP einen Fehler aus. Es gibt zwei Syntaxformen; die zweite ist eine unbedeutende, aber sinnvolle Erweiterung der ersten Syntax:

```
foreach (array_expression as $value) Anweisung  
foreach (array_expression as $key => $value) Anweisung
```

Die erste Form durchläuft das *array_expression*-Array. Bei jedem Durchgang wird der Wert des aktuellen Elements *\$value* zugewiesen und der interne Array-Zeiger um eins erhöht. Dadurch wird beim nächsten Durchgang automatisch das nächste Element ausgewertet.

Die zweite Form arbeitet genauso, außer dass bei jedem Durchlauf auch der aktuelle Schlüssel der Variablen *\$key* zugewiesen wird.

Anmerkung: Sobald *foreach* zum ersten Mal ausgeführt wird, wird der interne Arrayzeiger automatisch auf das erste Element des Arrays gesetzt. Das bedeutet, dass Sie vor einem Durchlauf von *foreach* [reset\(\)](#) nicht aufrufen müssen.

Anmerkung: Beachten Sie auch, dass *foreach* mit einer Kopie des angegebenen Arrays arbeitet, nicht mit dem Array selbst. Deshalb wird auch der Arrayzeiger nicht wie bei dem [each\(\)](#)-Konstrukt verändert und Veränderungen an ausgegebenen Arrayelementen haben keine Auswirkung auf das originale Array. Trotzdem *wird* der interne Arrayzeiger des originalen Arrays bei der Verarbeitung bewegt. Angenommen, die *foreach*-Schleife ist komplett abgearbeitet, wird der interne Arrayzeiger (des originalen Arrays) auf das letzte Element zeigen.

Anmerkung: Bei *foreach* ist es nicht möglich Fehlermeldungen durch den Gebrauch von '@' zu unterdrücken ist

Beachten Sie, dass die folgenden Beispiele in ihrer Funktionalität identisch sind:

```

<?php
$arr = array("eins", "zwei", "drei");
reset ($arr);
while (list(, $value) = each ($arr)) {
    echo "Wert:  $value<br />\n";
}

foreach ($arr as $value) {
    echo "Wert:  $value<br />\n";
}
?>

```

Auch hier funktioniert alles gleich:

```

<?php
$arr = array("eins", "zwei", "drei");
reset ($arr);
while (list($key, $value) = each ($arr)) {
    echo "Schlüssel: $key; Wert: $value<br />\n";
}

foreach ($arr as $key => $value) {
    echo "Schlüssel: $key; Wert: $value<br />\n";
}
?>

```

Noch einige Beispiele, die die Anwendung verdeutlichen:

```

<?php
/* foreach Beispiel 1: Nur der Wert */

$a = array(1, 2, 3, 17);

foreach ($a as $v) {
    echo "Aktueller Wert von \$a: $v.\n";
}

/* foreach Beispiel 2:
Wert (mit Ausgabe des Arrayschlüssels zur Veranschaulichung) */

$a = array(1, 2, 3, 17);

$i = 0; /* nur zu Veranschaulichung */

foreach($a as $v) {
    echo "\$a[$i] => $v.\n";
    $i++;
}

```

```

/* foreach Beispiel 3: Schlüssel und Wert */

$a = array(
    "eins" => 1,
    "zwei" => 2,
    "drei" => 3,
    "siebzehn" => 17
);

foreach($a as $k => $v) {
    echo "\$a[$k] => $v.\n";
}

/* foreach Beispiel 4: multidimensionale Arrays */

$a[0][0] = "a";
$a[0][1] = "b";
$a[1][0] = "y";
$a[1][1] = "z";

foreach($a as $v1) {
    foreach ($v1 as $v2) {
        echo "$v2\n";
    }
}

/* foreach Beispiel 5: dynamische Arrays */

foreach (array(1, 2, 3, 4, 5) as $v) {
    echo "$v\n";
}
?>

```

break

break bricht die Ausführung der aktuellen *for*, *foreach while*, *do..while* Schleife oder einer *switch* Anweisungssequenz ab.

Einem *break* kann optional ein numerisches Argument angehängt werden, das die Anzahl der abubrechenden Befehlssequenzen enthält.

```

<?php
$arr = array('eins', 'zwei', 'drei', 'vier', 'stop', 'fünf');
while (list ( , $val) = each ($arr)) {
    if ($val == 'stop') {
        break; /* Sie könnten hier auch 'break 1;' schreiben. */
    }
}

```

```

    echo "$val<br />\n";

/* Benutzung des optionalen Argumentes. */

$i = 0;
while (++$i) {
    switch ($i) {
        case 5:
            echo "Bei 5<br />\n";
            break 1; /* Beendet nur switch. */
        case 10:
            echo "Bei 10; aussteigen<br />\n";
            break 2; /* Beendet switch und while. */
        default:
            break;
    }
}
?>

```

continue

continue wird innerhalb von Schleifen verwendet. Die Schleife wird an der aktuellen Stelle abgebrochen und es wird der nächste Durchlauf begonnen.

Anmerkung: Beachten Sie, dass in PHP die [switch](#) Anweisung als Schleifenstruktur zum Zweck von *continue* angesehen wird.

Bei *continue* können Sie ein optionales numerisches Argument mitgeben, das angibt, wie viele Ebenen von enthaltenen Schleifen übersprungen werden sollen.

```

<?php
while (list ($key, $value) = each ($arr)) {
    if (!($key % 2)) { // überspringe ungerade Werte
        continue;
    }
    tue_was_mit_ungerade ($value);
}

$i = 0;
while ($i++ < 5) {
    echo "Außen<br />\n";
    while (1) {
        echo "&nbsp;&nbsp;&nbsp;Mitte<br />\n";
        while (1) {
            echo "&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;Innen<br />\n";
            continue 3;
        }
    }
}

```

```

        echo "Das wird nie ausgegeben.<br />\n";
    }
    echo "Das hier auch nicht.<br />\n";
}
?>

```

Lassen Sie das Semikolon nach *continue* weg, kann dies zu verwirrenden Ergebnissen führen. Es folgt ein Beispiel, wie Sie es nicht machen sollten.

```

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

```

Sie könnten erwarten, dass das Ergebnis wie folgt aussieht,

```

0
1
3
4

```

aber tatsächlich sieht die Ausgabe so aus,

```

2

```

weil der Rückgabewert von [print\(\)](#) *int(1)* ist und das wird als das oben angesprochene optimale Argument gewertet.

switch

Die *switch*-Anweisung ist gleichbedeutend einer Reihe von *if*-Anweisungen mit dem gleichen Parameter. Häufig wollen Sie ein und dieselbe Variable (bzw. den selben Ausdruck) mit verschiedensten Werten vergleichen und in Abhängigkeit vom Auswertungsergebnis verschiedene Programmteile ausführen. Genau das ermöglicht die *switch*-Anweisung.

Anmerkung: Beachten Sie bitte, dass im Unterschied zu anderen Programmiersprachen die [continue](#) Anweisung auch bei *switch* ihre Gültigkeit hat und ähnlich wie *break* funktioniert. Falls Sie *switch* innerhalb einer Schleife verwenden und mit dem nächsten Durchlauf der äußeren Schleife beginnen möchten, verwenden Sie *continue 2*.

Das folgende Beispiele zeigt Ihnen zwei verschiedene Möglichkeiten auf, das Gleiche zu erreichen. Einmal werden *if* und *elseif* Anweisungen benutzt, das andere Mal eine *switch*-Anweisung:

```

<?php
if ($i == 0) {
    echo "i ist gleich 0";
} elseif ($i == 1) {
    echo "i ist gleich 1";
} elseif ($i == 2) {
    echo "i ist gleich 2";
}

switch ($i) {
case 0:
    echo "i ist gleich 0";
    break;
case 1:
    echo "i ist gleich 1";
    break;
case 2:
    echo "i ist gleich 2";
    break;
}
?>

```

Es ist wichtig, die Ausführung einer *switch*-Anweisung zu verstehen, um Fehler zu vermeiden. Die *switch*-Anweisung wird Zeile für Zeile (also Anweisung für Anweisung) abgearbeitet. Zu Beginn wird kein Code ausgeführt. Erst wenn bei einem *case*-Teil eine Entsprechung zum *switch*-Ausdruck vorliegt, werden die darin enthaltenen Anweisungen von PHP ausgeführt. PHP fährt dann mit der Abarbeitung des restlichen Codes innerhalb des *switch*-Blocks fort oder bis zum ersten Auftreten einer *break*-Anweisung. Ohne *break* am Ende eines *case*-Teils werden also noch die folgenden *case*-Blöcke ausgeführt. Zum Beispiel:

```

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

```

Wenn hier *\$i* gleich 0 ist, würde PHP alle *echo* Anweisungen ausführen! Ist *\$i* gleich 1, werden die letzten beiden *echo* Anweisungen ausgeführt. Nur wenn *\$i* gleich 2 ist, erhalten Sie das erwartete Ergebnis: die Ausgabe von "i ist gleich 2". Deshalb ist es wichtig *break*-Anweisungen zu setzen (abgesehen von bestimmten Fällen, in denen Sie diese mit Absicht weglassen).

Bei einer *switch*-Anweisung wird die Bedingung also nur einmal überprüft und das Ergebnis mit jeder *case*-Anweisung verglichen. Bei einer *elseif*-Anweisung

wird die Bedingung neu geprüft. Ist ihre Bedingung komplizierter als ein einfacher Vergleich und/oder in einer umfangreichen Schleife eingebettet, kann eine *switch*-Anweisung schneller sein.

Der Anweisungsteil von case kann auch leer sein. Dann wird die Kontrolle einfach an den nächsten case-Teil übergeben.

```
<?php
switch ($i) {
case 0:
case 1:
case 2:
    echo "i ist kleiner als 3 aber nicht negativ";
    break;
case 3:
    echo "i ist gleich 3";
}
?>
```

Ein Spezialfall ist *default*. Dieser Fall trifft auf alles zu, was nicht von den voranstehenden case-Ausdrücken erfasst wurde und sollte als letzte case Anweisung angegeben werden. Zum Beispiel:

```
<?php
switch ($i) {
case 0:
    echo "i ist gleich 0";
    break;
case 1:
    echo "i ist gleich 1";
    break;
case 2:
    echo "i ist gleich 2";
    break;
default:
    echo "i ist weder 0, 1 noch 2";
}
?>
```

Der *case*-Ausdruck kann eine Prüfung einfacher Typen sein, also von Integer- oder Fließkomma-Zahlen oder von Strings/Zeichenketten. Arrays oder Objekte können nicht benutzt werden, es sei denn, sie wurden in einfache Typen umgewandelt.

Die alternative Syntax der Kontrollstrukturen gilt auch für switch-Sequenzen. Mehr Informationen dazu erhalten Sie unter [Alternative Syntax für Kontrollstrukturen](#).

```
<?php
switch ($i):
case 0:
```

```
    echo "i ist gleich 0";
    break;
case 1:
    echo "i ist gleich 1";
    break;
case 2:
    echo "i ist gleich 2";
    break;
default:
    echo "i ist weder 0, 1 noch 2";
endswitch;
?>
```

declare

Das Sprachkonstrukt *declare* wird dazu verwendet, um Ausführungsdirektiven für einen Codeblock anzugeben. Die Schreibweise von *declare* ist der anderer Kontrollstrukturen ähnlich:

```
declare (Direktive) Anweisung
```

Die *Direktive* gibt Ihnen die Möglichkeit, das Verhalten des *declare*-Blocks zu bestimmen. Zur Zeit wird nur eine Direktive unterstützt: die *ticks* (Weiter unten finden Sie mehr Informationen zu den [ticks](#)).

Der *Anweisungsteil* des *declare*-Blocks wird ausgeführt - wie genau diese Ausführung passiert und welche Nebeneffekte während der Ausführung auftreten, hängt von der Direktive ab, die Sie im *directive*-Block angegeben haben.

Das *declare* Konstrukt kann auch im globalen Geltungsbereich benutzt werden und gilt dann für den folgenden Code.

```
<?php
// bewirkt das Gleiche:

// sie können declare so benutzen:
declare(ticks=1) {
    // hier folgt Ihr ganzes Skript
}

// oder wie hier:
declare(ticks=1);
// hier folgt Ihr ganzes Skript
?>
```

Ticks

Ein tick ist ein Ereignis, das bei jedem N -ten Auftreten der low-level Anweisungen innerhalb des *declare* Blocks, die vom Parser ausgeführt werden, auftritt. Der Wert von N wird durch die Angabe von *ticks=N* innerhalb des *declare*-Blocks in dem *directive* Abschnitt bestimmt.

Das Ereignis/die Ereignisse, die bei jedem tick eintreten, legen Sie mit der Funktion **register_tick_function()** fest. Weitere Einzelheiten können Sie dem Beispiel unten entnehmen. Beachten Sie, dass mehr als ein Ereignis für jeden tick eintreten kann.

Beispiel 16-1. Profil eines Bereichs von PHP Code

```
<?php
// Funktion, die bei Aufruf die Zeit aufzeichnet
function profile($dump = FALSE)
{
    static $profile;

    // Rückgabe der gespeicherten Zeit aus profile, danach löschen
    if ($dump) {
        $temp = $profile;
        unset ($profile);
        return ($temp);
    }

    $profile[] = microtime ();
}

// Einen tick handler bestimmen
register_tick_function("profile");

// Funktion vor dem declare-Block initialisieren
profile();

// Ausführen eines Code-Blocks, jede 2te Anweisung löst einen tick
aus

declare(ticks=2) {
    for ($x = 1; $x < 50; ++$x) {
        echo similar_text(md5($x), md5($x*$x)), "<br />";
    }
}

// Ausgabe der gespeicherten Daten aus dem Profiler
print_r(profile (TRUE));
?>
```

Dieses Beispiel 'profilert' den PHP Code der im 'declare'-Block steht, indem die Zeit festgehalten wird, zu der jede zweite low-level Anweisung im Codeblock ausgeführt wird. Diese Information können Sie dazu benutzen, langsame Bereiche innerhalb bestimmter Codesegmente zu identifizieren. Das gleiche Ziel können Sie auch mit anderen Methoden erreichen: die Benutzung von ticks ist bequemer und einfacher zu implementieren.

Ticks sind gut für Debugging, einfaches Multitasking, Hintergrund I/O und viele andere Aufgaben geeignet.

Siehe auch [register_tick_function\(\)](#) und [unregister_tick_function\(\)](#).

return

Wird die [return\(\)](#) Anweisung innerhalb einer Funktion aufgerufen, wird die Ausführung der Funktion sofort beendet und das Argument als Wert des Funktionsaufrufs zurückgegeben. [return\(\)](#) beendet auch die Ausführung einer [eval\(\)](#) Anweisung oder einer Skriptdatei.

Erfolgt der Aufruf innerhalb des globalen Bereichs, wird die Ausführung des aktuellen Skripts beendet. Wurde das aktuelle Skript [include\(\)](#)ed oder [require\(\)](#)ed, wird die Kontrolle an das aufrufende Skript zurückgegeben. Wurde das aktuelle Skript [include\(\)](#)ed, wird der Wert, der [return\(\)](#) zugewiesen wurde, als Wert des Aufrufs von [include\(\)](#) zurückgegeben. Wird [return\(\)](#) innerhalb des Hauptskripts aufgerufen, wird die Ausführung beendet. Handelt es sich bei dem Skript um eine Datei, die über die Einträge [auto_prepend_file](#) oder [auto_append_file](#) in der php.ini aufgerufen wurde, wird die Ausführung dieses Skripts beendet.

Weitere Informationen erhalten Sie im Abschnitt [Rückgabewerte](#).

Anmerkung: Beachten Sie, dass [return\(\)](#) ein Sprachkonstrukt und keine Funktion ist. Die Klammern um ein Argument sind deshalb *nur* zwingend notwendig, wenn es sich um einen Ausdruck handelt, dessen Ergebnis zurückgegeben werden soll. Es ist gebräuchlich die Klammern wegzulassen, wenn eine Variable zurück gegeben soll.

Kapitel 17. Funktionen

Inhaltsverzeichnis

[Vom Nutzer definierte Funktionen](#)

[Funktionsparameter](#)

[Rückgabewerte](#)

[old_function](#)

[Variablenfunktionen](#)

Vom Nutzer definierte Funktionen

Eine Funktion kann wie folgt definiert werden:

```
function foo ($arg_1, $arg_2, ..., $arg_n)
{
    echo "Beispielfunktion.\n";
    return $retval;
}
```

Jeder beliebige korrekte PHP-Code kann in einer Funktion vorkommen, sogar andere Funktionen und [Klassen](#)- Definitionen.

In PHP 3 müssen Funktionen definiert sein, bevor man auf sie verweist. In PHP4 ist das nicht mehr erforderlich, *außer* wenn eine Funktion nur bedingt definiert wird, wie in den beiden untenstehenden Beispielen gezeigt.

Wenn eine Funktion nur unter bestimmten Bedingungen definiert wird, muß die Definition dieser Funktion noch *vor* deren Aufruf abgearbeitet werden.

Beispiel 17-1. Bedingte Funktionen

```
<?php

$makefoo = true;

/* Wir können foo() von her aus nicht
   aufrufen, da sie noch nicht existiert,
   aber wir können bar() aufrufen */

bar();

if ($makefoo) {
    function foo ()
    {
        echo "Ich existiere nicht, bis mich die Programmausführung
        erreicht hat.\n";
    }
}

/* Nun können wir foo() sicher aufrufen,
   da $makefoo als true ausgewertet wurde */

if ($makefoo) foo();

function bar()
{
    echo "Ich existiere sofort nach Programmstart.\n";
}
```

?>

Beispiel 17-2. Funktionen innerhalb von Funktionen

```
<?php
function foo()
{
    function bar()
    {
        echo "Ich existiere nicht, bis foo() aufgerufen wurde.\n";
    }
}

/* Wir können bar() noch nicht
   aufrufen, da es nicht existiert */

foo();

/* Nun können wir auch bar() aufrufen,
   da sie durch die Abarbeitung von
   foo() verfügbar gemacht wurde */

bar();

?>
```

PHP unterstützt weder das Überladen von Funktionen, noch ist es möglich, zuvor deklarierte Funktionen neu zu definieren oder die Definition zu löschen.

PHP 3 unterstützt keine variable Anzahl von Parametern, obwohl Vorgabewerte für Parameter unterstützt werden (weitere Informationen finden Sie unter [Vorgabewerte für Parameter](#)). PHP 4 unterstützt beides: siehe [Variable Parameteranzahl](#) und die Funktionsreferenzen für [func_num_args\(\)](#), [func_get_arg\(\)](#) und [func_get_args\(\)](#) für weitere Informationen.

Funktionsparameter

Mit einer Parameterliste kann man Informationen an eine Funktion übergeben. Die Parameterliste ist eine durch Kommas getrennte Liste von Variablen und/oder Konstanten.

PHP unterstützt die Weitergabe von Parametern als Werte (das ist der Standard), als [Verweise](#), und als [Vorgabewerte](#). Die Übergabe einer variablen Anzahl von Parametern wird nur von PHP 4 und höher unterstützt, siehe [Variable Parameteranzahl](#) und die Funktionsreferenzen für [func_num_args\(\)](#), [func_get_arg\(\)](#) und [func_get_args\(\)](#) für weitere Informationen. Durch die Übergabe eines Arrays mit Parametern kann man auch in PHP 3 einen ähnlichen

Effekt erreichen:

```
function rechne_array($eingabe)
{
    echo "$eingabe[0] + $eingabe[1] = ", $eingabe[0]+$eingabe[1];
}
```

Verweise als Parameter übergeben

Normalerweise werden den Funktionen Werte als Parameter übermittelt. Wenn man den Wert dieses Parameters innerhalb der Funktion ändert, bleibt der Parameter außerhalb der Funktion unverändert. Wollen Sie aber genau das erreichen, dann müssen Sie die Parameter als Verweise (Referenzen) übergeben.

Wenn eine Funktion einen Parameter generell als Verweis behandeln soll, setzt man in der Funktionsdefinition ein kaufmännisches Und (&) vor den Parameternamen:

```
function fuege_etwas_anderes_an (&$string)
{
    $string .= 'und nun zu etwas anderem.';
}
$str = 'Dies ist ein String, ';
fuege_etwas_anderes_an ($str);
echo $str; // Ausgabe: 'Dies ist ein String, und nun zu etwas anderem.'
```

Vorgabewerte für Parameter

Eine Funktion kann C++-artige Vorgabewerte für skalare Parameter wie folgt definieren:

```
function machkaffee ($typ = "Cappucino")
{
    return "Ich mache eine Tasse $typ.\n";
}
echo machkaffee ();
echo machkaffee ("Espresso");
```

Die Ausgabe von diesem kleinen Skript ist:

```
Ich mache eine Tasse Cappucino.
Ich mache eine Tasse Espresso.
```

Der Vorgabewert muss ein konstanter Ausdruck sein, darf also (zum Beispiel) keine Variable oder Element einer Klasse sein.

Bitte beachten Sie, dass alle Vorgabewerte rechts von den Nicht-Vorgabeparametern stehen sollten; - sonst wird es nicht funktionieren. Betrachten Sie folgendes Beispiel:

```
function mach_joghurt ($typ = "rechtsdrehendes", $geschmack)
{
    return "Mache einen Becher $typ $geschmack-joghurt.\n";
}

echo mach_joghurt ("Brombeer"); // arbeitet nicht wie erwartet
```

Die Ausgabe dieses Beispiels ist::

```
Warning: Missing argument 2 in call to makeyogurt() in
/usr/local/etc/httpd/htdocs/php3test/functest.html on line 41
Mache einen Becher Brombeer-joghurt.
```

Nun vergleichen Sie bitte oberes Beispiel mit folgendem:

```
function mach_joghurt ($geschmack, $typ = "rechtsdrehendes")
{
    return "Mache einen Becher $typ $geschmack-joghurt.\n";
}

echo mach_joghurt ("Brombeer"); // arbeitet wie erwartet.
```

... und jetzt ist die Ausgabe:

```
Mache einen Becher rechtsdrehendes Brombeer-Joghurt.
```

Variable Anzahl von Parametern

PHP 4 unterstützt eine variable Anzahl von Parametern in benutzerdefinierten Funktionen. Das Handling dieser Parameter fällt mittels der Funktionen [func_num_args\(\)](#), [func_get_arg\(\)](#) und [func_get_args\(\)](#) sehr leicht.

Es ist keine spezielle Syntax erforderlich. Die Parameter können wie gehabt explizit in den Funktionsdeklarationen angegeben werden, und werden sich wie gewohnt verhalten.

Rückgabewerte

Sie können Werte mit dem optionalen Befehl "return" zurückgeben. Es können Variablen jedes Typs zurückgegeben werden, auch Listen oder Objekte. Die beendet sofort die Funktion, und die Kontrolle wird wieder an die aufrufende Zeile zurückgegeben. Weitere Informationen finden Sie unter [return\(\)](#).

```
function quadrat ($zahl)
{
    return $zahl * $zahl;
}

echo quadrat (4); // gibt '16' aus.
```

Es ist nicht möglich, mehrere Werte von einer Funktion zurückzugeben. Ein ähnliches Resultat kann man aber durch die Rückgabe von Listen erreichen.

```
function kleine_zahlen()
{
    return array (0, 1, 2);
}
list ($null, $eins, $zwei) = kleine_zahlen();
```

Um von einer Funktion eine Referenz zurückzugeben, müssen Sie den Referenz-Operator & sowohl in der Funktionsdeklaration, als auch bei der Zuweisung des zurückgegebenen Wertes verwenden:

```
function &returniere_referenz()
{
    return $einereferenz;
}

$neuereferenz =& returniere_referenz();
```

Weitere Informationen über Referenzen finden Sie im Kapitel [Referenzen in PHP](#).

Variablenfunktionen

PHP unterstützt das Konzept der Variablenfunktionen. Wenn Sie an das Ende einer Variablen Klammern hängen, versucht PHP eine Funktion aufzurufen, deren Name der aktuelle Wert der Variable ist. Dies kann unter anderem für Callbacks, Funktionstabellen, usw. genutzt werden.

Variablenfunktionen funktionieren nicht mit Sprachkonstrukten wie [echo\(\)](#), [print\(\)](#), [unset\(\)](#), [isset\(\)](#), [empty\(\)](#), [include\(\)](#) und [require\(\)](#). Sie müssen Ihre eigenen Wrapperfunktionen verwenden, um diese Konstrukte als variable Funktionen benutzen zu können.

Beispiel 17-3. Beispiel für Variablenfunktionen

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

function bar($arg = '')
{
    echo "In bar(); der Parameter ist '$arg'.<br>\n";
}

// Dies ist eine Wrapperfunktion für echo
function echoit($string)
```

```

{
    echo $string;
}

$func = 'foo';
$func();      // Dies ruft foo() auf

$func = 'bar';
$func('test'); // Dies ruft bar() auf

$func = 'echoit';
$func('test'); // Dies ruft echoit() auf
?>

```

Sie können auch die Methode eines Objektes mittels der variablen Funktionen aufrufen.

Beispiel 17-4. Variable Methode

```

<?php
class Foo
{
    function Var()
    {
        $name = 'Bar';
        $this->$name(); // Dies ruft die Bar() Methode auf
    }

    function Bar()
    {
        echo "Das ist Bar";
    }
}

$foo = new Foo();
$funcname = "Var";
$foo->$funcname(); // Dies ruft $foo->Var() auf

?>

```

Siehe auch [call_user_func\(\)](#), [Variable Variablen](#) und [function_exists\(\)](#).